Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Προηγμένα Συστήματα Πληροφορικής»

**Μεταπτυχιακή Διατριβή**

| | |
|---|---|
| Τίτλος Διατριβής | **PTLsim x86 architectural simulator: extending the data and instruction prefetching subsystem** |
| Ονοματεπώνυμο Φοιτητή | **Αθανάσιος Χατζηδημητρίου** |
| Πατρώνυμο | **Θεόδωρος** |
| Αριθμός Μητρώου | **ΜΠΣΠ/ 11056** |
| Επιβλέπων | **Δημήτριος Γκιζόπουλος, Αναπληρωτής Καθηγητής** |

Ημερομηνία Παράδοσης **Σεπτέμβριος 2013**

**Τριμελής Εξεταστική Επιτροπή**

| (υπογραφή) | (υπογραφή) | (υπογραφή) |
|---|---|---|
| Δημήτρης Γκιζόπουλος<br>Αναπληρωτής Καθηγητής<br>ΕΚΠΑ | Μιχάλης Ψαράκης<br>Επίκουρος Καθηγητής<br>ΠΑΠΕΙ | Άγγελος Πικράκης<br>Λέκτορας<br>ΠΑΠΕΙ |

## Abstract

Primary goal of existing x86 architectural simulation tools is to provide a working model that accurately resembles most of the mainstream implementation techniques. Lower priority is given to performance units, which in contrast, greatly interest the industry. We expanded PTLsim architectural simulator by integrating two hardware prefetcher components for data and instructions, respectively, and by adding more detail in two critical performance-related parts of the microprocessor, the cache hierarchy and branch prediction unit. All our modifications resemble commercial microprocessor implementations and the performance measurement results correspond to the design estimates. Our expanded PTLSim version with the integrated prefetching sub-system can achieve a total of 4-5% performance boost compared to the initial design.

## Περίληψη

Κύριος στόχος των προσομοιωτών αρχιτεκτονικής x86 είναι να παρέχουν ένα λειτουργικό μοντέλο που προσεγγίζει τις πιο διαδεδομένες τεχνικές υλοποίησης. Μικρότερη προτεραιότητα δίνεται σε μονάδες απόδοσης, οι οποίες όμως ενδιαφέρουν σημαντικά την βιομηχανία. Επεκτείναμε τον αρχιτεκτονικό προσομοιωτή PTLsim με την προσθήκη δύο μονάδων προ-ανάκλησης (prefetch), για δεδομένα και εντολές αντίστοιχα και αυξήσαμε την λεπτομέρεια σε δύο τμήματα του μοντέλου του επεξεργαστή που σχετίζονται άμεσα με την απόδοση, την ιεραρχία προσωρινής μνήμης (cache hierarchy) και την μονάδα πρόβλεψης διακλάδωσης. Όλες οι τροποποιήσεις βασίζονται σε εμπορικές υλοποιήσεις επεξεργαστών και τα αποτελέσματα τους επαληθεύουν τις προβλέψεις της σχεδίασης. Η επεκταμένη έκδοση του PTLsim που περιέχει το υποσύστημα προ-ανάκλησης (prefetching subsystem) μπορεί να πετύχει συνολική αύξηση απόδοσης της τάξης του 4-5% συγκριτικά με την αρχική σχεδίαση.

## Contents


Abstract ..... 3
Περίληψη ..... 3
Contents ..... 4
Tables ..... 5
Figures ..... 6
1 PTLsim x86 architectural simulator: Expanding the prefetching sub-system ..... 7
2 PTLsim architectural simulator ..... 10
2.1 Overview ..... 10
2.2 x86 Instructions, translation and micro-ops ..... 10
2.3 Out-of-Order core ..... 11
2.4 Cache hierarchy and load-store uops ..... 13
3 Component integrations ..... 15
3.1 Data and Instruction prefetcher components ..... 16
3.1.1 Prefetching schemes ..... 16
3.1.2 Data Prefetcher ..... 18
3.1.3 Instruction Prefetcher ..... 19
3.1.4 Prefetch requests and design summary ..... 20
3.2 Single tag port emulation mechanism ..... 21
3.2.1 Contention and the arbitration unit ..... 21
3.3 Validation ..... 23
4 Modifications – Design changes and fixes ..... 28
4.1 Miss-address buffer ..... 28
4.1.1 Coupling dedicated MABs to cache memories ..... 28
4.1.2 Prioritization inside the buffer ..... 30
4.2 Branch prediction unit ..... 31
4.3 Out-of-Order core modifications ..... 35
4.3.1 Adaptation to the updated memory system ..... 35
4.3.2 Adaptation to the updated BPU ..... 37
5 Experimental results ..... 39
5.1 Configuration and experimental models ..... 39
5.2 Simulation results ..... 42
5.3 Expanded 10-billion simulation results ..... 52
5.4 Design evolution through simulation results ..... 54
6 Conclusions ..... 59
7 References ..... 60
Appendix A - Miscellaneous simulator modifications ..... 61
I. Commit-store bug ..... 61
II. Statistics tree and new command line options ..... 62

## Tables

Table 1 Data prefetcher configuration ..... 24
Table 2 Validation program 1 simulation results ..... 24
Table 3 Validation program 1 - prefetcher statistics ..... 25
Table 4 Validation program 2 simulation results ..... 25
Table 5 Validation program 3 simulation results ..... 26
Table 6 Validation program 3 prefetcher statistics ..... 26
Table 7 Validation program 4 simulation results ..... 27
Table 8 Validation program 4 prefetcher statistics ..... 27
Table 9 List of benchmark programs ..... 39
Table 10 Golden configuration model ..... 40
Table 11 BPU settings configuration model ..... 40
Table 12 Memory system configuration model ..... 40
Table 13 Experimental configuration model ..... 41
Table 14 Average IPC ..... 44
Table 15 Average prefetch request hit distribution ..... 51
Table 16 Average data cache miss rate ..... 52
Table 17 Average instruction cache miss rate ..... 52

## Figures

Figure 1 Categorization of architectural simulators ..... 7
Figure 2 a) Prefetch-on-miss b) Tagged prefetching ..... 17
Figure 3 Data prefetcher layout ..... 19
Figure 4 Instruction prefetcher layout ..... 20
Figure 5 Prefetching subsystem layout ..... 20
Figure 6 Arbiter block diagram ..... 23
Figure 7 Memory system block diagram ..... 29
Figure 8 BPU block diagram ..... 33
Figure 9 Conditional Branches data flow ..... 34
Figure 10 Indirect branches data flow ..... 35
Figure 11: IPC graph for Spec2000 benchmark suite using ref dataset ..... 42
Figure 12: IPC graph for Spec2006 benchmark suite using ref dataset ..... 42
Figure 13: IPC graph for Spec2000 benchmark suite using test dataset ..... 43
Figure 14: IPC graph for Spec2006 benchmark suite using test dataset ..... 43
Figure 15: Deviation between Baseline and Experimental models for Spec2000 using ref dataset ..... 44
Figure 16: Deviation between Baseline and Experimental models for Spec2006 using ref dataset ..... 44
Figure 17: Deviation between Baseline and Experimental models for Spec2000 using test dataset ..... 45
Figure 18: Deviation between Baseline and Experimental models for Spec2000 using test dataset ..... 45
Figure 19: Data prefetcher accuracy graph for Spec2000 benchmark suite using ref dataset ..... 46
Figure 20: Data prefetcher accuracy graph for Spec2006 benchmark suite using ref dataset ..... 46
Figure 21: Data prefetcher accuracy graph for Spec2000 benchmark suite using test dataset ..... 47
Figure 22: Data prefetcher accuracy graph for Spec2006 benchmark suite using test dataset ..... 47
Figure 23: Data prefetcher coverage graph for Spec2000 benchmark suite using ref dataset ..... 48
Figure 24: Data prefetcher coverage graph for Spec2006 benchmark suite using ref dataset ..... 48
Figure 25: Data prefetcher coverage graph for Spec2000 benchmark suite using test dataset ..... 49
Figure 26: Data prefetcher coverage graph for Spec2006 benchmark suite using test dataset ..... 49
Figure 27: Prefetch request sources for Spec2000 benchmark suite using ref dataset ..... 50
Figure 28: Prefetch request sources for Spec2006 benchmark suite using ref dataset ..... 50
Figure 29: Prefetch request sources for Spec2000 benchmark suite using test dataset ..... 51
Figure 30: Prefetch request sources for Spec2006 benchmark suite using test dataset ..... 51
Figure 31 IPC for Spec2000 10-billion x86 commit simulation ..... 53
Figure 32 IPC for spec2006 10-billion x86 commit simulation ..... 53
Figure 33 Deviation for Spec2000 10-billion x86 commits simulation ..... 54
Figure 34 Deviation for Spec2006 10-billion x86 commits simulation ..... 54
Figure 35 Initial design stage simulation results ..... 55
Figure 36 Stage 2 Deviation results for Spec2006 and ref dataset ..... 56
Figure 37 Stage 2 results for spec2006 ref dataset, after increasing L1 data cache MAB size ..... 56
Figure 38 Stage 3 results for spec2006 ref dataset after complete prioritization – fix of wrf's performance degradation ..... 57

# 1 PTLsim x86 architectural simulator: Expanding the prefetching sub-system

Simulation is the imitative representation of the functionality and timing of a real-world process or system over time [ 1]. It is used to find a cause of a past occurrence or forecast future effects of assumed circumstances without exposure to risk. Alan Turing used the term to refer on what happens when a universal Turing machine executes a state transition table that describes both state transitions and input/output. Simulation is widely used in many fields of computer science for study and analysis of a variety of complex systems. Results are often used for validation of correctness and debugging, drawing of conclusions and detailed performance metrics for research purposes and evaluation of alternative designs without actually building physical systems. Accuracy of simulation models provides significant support to the design of correct and efficient computing systems.

In computer architecture, a simulator is a computer program that performs a simulation of another subject-computer's operation [ 1]. This simulation usually includes the generation of input/output signals and statistical data concerning the functionality and performance of the subject machine (execution time, power consumption etc.). The subject can vary from a single microprocessor to a complete computer system, including memory system, microprocessor, I/O devices etc. Depending on their purpose and level of detail, architectural simulators can be categorized as functional and performance simulators. Functional simulators emphasize on achieving the same function as the subject machine, while performance simulators aim to accurately reproduce the performance behavior, in addition to functionality. Therefore, performance simulators usually maintain a more detailed description of the subject machine and in most cases generate much more detailed results. Performance simulators are divided to cycle accurate simulators and instruction schedulers. The first category simulates the microarchitecture of a processor on a cycle-by-cycle basis, maintaining data for representing the state of all components at any time during the simulation, while instruction schedulers only simulate an instruction set architecture and lack the technical details of specific implementations. Instruction schedulers (or instruction set) simulators are usually faster but are not capable to accurately emulate implementation techniques that exist below the architectural level (such as pipelining, out of order execution etc.). They are often used for emulating older hardware and when timing information of the modeled design is not required.

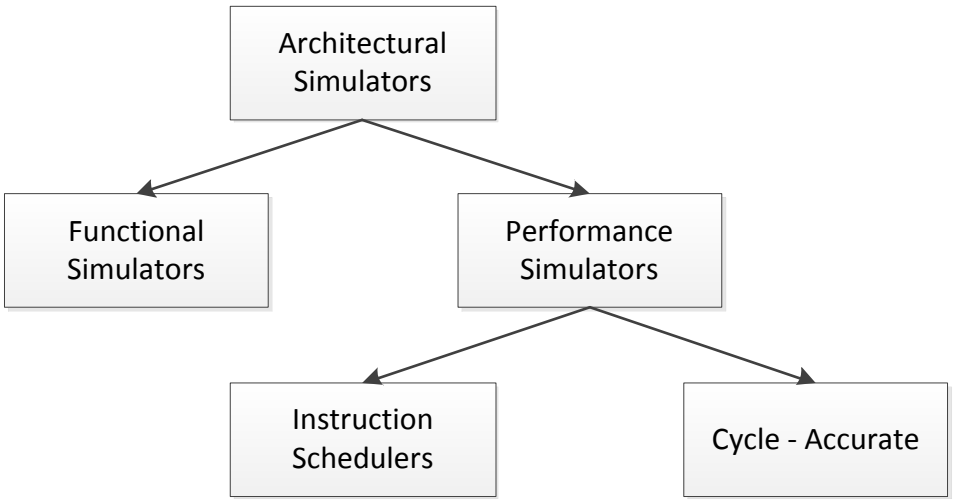


Figure 1 Categorization of architectural simulators

Cycle accurate simulators constitute the category of architectural simulators with the greatest accuracy. More specifically, they emulate a computer system on microarchitecture level. By the term "microarchitecture" we refer to the way a given instruction set architecture is implemented on a processor. This level of detail gives the ability to accurately model a processor design exactly as implemented, taking into account the timing information of possible pipelining stages, multiple functional units, inter-cluster communication bandwidths, micro-operations, out-of-order models, non-blocking memories and other similar details that may exist under the layer of instruction set architecture. Depending on the required accuracy of results, the detail of the subject model may vary.

To highlight the importance of simulation software, it is enough to take a look on their applications. Simulators are widely used in several scientific fields, including computer architecture and many research fields actually rely on them. The growing urge of accurate tools has led to a lot of study and research around simulation techniques. As a result we have a constant improvement of existing simulation tools and development of many new.

Several simulation tools exist today with a wide range of design options and almost every possible major implementation technique being available. Most of these tools however usually concern RISC architectures which are much simpler to model and are employed typically in embedded and low-power application fields. Widely used commercial CISC architectures generally lack decent publicly available simulation tools. A good example is the IA32 (x86) which is the today's most common architecture in personal computers, servers and workstations. The high complexity of modern implementations has been a discouraging factor for the development of cycle-accurate simulators and despite their publicity, only a few tools exist to serve this purpose. All microprocessor vendors of x86 machines rely on their internal, accurate but not publicly available tool sets.

PTLsim is one of the existing x86/x64 cycle-accurate architectural simulators. It implements a custom processor model, which is a combination of several modern commercial implementations. The design is placed chronologically close to the Intel Core duo generation. Its primary focus is to supply a working simulation model that includes all the mainstream implementation techniques (pipelining, out-of-order execution, cache hierarchy, branch prediction etc.) for statistical analysis and experimentation. Even though it does include some aggressive performance enhancing components, it seems that the actual performance boost of the processor design was not considered of a high interest. Therefore, there is a significant gap between the PTLsim processor design and realistic modern industrial x86 processors, regarding performance units. This gap is identified by the lack of several details in the modeled hardware units and extends to the lack of certain important performance components inside the processor.

The goal of this master thesis is to develop an enhance version of the PTLsim simulator and aims to reduce this modeling gap and make an already useful tool into a state-of-the-art one. The enhancement includes integration of missing performance components (such as data and instruction hardware prefetchers) and modifications to existing performance-related parts of the design. Following the simulator's philosophy, all of the additions / modifications resemble modern commercial implementations

The integration of prefetcher units report a performance boost close to the expected 4-5%, based on the description of the extra components that we modeled. Along with the rest of the modifications, both in the memory system and the branch prediction unit, our custom version has a significantly

different behavior than the original PTLsim and can be used for experimentation focusing on the performance units of the processor.

# 2 PTLsim architectural simulator

## 2.1 Overview

PTLsim is an open source, cycle-accurate, x86 microprocessor simulator for the IA-32 (x86) instruction set and its 64-bit extension (x86-64) [ 2], designed and developed by Matt T. Yourst. It resembles a modern superscalar, out-of-order, x86-64 compatible processor core model and is highly configurable on the level of detail, ranging from full-speed native execution on the host CPU all the way down to RTL level models of all key pipeline structures. PTLsim runs directly on the same platform it is simulating, having the ability to switch in and out of full out of order simulation mode and native x86 or x86-64 mode at any time completely transparent to the user code being executed. In order to achieve high performance it has undergone through extensive tuning, cache profiling and use of x86-specific accelerated vector operations and instructions. As a result, it significantly reduces simulation time compared to traditional research simulators.

In addition to the microprocessor design, PTLsim models all microcode, the complete cache hierarchy, memory subsystem and supporting hardware devices in order to achieve higher precision by accurately simulating a full computing system. It was designed to run on hosts with Linux operating system and comes in two flavors: Classic userspace in Linux (as a single threaded Linux application) and Xen hypervisor mode for a full system mode (offering SMT and multi core capabilities).

PTLsim allows a significant amount of flexibility for easy experimentation through the use of optimized C++ template classes and libraries suited to synchronous logic design. It is used extensively at hundreds of major universities, industry research labs and the well-known x86 microprocessor vendors Intel and AMD. It is currently among the very few publicly available tools that support true cycle accurate modeling of real x86 microarchitectures, providing a new option for researchers to use a simulation tool for a contemporary and widely used instruction set with readily available hardware implementations.

## 2.2 x86 Instructions, translation and micro-ops

PTLsim is capable of handling a full implementation of x86 and x86-64 instruction set, including most user and kernel instructions supported by the Intel Pentium 4 and AMD K8 processors (extensions such as MMX, SSE/SSE2, x87 FP etc.).

Like most modern x86 microprocessors, PTLsim does not directly execute x86 instructions; it uses its own microcode instead. It is a common technique to translate complex x86 instructions to lower level microcode routines with equivalent effect in order to simplify the design of the CPU. This method has proven much more efficient, especially in out-of-order models. Microcode routines are based on series of micro-operations (micro-ops or uops) which are very similar to classical load-store RISC instructions, unlike the original x86 instruction set which is based on a two operand CISC concept of load-and-compute and load-compute-store. Micro-ops are below the system architecture and are part of the microprocessor implementation. PTLsim also adopts this concept and has its own version of microcode implemented.

According to the PTLsim model, micro-ops can be found in two states: uops and transops. Each micro-op has three source registers and one destination register. The term “transops” stands for “translated uops” and is used to refer to micro-ops that have just been translated and have not reached the renaming stage of the OoO pipeline. Transops’ sources and destination have not yet been

assigned to architectural registers (they are represented as un-renamed architectural registers). On the other hand, micro-ops (μops or uops) have already passed through the renaming stage and their sources and destination have been assigned to architectural registers.

The processor has the responsibility to translate x86 instructions to equivalent microcode. This process requires some complex logic itself and on a simulation environment, it is quite time consuming. However, there is no real reason (performance-wise) to repeat the translation and spend extra time for already translated sequences of instructions on a simulator. In order to achieve high performance, PTLsim maintains a basic block cache (BB cache) to accelerate this translation stage. This cache contains sequences of translated uops in program order that correspond to original basic blocks in the program. This way, program code that has been used in the recent past can be used again directly on its translated version. Each basic block is identified by the Program Counter (Register Instruction Pointer – RIP) of the basic block's entry point (this applies in a userspace simulation where only a single address space is used). Basic blocks are terminated by either a control flow operation or a barrier operation.

Uops are the actual operations executed by the processor core. Each uop has its own implementation and in many cases, the simulated core uses a real x86 instruction on the host machine corresponding to uop's operation and captures the generated flags, rather than manually emulate the same condition codes. In PTLSim, translation of x86 instructions is independent of the actual processor core and maintains its own statistics tree (short reference on PTLsim's stats collector model can be found at Appendix A - II). However, the delay for decoding instructions is placed inside the core and is being simulated at the pipeline's front end.

## 2.3 Out-of-Order core

PTLsim comes with two different machine models, a sequential in-order core (by the name "seq") and an out-of-order core ("ooo"). Other models can be easily added by users through a well-structured hierarchy tree. In this subsection we will make a brief reference on the built-in out-of-order core.

The built-in "ooo" machine is an x86 processor core which implements an out-of-order execution pipeline. Out of order execution is an approach that efficiently uses instruction cycles and reduces costly delays. The basic concept is to complete the processing of instructions that have all their operands available in different order to fill idle cycles caused by memory or other dependency delays. It is widely used in high performance microprocessors and requires multiple functional units to allow parallel execution of basic functions. PTLsim completely models a modern out of order x86 compatible processor with true cycle accurate simulation. The microarchitecture adopts features from Intel Pentium 4, AMD K8 and Intel Core 2 but also incorporates some ideas from IBM Power4/Power5 and Alpha EV8.

### *Pipeline stages*

The PTLsim ooo core implements the following pipeline stages:

- Instruction fetch – Fetch a stream of x86 instructions from the L1 I-cache along predicted branch paths.
- Decode – Decoding and translation of x86 instructions to micro-ops
- Rename – Allocate and rename stage, allocates Reorder-Buffer entries and physical resources for a uop.

- Dispatch (and cluster selection) – locates the source operands for a uop and adds entries to the issue queues.
- Issue (and execution) – Issues and executes a uop.
- Writeback – Write results back.
- Commit – Commit stage.

### *Instruction fetching*

Ooo core fetches sequences of uops (transops at this point) that have been previously translated or loaded through the Basic Block Cache, as described in the previous subsection. Despite the fact that instructions are fetched by the BBcache, PTLsim emulates delays as if instructions were actually being fetched by the L1 instruction cache.

Fetched transops are initially placed inside a fetching buffer where additional information is attached, such as instruction size, corresponding x86 instruction virtual address, start or end of x86 instruction (for instructions that have been translated to multiple transops) etc. Then, they are placed inside the Fetch queue and are ready to go to renaming stage. For simulation, statistical and debugging purposes, each transop is identified by a unique id number (uuid). When a control flow instruction is fetched, branch prediction mechanisms are triggered and indicate whether the fetching should continue sequentially or be redirect to another memory address. A more detailed reference on branch prediction can be found at section 4.2. Up to four uops can be fetched at each clock-cycle; this number is configurable. Fetching stage also defines several states for stalling in order to cover possible cases during execution.

### *Reorder Buffer*

Every out-of-order core that uses Tomasulo-like algorithm maintains a Reorder Buffer (ROB) structure so that it can track and commit instructions in program order. The ROB is treated as a queue where entries are allocated from the tail and committed from the head. During rename and allocate stage, uops from the fetch queue are examined for their resource dependencies and are placed in ROB slots. If any of the resources is not available (ROB slots, physical registers), the renaming stage is stalled. Every ROB entry maintains a variety of information regarding each uop, including pointers to physical registers, counters, states of the uop, etc. In addition, all the information attached from the fetching stage remains intact. Uops are tracked inside the pipeline through their ROB slot.

### *Functional units clusters and execution*

PTLsim organizes functional units into clusters. This gives the ability to implement several features of modern microprocessors. Each ROB entry is routed into a specific cluster by the time it is dispatched; uops can't switch clusters after that. Selection of the destination cluster for each uop comes after evaluation the available choices, with heuristic methods that take into account various runtime information, such as availability and dependencies. Only clusters that are capable to serve the specific uop participate on this evaluation (for instance, a floating point instruction cannot be routed into integer clusters with no FP units). This cluster organization offers several features (like multiple functional units) and can be highly customized. Clusters themselves along with all Inter-cluster communication bandwidths and delays (regarding forwarding) are fully configurable by the simulator user.

Each cluster maintains its own issue queue and is capable of issuing a specific (configurable) number of instructions per clock-cycle. When a uop is dispatched, it is placed at the end of the issue queue for its cluster and several associative arrays are updated to indicate which operands it is still waiting for. When a uop finishes its execution, it broadcasts its result so that it will be filled in all uops that depend on it. Uops that have all their operands ready update a flag on a ready-to-issue bitmap (by the name "allready"). On each clock-cycle, the first valid bit inside the ready-to-issue bitmap is selected (corresponding to the oldest ready (in-order) instruction) and the uop is issued. PTLsim clusters are capable of multiple-issue and more than one uops can be issued on the same cycle by this way. The processor then selects a functional unit for the uop and executes it. At this point and in order to simplify the implementation, the result is calculated and put directly into the assigned physical register since the beginning, but it is not used until the actual execution time passes. After execution, the corresponding slot is released for ready-to-dispatch instructions. Uops that were issued but turn out that they could not actually complete, must be "replayed". This means that they are turned back to the initial dispatched state and remain at that state until all their operands are available again. Additional dependencies on events can be set in such cases.

### *Misspeculation and recovery*

PTLsim implements three methods to handle misspeculation cases:

- Replay – Already mentioned in the previous subsections, replay takes issued uops and turns them back to the initial dispatched state inside the issue queue. It is used for scheduling and dependency mispredictions only.
- Redispatch – It is used for load-store aliasing recovery, value mispredictions and cases where the uops are valid but their outputs are not. It finds all ROB entries that depend on a mis-speculated uop and turns them back to ready-to-dispatch state so that they can be dispatched and issued again with the correct operands and produce correct results.
- Annulment – It is used for annulling any uops that were mistakenly fetched inside the pipeline, in cases of branch mispredictions and misalignment recovery.

In addition, PTLsim model implements a mechanism that monitors the core for possible deadlocks after redispatch recovery. It is triggered when no uops have been dispatched for 64 cycles and forces a pipeline flush. These three methods provide different levels of recovery and can effectively reduce misspeculation costs to the actual required. With lower misspeculation costs, more aggressive techniques can be used (on uop issue stage) and further performance improvement can be gained.

## 2.4 Cache hierarchy and load-store uops

The cache memory hierarchy in PTLsim consists of four levels by default:

- L1 data cache
- L1 instruction cache
- L2 cache
- L3 cache
- Main memory

The main memory is considered of infinite size and only cache memories have configurable sizes. The hierarchy is assumed inclusive[1] and all levels are write-through[2]. Cache memories have a highly configurable set-associative array structure which does not store any program data; the host machine's memory is used to hold the actual data instead. The cache structures only maintain information about which addresses are valid or not. Detailed reference on the cache memory implementation can be found at Appendix A - I.

PTLsim implements a non-blocking[3] memory model, with caches capable of serving requests upon multiple cache-misses. Therefore, the subsystem also implements structures such as Miss Address[3] buffers and Load-Fill-Request[3] Queue.

When a load instruction is issued, it goes through several steps before reaching the actual cache memory. The address generation unit calculates its physical address and then, the load-store queue is examined backwards to see whether this load corresponds to a store. It is possible that a prior store has not successfully resolved its address and a possible dependency is not discovered at this point, causing a load-store aliasing which will lead to a redispatch misspeculation recovery when the aliasing is identified. We leave the discussion of this issue for later in the text. If the corresponding address cannot be served internally by the core, the cache subsystem is checked. L1 cache is probed and if the address is valid, the load is served and execution completes normally. In contrast, on an L1-cache miss, a new Load-Fill-Request[3] queue entry is allocated and the ROB entry (uop) enters an idle state, waiting for a wake-up event. The corresponding address is searched in lower levels of the memory hierarchy and it is served by the highest memory level that holds the referenced address. All memory access latencies are simulated and when the referenced line becomes available in L1 cache, a wake-up event occurs, the outstanding miss is filled and the load instruction continues its execution.

[1] Inclusive cache hierarchy states that all data inside higher memory levels is also present at all lower memory levels
[2] Store instructions write their data directly in all memory levels instead of waiting for dirty lines to be evicted
[3] Brief reference at section 4.1

# 3 Component integrations

Over the years, the increase in the rate of microprocessor clocks by far exceeds the one in main memory dynamic RAM. The main reason is the different directions in the development of technology between microprocessor and memory industries which primarily aimed at capacity, rather than speed. The continuously increasing difference on the performance between microprocessors and main memory is a major drawback in computer performance. In order to fill this gap, the use of new aggressive techniques designed to reduce or hide memory latencies becomes necessary [ 3].

The most common technique for reducing memory access penalties and improve the throughput of the memory system is the use of cache memory hierarchies [ 4]. Keeping frequently used items in faster and more expensive (Static RAM – SRAM) memory levels close to the processor effectively reduces the average memory access penalty that is met in the use of main memory DRAM. Programs with high degree of locality in their memory access patterns receive more benefits from a cache hierarchy. However, most data-intensive programs still spend a large portion of their run time stalled on memory requests [ 5] and at the same time report poor cache memory utilization, which indicates that there is room for further improvement.

One approach that effectively reduces the negative performance impact of low cache memory utilization is prefetching [ 6]. Its applications are continuously increasing and it is widely used for performance boost on both software and hardware level. The idea is to fetch items in higher memory levels before the processor actually requests them. Idle memory time is used for prefetching, increasing utilization and decreasing memory stalls. This can apply to both data and instructions and it requires non-blocking[4] memories [ 7] as the ones modeled by PTLsim. In order to benefit from the technique, the processor must be able to keep operating while data is requested by the memory.

Apart from the benefits however, on a real-life implementation, prefetching can significantly increase memory traffic, a fact that can have a negative impact on performance. Improper use can lead to performance degradation caused by the increased competition in the use of resources and cache pollution. By the term “cache pollution” we refer to the situation where “non-useful” (= are not going to be requested) items are fetched to higher levels in the memory hierarchy, displacing useful content of the cache memory. This is a possible scenario, as prefetching schemes rely on prediction of future memory accesses and such predictions may be frequently wrong in certain applications.

Prefetching can be controlled by two sources: software and hardware; they are meant to work alongside and complement each other.

Software controlled prefetching is a directed method where the user (or perhaps a compiler) indicates what and when should be prefetched through architecture specific prefetch instructions. Software prefetching is pre-defined and static. Decisions about prefetching are taken at the design stage and this optimization requires user or compiler intervention. Prefetch instructions can add some execution overhead. Proper use of software prefetching can significantly improve execution performance [ 8] and usually applied during software optimization stage [ 5].

Hardware controlled prefetching on the other hand does not require changes to existing executables [ 9], since the directions for prefetch requests are given below the architectural level and are invisible for the program. This means that it can apply even to legacy applications without any further modification or re-building. Another advantage is that it uses run-time information in order to

[4] Brief reference to non-blocking memories can be found under section 4.1 Miss-address buffer

perform prefetches and this is a feature that makes hardware prefetching more flexible. This flexibility allows hardware prefetchers to implement different algorithms, with varying aggressiveness and prefetching patterns and to achieve significant performance gains. A hardware component that implements prefetching logic is called Hardware Prefetcher and is often referred as Data or Instruction Prefetcher, depending on its application to the data or instruction stream of an application.

Hardware prefetchers are very common nowadays and can be found integrated in most modern CISC processor microarchitectures (such as Intel Xeon, Core, AMD Opteron, Bulldozer etc.). Our subject model of PTLsim processor though does not have any prefetcher component implemented. In our effort to extend the simulator closer to modern microarchitectures, we decided to design and integrate a data and an instruction prefetcher unit for L1 cache. In addition, we also extended and modified every part of the simulator that is directly or indirectly involved with the prefetcher components in order to be able to observe their impact on the rest of the processor.

## 3.1 Data and Instruction prefetcher components

### 3.1.1 Prefetching schemes

Several hardware prefetching schemes have been proposed over the past for adding prefetching capabilities to a system. Those vary on triggering patterns, level of complexity and aggressiveness, with more complex and aggressive schemes usually targeting on lower cache memory levels while more conservative schemes often target level one cache. All existing schemes propose logic for predicting future memory accesses. Based on this prediction, prefetch requests are initiated by the components. There is currently no existing scheme that can guarantee performance improvement. Along with the prediction logic, prefetchers consist of several other parts (such as dedicated arrays and buffers). We will make a short reference on the schemes we used in our implementation.

#### *Sequential prefetching*

Sequential prefetching is based on the principle of spatial locality. Items that follow or are close to an accessed item are likely to be referenced in the near future. Multiple word cache blocks are themselves a form of sequential prefetching. Prefetchers that implement a sequential prefetching scheme initiate prefetches on the N sequential memory blocks upon a memory access event (where N is configurable depending on the algorithm and is defined as degree of prefetching). They can vary on their trigger source and prefetching degree. It is known from the cache memory theory that the degree to which large blocks can be effective is limited by cache pollution effects. This also applies to sequential prefetching schemes and therefore degree must be chosen carefully [ 9].

Prefetch-on-miss and tagged-prefetching are the two main algorithms that are used to trigger prefetch requests. Prefetch-on-miss is an algorithm that initiates prefetches whenever a cache access results in a miss while tagged-prefetching initiates prefetches whenever a demand-fetched block is accessed, or when a pre-fetched block is accessed for the first time. Tagged-prefetching requires a flag bit to indicate prefetched blocks. Depending on the aggressiveness of the implementation, the following N sequential blocks will be prefetched ($N \geq 1$). Large prefetching degrees can reduce miss rates in sections that show high spatial locality but also can cause large traffic and cache pollution in sections with small spatial locality.

Figure 2 demonstrates the sequential prefetching scheme in the two different trigger variants. The blocks on blue color are prefetched blocks, while the arrow indicates demand accesses.

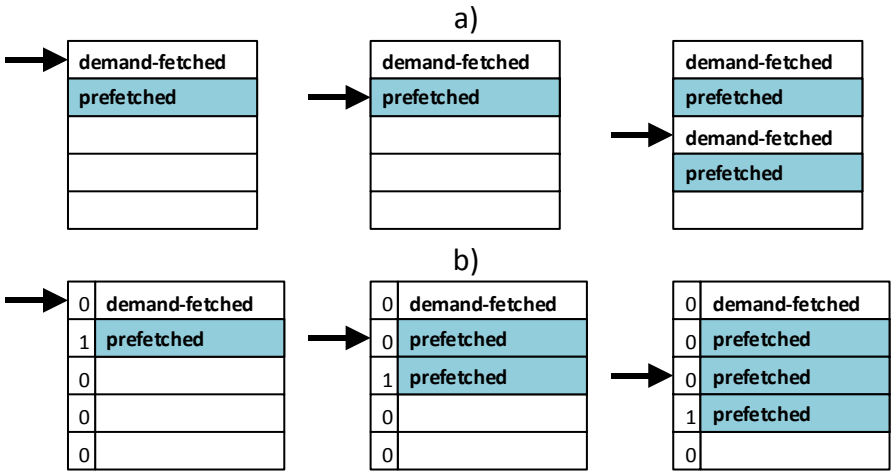


Figure 2 a) Prefetch-on-miss b) Tagged prefetching

Adaptive sequential prefetching policy [ 10] allows the degree of prefetching to vary during the program execution depending on the degree of spatial locality at a particular point of time. A common way to achieve this is by tracking prefetching efficiency during execution. Efficiency (or accuracy) is defined by the ratio of useful prefetches (prefetched blocks that were referenced by the processor) to total prefetches. Prefetching degree decreases when accuracy is low, and increases when accuracy is high.

### *Stride prefetching*

Stride prefetching is a more aggressive and complex scheme of prefetching. It employs logic to monitor processor's memory address references in order to detect patterns with constant stride and initiate prefetches on steps that are predicted to follow. This scheme aims to detect and prefetch very common array references originating from looping structures of the executed programs. The prefetching component monitors successive addresses generated for load and store instructions and tries to detect constant strides. When a constant stride is detected, the scheme indicates a successful prediction and triggers new prefetch requests on this constant stride. This procedure is called prefetcher training. Prefetching then continues as long as the step on the memory address reference for that specific load or store instruction is the same as the previous (i.e. remains constant).

Stride prefetching requires keeping track of the addresses generated for load/store instructions, for comparison and calculation of the strides. Furthermore, the unit should be able to separate which address references correspond to which instructions, otherwise it would be unable to detect strides on loop structures with more than one memory references. A component with stride prefetching logic maintains a dedicated cache called Prefetcher Table (PT) for this purpose. It stores all the information required to identify an instruction, track its address references and successfully calculate and initiate prefetches. Since the size capabilities of this structure are often limited, PT holds information only for the most recently recorded memory accesses.

As in the case of sequential prefetching, the degree may vary depending on the aggressiveness of the component. Implementations that target L1 cache are usually more conservative and have a prefetching degree of one cache line while implementations with L2 cache as target are more aggressive and can reach a prefetching degree of four lines.

Aggressiveness on stride prefetching not only depends on the number of steps that will be prefetched, but also on other factors that are relative to the prediction logic. Some algorithms for instance do not initiate prefetches until they meet a constant stride for a required number of times while others tend not to replace recorded strides with strong history when new strides are detected. To add such features on their logic, stride prefetchers also keep track of a metric called Confidence, which represents the "strength" of the recorded stride. By "strength" of a stride, we refer on a value that indicates how many times the recorded stride was detected in the previous memory references.

Unlike sequential prefetching, stride prefetcher variants may initiate prefetches that start more than one step ahead to increase the aggressiveness. However, those are currently only implemented for research purposes and most common industrial implementations adopt the regular scheme, where prefetches start from the next step after they detect a constant stride.

Policies that dynamically adjust degree, stride thresholds, number of steps ahead and other similar attributes are still under research and are not used in commercial implementations.

### 3.1.2 Data Prefetcher

The data prefetcher we designed implements a stride prefetching scheme. It is trained by the stream of physical addresses coming from the address generation units (including those that are not committed) and prefetch requests are buffer to a prefetch queue before propagated to the single port[5] data cache. The data prefetcher integrates a set-associative Prefetch Table (PT) that stores the training data. Each entry is comprised of the following fields:

1. Valid bit – Indicates whether the entry is valid or not. Used to avoid accessing trash data.
2. Past physical address used - Holds the previous address reference.
3. Instruction pointer - (RIP) of the corresponding x86 instruction, used as a tag.
4. Stride - Most recently recorded stride, for comparison with the new reference.
5. Confidence - m-bit saturating counter that indicates the strength of the recorded stride.
6. LRU state bits – For the associative array implementation.

Every physical address generated by the address generation units leads to a prefetcher notification (we define as "prefetcher notification" the event where input in supplied to the prefetcher for training). Each notification allocates an entry on the tail of the prefetcher's input buffer. If the buffer is full, the newest entry is dropped. On each cycle, the oldest entry inside the input buffer is propagated to the prefetch table. A set in the Prefetch Table is selected using the RIP's (x64 Instruction Pointer) least significant bits (depending on the PT size) as set index. After a set is selected, a tag search occurs using the RIP provided as input and the RIP field on each entry of the set. In prefetch table hit, the matching entry is selected. In contrast, on a miss, an entry is allocated in the prefetch table based on the LRU policy if all ways are occupied.

- Prefetch Table hit: A new stride is calculated using the address provided as input and the past physical address stored on the table. Both addresses are in 16 byte resolution

[5] Level one data cache is emulated as single port cache memory after custom modification, refer 3.2

(dropped down to the closest 16 byte boundary) and the new stride is also calculated in 16 byte resolution. Then, if the new stride is within a valid range but not the same as the recorded stride, the confidence field is examined, whether it is below or above a pre-defined threshold. Stride replacement and resetting of the confidence value occurs for confidence values below the threshold. On the contrary, the confidence value is decreased and the stride field is not updated. In either case, no prefetch requests are issued. For a stride match (new stride is the same as the recorded stride), new prefetch requests are issued and the confidence value is increased.

- Prefetch Table miss: The selected entry stride and confidence fields are both set to zero and no prefetch request is issued.

In all cases, the valid bit is set to 1 (indicating the entry is now valid) and the LRU field is updated (to indicate the most recent used entry). Also, the address and RIP fields are updated to the provided address and RIP (which in the case of a PT hit has no effect).

It should be noted that any prefetch requests issued by a notification that cross page bounds are dropped, since we are using physical addresses and we cannot know if the next physical frame corresponds to the next virtual page. Furthermore, null and off-range strides do not contribute to the training of the prefetcher and therefore don't update the prefetch table[6]. Finally, new requests that are on the same cache line as the input address are dropped. All remaining prefetch requests are placed in the data prefetch queue.

The following figure presents a block diagram of the data prefetcher described above.

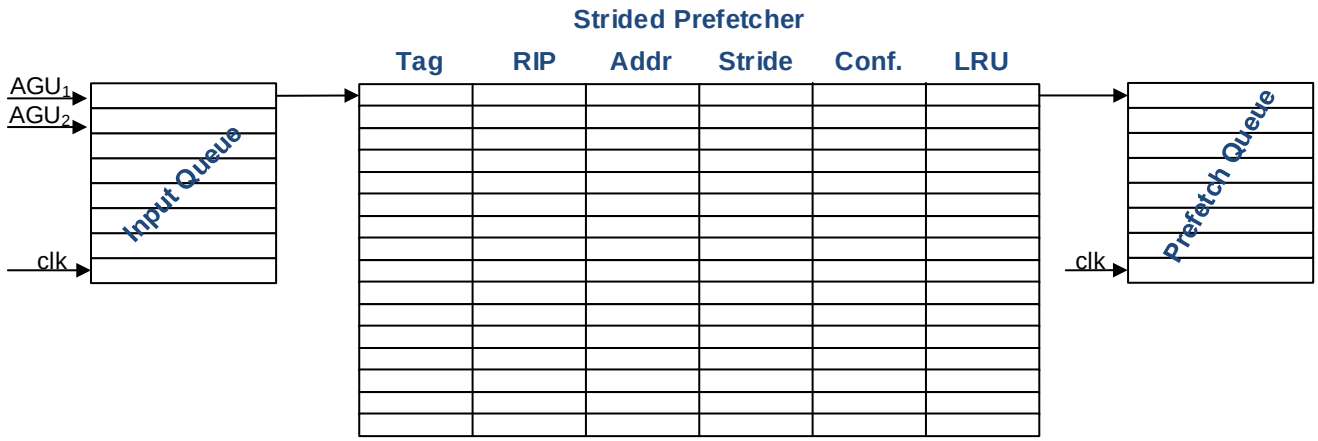


Figure 3 Data prefetcher layout

### 3.1.3 Instruction Prefetcher

The instruction prefetcher implements a prefetch-on-miss sequential prefetching scheme and is trained on every instruction cache miss. Similarly with the data prefetcher, supplied input is initially placed in the input queue. On each cycle, the component creates a prefetch request for the next sequential instruction cache line and deploys it in the instruction prefetch queue. Then, the request is forwarded to the L1 instruction cache and in case of miss, a Miss Address Buffer[7] (MAB) entry is

[6] Address, RIP, LRU and valid fields are updated by the notification, stride and confidence fields remain intact

allocated and the request is propagated to the L2 cache. Finally, it should be noted that requests that already exist on the Miss Address Buffer[7] are dropped. The following figure shows the block diagram of the implemented instruction prefetch unit.

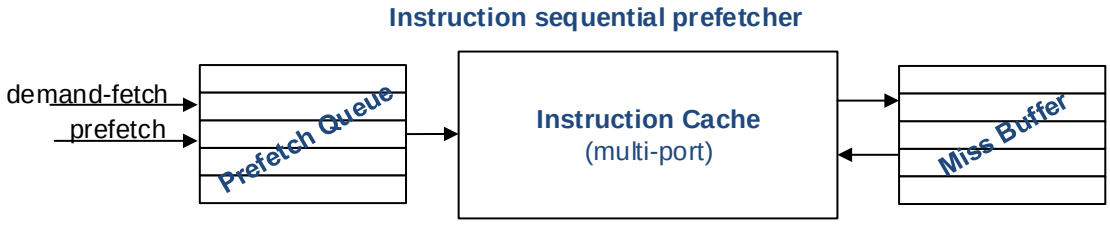


Figure 4 Instruction prefetcher layout

### 3.1.4 Prefetch requests and design summary

Both data and instruction prefetchers initiate prefetch-requests with similar structure, giving the option to share physical parts. However, our design states two independent prefetch queues, one dedicated for each prefetcher. Prefetch requests from both prefetch queues (instruction and data) are treated the same way. Upon each clock cycle, L1 cache memory is probed in order to see if the corresponding line (=memory block) is already valid. In such case, the request is dropped. On the contrary, the Miss Address buffer (MAB)[7] is also probed to check whether this specific block is already being requested. A MAB entry is allocated for the request, in order to fetch the line from lower memory levels. If there is no MAB entry available or another reference to the same memory block already exists in the MAB, the request is dropped. After MAB allocation, the memory subsystem is responsible to serve the request, similarly to the demand requests, with the difference that it will not cause a wake-up event (since the requested memory block was not referred by a load instruction).

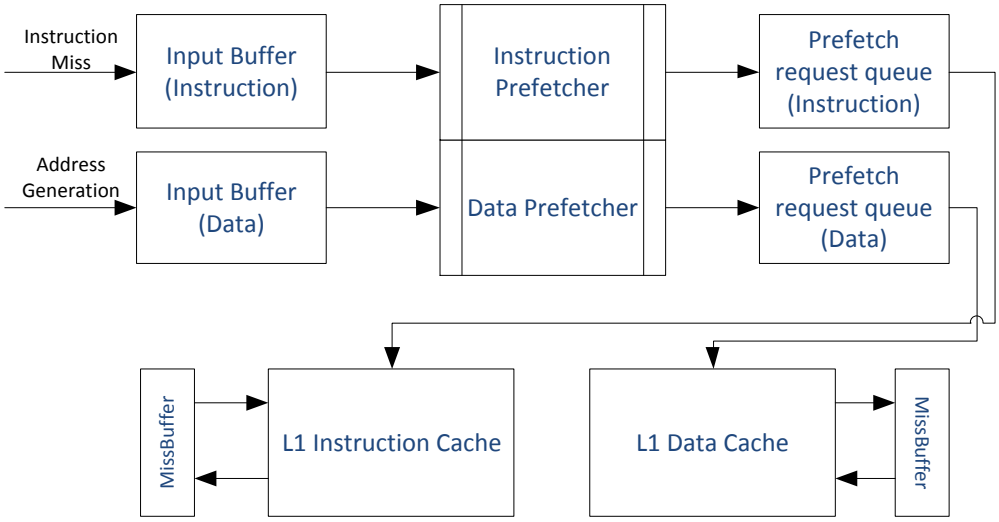


Figure 5 Prefetching subsystem outline

[7] Brief reference at section 4.1

In our implementation, data and instruction prefetchers share functional parts but are internally separated and can be considered as two independent components. Overall, there are two input buffers with four entries each, two prefetch queues with eight entries each and one 4-way set-associative Prefetch Table with 64 entries in total (16 lines, 4 ways). The structure of each entry is:

Input buffer entry:

- Memory address reference
- Instruction pointer
- Thread id (for use in multi-core systems)
- Valid bit

Prefetch queue entry:

- Prefetch request address
- Thread id (for use in multi-core systems)
- Valid bit

Prefetch Table entry:

- Last memory address
- Instruction pointer (used as tag)
- Recorded stride
- Confidence
- LRU
- Valid bit

The component is fully customizable in terms of queue sizes, buffer sizes, prefetch degree, PT associativity and size, thresholds, stride resolution and range. It is placed within the data cache subsystem and belongs to Cache Hierarchy of PTLsim model.

## 3.2 Single tag port emulation mechanism

One of the goals of this thesis is to successfully observe the impact of the expansion on the prefetcher sub-system on the performance of the processor. This includes the positive impact (the performance boost with the decrease of memory stalls) as well as the negative impact. As already mentioned, prefetching can increase memory traffic. This can lead to a general increase in resource competition within the cache memory subsystem. This applies to both size and bandwidth capabilities. Prefetch and demand requests compete on the same system and in many cases reach its limits. As a result, requests on both sides may be blocked out, stalled or even dropped. Such events affect the system's performance and cannot be ignored. In order to achieve accurate simulation, limitations like these, which exist on the real world and apply on real hardware, must be taken into account. Further modification was required on the PTLsim memory sub-system model to add limitation capabilities, which are described in the next sub-section.

### 3.2.1 Contention and the arbitration unit

The number of concurrent accesses that happen on a memory bank depends on the number of tag-ports this bank has. Real hardware memories have a finite number of ports implemented which implies that we should limit the memory accesses that can be made at the same time on a simulation. In the case of PTLsim, this does not happen.

In addition to prefetcher units' integration, our expansion includes the implementation of a cache access contention mechanism and an arbitration unit that prioritizes cache requests in order to accurately emulate single tag port cache memories. This allows us to observe side effects of a prefetcher component apart from the theoretical impact.

PTLsim has separate L1 cache memories for data and instructions.

L1 Data cache can be accessed by:

- Miss Address Buffer
- Out of Order Core (load issue and store commit stages)
- Data Prefetcher

L1 Instruction cache can be accessed by:

- Miss Address Buffer
- Out of Order Core (Instruction fetch stage)
- Instruction Prefetcher

Our implementation models contention, which practically means that a memory bank can be accessed by only one source at a time. In a higher level, this equals to single tag port emulation. Furthermore, it gives the ability to manually set time duration for each access if, for instance, we assume that in our system a memory cannot be accessed until it fully serves a request.

The configuration we use implies that we have a single-port data cache and a multi-port[8] instruction cache. The single-port data cache can be accessed one time at each cycle (contention is active for one cycle and not until the request is fully served).

Our mechanism was not designed to support a configurable number of ports for a memory bank since this was out of scope. However it can be easily modified to implement a semaphore approach (instead of a mutex) and support configurable number of simultaneous accesses.

An issue that arises when there are many sources of memory access requests is which of them should be served and which of them should be blocked in a case of simultaneous arrival. This cannot be left to random since the significance of each source does vary. For instance, a prefetch request should not be able to block out a load request that arrived at the exact same clock cycle, since the first one tries to predict a future reference and possibly save a few execution cycles while the second serves an actual program instruction and is quite possibly going to stall the processor. Therefore, static priorities have been set to each cache access source of the microprocessor and an arbiter unit decides which request will be served at each cycle. It is important to notice that any requests with lower priority that refer to the exact same cache line with the request that is being served, will also be served.

The priority for each L1 cache memory is as follows:

L1 Data Cache:

- Miss Address Buffer (serving demand requests)[9]
- Load issue stage
- Store commit stage

[8] Multi-port equals to zero-cycle lock duration for the instruction L1 cache

[9] After custom modification, ref 4.1, MissBuffer separates prefetch and demand requests

- Miss Address Buffer (serving prefetch requests)[9]
- Data Prefetcher

L1 Instruction Cache[10]:

- Miss Address Buffer (serving demand requests)[9]
- Instruction Fetch stage
- Miss Address Buffer (serving prefetch requests)[9]
- Instruction Prefetcher

Miss Address Buffer must have the highest priority since it accesses L1 cache to validate/enable memory lines. Blocking any incoming data can lead to starvation and cause a deadlock. Second priority has been given to the pipeline and last priority is given to the prefetcher components which do not have any critical role in the operation of the microprocessor or the execution of the program. Assigning different priority orders to the access requests can lead to less aggressive or more aggressive prefetcher design. We selected the previous priority order in an effort to resemble a moderate aggressiveness system.

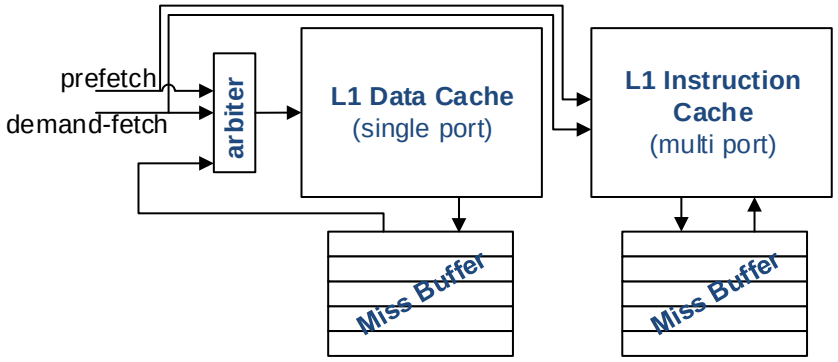


Figure 6 Arbiter block diagram

## 3.3 Validation

The verification campaign focuses mainly on the data prefetcher due to its design complexity. On the contrary, instruction prefetcher is implemented through a simple memory buffer. For validation purposes we have developed a series of custom programs, each aiming on different features of the data prefetcher component. In this section we will only mention the most representative which can provide a comprehensive picture of the validation stage. Due to the nature of PTLsim, which requires a complete ELF executable with dynamic links to standard libraries in order to stab itself into it, our programs were written in IA64 assembly in-lined to C source code and compiled using GNU C compiler instead of native IA64 assembly. Therefore some extra overhead is caused by the implementation of C runtime and initialization, which explains some small deviation from the expected results.

We will briefly mention the data prefetcher setup as we go through explanation of the validation programs. For a complete configuration report please refer to section 5.

[10] In current configuration I$ cache is considered multi-port, therefore, this priority has no actual impact

| Description | Setting |
|---|---|
| Prefetch Table | 4-way set associative |
| Prefetch Queue | 8 entries |
| Input Buffer | 8 entries |
| Confidence size | 3 bits (max confidence = 7) |
| Confidence threshold | 3 |
| Prefetch degree | 1 (single step) |
| Stride resolution | 16 byte |
| Stride range | 5 bits, +- 16 steps (equals to 4 cache lines) |
| Cache lock penalty | 1 cycle |

Table 1 Data prefetcher configuration

Our goal is to validate the correctness of our implementation. Specifically these custom programs aim to test:

- Prefetching on constant strides
- Off-rage strides
- Null strides
- Cross-pages
- Variable strides

The results reported in this section refer to versions Baseline Model and Experimental Model described in section 5.

### *Program 1 – Prefetching on constant stride*

This program aims to test and demonstrate the function of the data prefetcher providing a perfect scenario for it: a loop that requests data from memory on a fixed stride. In total, 50.000 memory loads take place on a standard 64 byte stride (equals to 1 cache line). In addition, it creates an artisan delay at each loop, enough for the prefetched data to arrive to L1 cache, in order to count these prefetches as Cache hits for the processor core by the time they are referenced.

| Simulator ver. | Cycles | D$ accesses | D$ hit | D$ miss |
|---|---|---|---|---|
| **Baseline** | 22400524 | 80139 | 22448 | 57691 |
| **Experimental** | 13965300 | 80176 | 72200 | 7976 |

Table 2 Validation program 1 simulation results

The program reports 37% performance improvement in execution time and a significant hit rate improvement. The number of accesses that lead to a hit is ~50.000 more, compared to Baseline version, which is the number of loads we performed on a fixed stride, as expected. The remaining accesses are part of the overhead caused by C runtime and are equal for both versions.

| Event | Count |
|---|---|
| Data notifications | 104398 |
| Data trains | 100183 (4215 full input queue drops) |
| PT misses | 21678 |

| | |
|---|---|
| PT hits | 78505 |
| Stride match and request | 49666 |
| Stride replaces | 4856 |
| Cross-page drops | 795 |
| Confidence decrease | 273 |
| Null strides | 21125 |
| Off-range | 992 |
| Same cache line drops | 798 |
| Requests that were already in queue | 93 |
| Prefetch requests delivered to cache hierarchy | 49573 |

Table 3 Validation program 1 - prefetcher statistics

Again we can see the number of stride matches and requests is close to the number of program's loads. 50.000 loads on 64 byte stride cover an area of 3.200.000 bytes. Assuming the page size is 4096 bytes, this area can be stored in 782 virtual pages, which also matches the number of drops caused by Cross-pages. An interesting observation is that the number of notifications for the prefetcher differs from the number of data accesses of the program. This happens because notifications are triggered by the address generation units and not by cache accesses and this also applies to addresses generated for instructions that are not committed. The results indicate correct operation of the data prefetcher.

### *Program 2 – Prefetching and off-range strides*

Design states stride-width limitations. More specifically, based on current configuration, the prefetcher cannot monitor accesses with a distance greater than 256 bytes (5 bits and 16 byte stride resolution define a range of -256 to +240 bytes). This program aims to test if this limitation is correctly implemented. We repeat program 1 but with 256 byte step this time, which is out of range distance for the prefetcher.

| Simulator ver. | Cycles | D$ accesses | D$ hit | D$ miss |
|---|---|---|---|---|
| **Baseline** | 22400496 | 80115 | 22468 | 57647 |
| **Experimental** | 22400678 | 80008 | 22624 | 57384 |

Table 4 Validation program 2 simulation results

Simulation for Baseline and Experimental model is almost identical and prefetcher does not seem to affect execution time on this program. Event statistics report that off-range drops are 50980 as we expected. To complete our test we modified the program to have a stride of 240 bytes, which is the upper limit of the range. Simulation stopped after 14.343.356 cycles, which is similar to the results of program 1. The small deviation is due to the increase of the area covered by loads and causes increase of cross-page drops.

An important notice is that both versions have almost the same execution time, but even with the data prefetcher almost idle there is still an instruction prefetcher that is operating normally and has a positive impact on execution. The question that arises is how it is possible to have the same execution time. This issue is discussed in section 5 Experimental results.

The results indicate again correct operation of the data prefetcher.

### *Program 3 – Null strides*

This program tries to test and demonstrate a special behavior of the prefetcher. As stated on the description, null strides do not contribute on training and are ignored. At the same time, address input is dropped down to stride resolution, reaching the closest 16 byte boundary downwards. The combination of these two basically means that strides that are less than 16 bytes can actually be observed when crossing 16 byte boundaries. This gives the ability for the prefetcher to continue contributing in smaller memory access sequences like reading strings. However, this can be beneficial on a limited degree. The fact that resolution is smaller than cache line means that detection of stride = 1 will probably lead to a request within the same cache line, which will be dropped. Unless the resolution is same or greater than cache line, this will lead to successful requests only on the detection of the last 16 byte-boundary cross within a cache line. Still, this can lead to a 100% hit rate on a string accessing.

| Simulator ver. | Cycles | D$ accesses | D$ hit | D$ miss |
|---|---|---|---|---|
| **Baseline** | 14967347 | 80136 | 66197 | 13939 |
| **Experimental** | 13844940 | 80170 | 72881 | 7289 |

Table 5 Validation program 3 simulation results

Due to small stride, the program itself has already a high data cache hit rate. Results verify that even with a stride smaller than prefetcher's resolution, the component is still able to improve execution time. The final results for the experimental version are almost identical to program 1, where prefetching was 100% accurate.

Data prefetcher statistics

| Event | Count |
|---|---|
| Stride match and request | 6598 |
| Stride replaces | 4856 |
| Cross-page drops | 112 |
| Confidence decrease | 273 |
| Null strides | 46121 |
| Off-range | 989 |
| Same cache line drops | 19548 |

Table 6 Validation program 3 prefetcher statistics

As expected, many null stride events and same line drops were recorded. This is natural since computable strides were only the ones that crossed a 16 byte boundary. Furthermore, most of them lead to a request on the same cache line. Finally, only requests that crossed a cache line boundary lead to a successful prefetch request.

### *Program 4 – Variable stride*

This test program tries to observe how the prefetcher behaves when the same instruction changes its memory access step at each loop. According to design description, this should lead to PT hits and stride mismatches. Since each stride only lasts for one loop, none of them (recorded strides) does go over the threshold and every time it is being replaced. In short, this program expects the prefetcher not to issue any prefetch requests for the loop and record approximately 50.000 stride replacements within the PT.

The way this is achieved is by changing the stride each time to 0, 64, 128 and 196 bytes sequentially. Since we actually do not expect any prefetches we do not create any delay between memory accesses (and this explains the short runtimes).

| Simulator ver. | Cycles | D$ accesses | D$ hit | D$ miss |
|---|---|---|---|---|
| **Baseline** | 937551 | 80102 | 22424 | 57678 |
| **Experimental** | 898352 | 80020 | 22846 | 57174 |

Table 7 Validation program 4 simulation results

Results indicate that the hit and miss rates for L1 data cache have not changed. Small runtime improvement for the experimental version is caused by the active instruction prefetcher.

Data prefetcher statistics:

| Event | Count |
|---|---|
| PT misses | 21610 |
| PT hits | 79273 |
| Stride match and request | 447 |
| Stride replaces | 42341 |
| Cross-page drops | 14 |
| Confidence decrease | 271 |
| Null strides | 34431 |
| Off-range | 974 |
| Same cache line drops | 795 |

Table 8 Validation program 4 prefetcher statistics

It seems that the 50.000 accesses initially lead to a PT hit and then were divided to stride replacements and null stride events. This is in line with the expected behavior and indicates the correct function of the data prefetcher unit.

# 4 Modifications – Design changes and fixes

PTLsim focuses primarily on implementing a fully working Out-of-Order Core that resembles a modern microprocessor for the IA64 architecture. The core adopts features from many modern microprocessors and accurately reproduces them in a simulation environment with very few assumptions derogating from an actual hardware design. However, these do not affect the function or performance of the core and only serve simulation purposes.

In addition to the Out-of-Order Core design, PTLsim models a full system, including supporting units of the microprocessor, all microcode, complete cache hierarchy subsystem and supporting hardware devices. Detailed study of the simulator, led to identification of some inaccuracies in the implementation of some supporting elements.

Part of this master thesis was to correct such misstatements and the adjustment of certain components in a more realistic model.

## 4.1 Miss-address buffer

An Out-of-Order processor is capable of executing instructions in different order than the program. This technique gives the ability to avoid stalls upon data cache misses. Non-blocking caches is a common optimization used in out-of-order processors which allows the data cache to continue serving memory requests during a miss [ 11]. This can effectively reduce the miss penalty by being helpful during a miss instead of stalling the subsequent requests of the processor.

Non-blocking caches can offer a more complex option of being able to overlap multiple misses. To do so, a non-blocking cache should have a dedicated Miss Address Buffer (MAB) for holding outstanding cache misses. For L1 caches, this is usually accompanied by a structure that maintains information about instructions that have missed in order to be able to return a filled load to the processor core. This structure is called Load Fill Request Queue.

### 4.1.1 Coupling dedicated MABs to cache memories

PTLsim adopts the described memory scheme and implements a non-blocking cache hierarchy capable of handling multiple misses. However, the design has an inaccuracy: There is only one Miss Address Buffer structure which is shared in all three levels of cache hierarchy. This scheme cannot be implemented on real hardware, but the problem is beyond that, as it also affects the execution performance. Having one large MAB for all cache memories instead of four dedicated to each one (L1 instruction, L1 data, L2, L3) leads to loss of detail in the simulation since this buffer is almost never filled completely, even with the increased memory traffic caused by the prefetcher components.

This design is likely to skip internal exceptions of the processor. It also provides flexibility on the memory system to share resources between independent parts when needed. With the existing design, we were unable to successfully capture the impact of the prefetcher in the memory subsystem's traffic, which could be critical in excessive or abusive prefetcher activity. Our modification includes the coupling of a separate dedicated MAB to each cache memory level in order to fix this inaccuracy. Figure 5 shows the updated memory design where a dedicated Miss Buffer is attached to each cache memory.

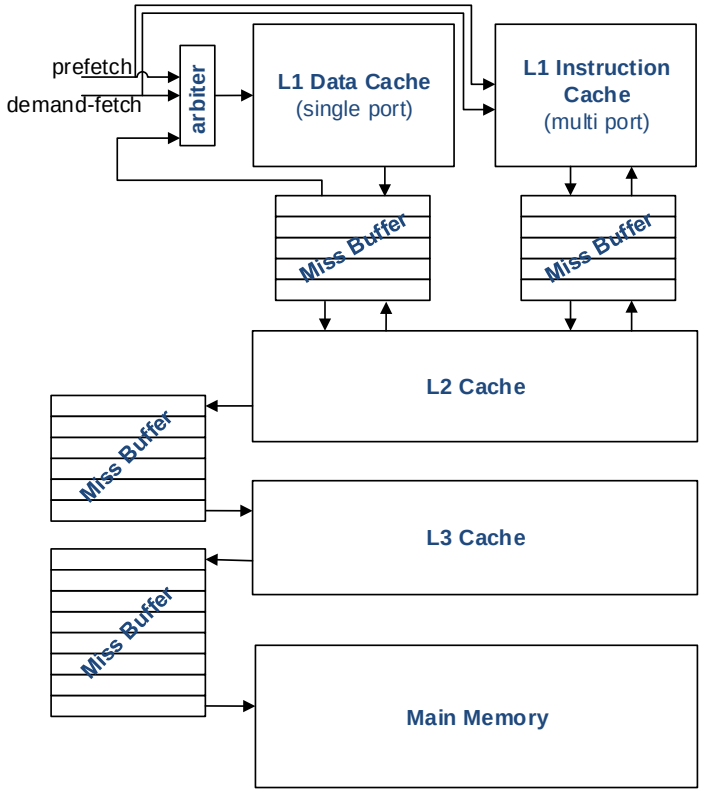


Figure 7 Memory system block diagram

Regarding the updated memory system design, with the addition of dedicated miss buffers in each cache memory, the data flow is the following: The address generation unit generates a load or store request. L1 data cache is looked up. In case of a miss, a slot in the L1/L2 (data) miss buffer is allocated and the request propagates to the L2 cache. This process is repeating until the request is served (=data found). The same data flow is followed on a miss in the L1 instruction cache. Finally, on a full miss buffer, the simulated microprocessor model does the following actions depending on the microprocessor structure that issue the memory request:

- Demand-fetch request: An exception is raised and the memory request is re-issued on the next cycle. This applies to L2 and L3 Full-MAB events also[11]. It is an issue out of this project's scope whether such events (L2 and L3 full MAB) should raise an exception or stall the request until a free MAB is available.
- Prefetch request: The prefetch request is dropped (never re-issued).

The MAB sizes used in our configuration are the following. L2 and L3 MABs equal to the summary of L1 MABs in order to avoid full-MAB events in lower memory levels:

- L1 Instruction MAB : 8 Entries
- L1 Data MAB : 32 Entries
- L2 MAB : 40 Entries

[11] Our configuration ensures Full MAB events can happen only at L1.

- L3 MAB : 40 Entries

This setup defines that our subsystem can handle 32 data misses and 8 instruction misses at the same time while the previous design could allow up to 64 outstanding misses.

Due to MAB coupling on each cache, a small functional difference occurred in relation to the previous design. Entries inside the MAB were flagged as Data and/or Instruction requests and when the requested line became available, it was enabled to the corresponding L1 cache depending on the indication of these flags. This practically meant that an entry inside the MAB could be both Data and Instruction request and should be enabled to both L1 data and L1 instruction caches. This is an extremely rare occasion since most executable formats support different sections for user code and user data, but still it is fully legal for the architecture and the processor should be able to support it. Our new design implements a one-by-one MAB scheme, meaning that a MAB entry corresponds to exactly one entry on the higher level. This means that a L2 / L3 MAB entry cannot be both data and instruction at the same time (since data and instruction L1 caches have different Miss Buffers). In order to handle this rare occasion, our design raises a Full MAB Exception when a data request is issued and an instruction request on the exact same address is already valid inside a MAB and vice versa. This causes a total of 6 cycle stall (L2 to L1 latency) until the second request is served.

### 4.1.2 Prioritization inside the buffer

Our data prefetcher design was intended to operate in the background and avoid preventing the operation of the processor core. It should always come in last priority when competing with the core on resource access. The memory arbiter logic (section 3.2.1) was based on this idea making sure that the component could not access resources when the core required them. However, there was still a small chance for the prefetcher units to be able to hinder the core through the Miss Buffer. A prefetch request arriving from a lower memory level is enabled to a cache memory through the Miss Buffer which always has the highest priority (blocking incoming data is dangerous and can cause a deadlock to the core). Furthermore, the Miss Buffer itself is a resource where prefetcher and core both have access. Even though priority for accessing the Miss Buffer is given through the cache memory (only the request that accesses a cache and leads to a miss allocates a MAB entry), resource competition still lies in the use of space within the limited size of the buffer.

Experimentation showed that there are cases where the prefetcher issues an excessive amount of requests and prevents the core from accessing the cache subsystem through the Miss Buffer. This can happen even in cases where the hit rate is increasing and the unit produces useful prefetch requests. This behavior in combination with the need of prioritizing the core's access inside the Miss Buffer led to the decision of separating demand and prefetch requests. A new bit-flag was added to the MAB entry structure for indicating whether it is a prefetch or a demand request.

The two kinds of MAB entries now accept different treatment. As already stated in section 3.2.1 the arbiter logic gives higher priority to the core than to the Miss Buffer with a prefetch request. In addition, we added a limitation on the number of MAB slots allocated for prefetch requests in order to ensure that the MAB will not be filled by excessive amount of prefetches.

The new design allows a prefetch MAB entry to turn into a demand MAB entry, but not the opposite. In a case where the core tries to allocate a new MAB (demand) entry for a cache miss and the referenced memory line already exists in the Miss Buffer as a prefetch request, the corresponding

entry clears the prefetch flag and is treated from now on as a demand request with increased priority. This modification also simplifies the measurement of effectiveness for the prefetcher unit.

## 4.2 Branch prediction unit

Pipelining is a common implementation technique that takes advantage of the parallelism that exists among the actions needed to execute an instruction (known as Instruction Level Parallelism – ILP) for improving execution time. Each action (often called pipe stage) is connected to the next forming a pipe where instructions enter at one end, progress through all the stages and exit at the other end. The key ingredient of this technique is that each stage operates simultaneously with the other stages, completing one execution action of many different instructions at each cycle. Every modern processor uses pipelining to overlap the execution of instructions. [ 11]

Pipelining can significantly improve performance, especially when it is fed with a continuous stream of instructions. A well-known type of hazards that are likely to break this instruction stream is Control hazards. Those arise from pipelining instructions that change the program counter. When a branch is executed it may (taken) or may not (not taken) change the program counter. This means that fetching of instructions could be redirected to another location or not. This information is usually not available until execution stage of a branch. Most pipelines try to continue fetching at one of the two possible directions. The choice is usually based on a prediction of the branch result. This prediction might be static, in terms where it is always predicted as taken / not-taken, or dynamic where it arises from a sophisticated logic. When execution stage completes and the result of a branch was correctly predicted and the correct direction for fetching instructions was chosen, execution continues without problems. However if the prediction was wrong (misprediction) and instructions were incorrectly fetched, a pipeline flush must occur and all mistakenly fetched instructions must be discarded along with any changes they caused. Control hazards can cause major performance loss, especially in very deep (=with many stages) pipelines, such as the ones implemented in high-performance x86 processors. The later the branch resolved, the greater the performance loss upon misprediction.

Many techniques have been proposed in order to dynamically predict the behavior of branches by hardware in execution time, in order to reduce branch costs. Some of them (such as two-level adaptive predictor with history buffer), with high level of complexity, can reach correct prediction rates over 97%. A hardware component that implements branch prediction logic and tries to guess which way (taken or not-taken) a branch will go is called Branch Predictor.

Branch predictors are very common in all types of processors, varying in complexity and accuracy. Simple and low cost predictors can be very efficient on small pipelined RISC processors, where misprediction penalty is small and an accuracy of 90% is satisfactory enough, while more complex and high-performance processors usually implement more sophisticated predictors that can achieve higher accuracy.

Knowing the result of a branch however is only the first aspect for successfully redirecting the instruction fetching stream. And it is usually enough when the prediction is not-taken and the program counter is not expected to change. But, in cases where the prediction is taken and the pipeline must redirect, it is also required to know where the instruction fetching stream will be redirected to. Branch instructions usually include the target address of the branch in a relative format. The final address is fully calculated at execution stage. This means it is useless knowing the result of the branch if the redirection address is unavailable.

A Branch Target Buffer is a small dedicated cache memory that supports branch prediction mechanisms by storing the last calculated target address for branch instructions. The usefulness of this cache is based on the fact that the same branch instruction is likely to have the same target address in the future. It is accessed at fetching stage alongside with branch predictor result. If the prediction is taken and a target address is already available for that specific instruction in the BTB, fetching redirects there, otherwise, depending on the model, sequential fetching or stalling may occur.

The PTLsim simulator model supports prediction mechanisms for every type of flow control instructions. Whenever a branch instruction is fetched, the branch prediction unit indicates where (memory address) will the program flow continue, in order to redirect fetching. The decision on the redirection address is taken with the assistance of a Branch Predictor, a Branch Target Buffer and a Return Address Stack[12]. The implemented BPU of PTLsim handles flow control instructions as follows:

1. Conditional Branches: Conditional branch instructions may or may not redirect the program flow. The instructions are predicted taken or not taken through a combined predictor. The unit consists of two independent predictors, a Bimodal predictor and a Two-level adaptive predictor. The final decision is based on a meta-predictor that decides which predictor was more accurate in the near past. If the prediction is taken, the target address of the branch was returned to the core for fetch redirection, otherwise, the sequential address was returned.
2. Indirect Jumps: Indirect jumps do not require prediction (it is known that they are taken) and only the Branch Target Buffer is invoked to get the redirection address. If a previous record exists on the BTB, fetching is redirected there, otherwise, fetching continues sequentially. A pipeline flush will discard all fetched instructions upon address resolution (at execution stage) and fetching will be redirected.
3. Call prediction: Subroutine call instructions are likely to return on the source instruction memory address. This address is temporarily saved (pushed) on a Stack structure, called Return Address Stack, and is recovered (popped) when the subroutine returns. This gives us the ability to know and use the return address of a call and avoid pipeline stalls.

Even though the implemented model resembles a branch prediction unit found in modern x86 processors, it has some shortcuts on the flow that cannot exist on real hardware. More specifically, the model includes one Branch Target Buffer that is accessed only for indirect branches. In case of conditional or direct branches the simulator uses the actual target address of the branch, which is known through the simulation environment but would not be available until execution stage[13] on real hardware. This inaccuracy benefits execution time in a way it should not.

### *BPU modification*

Our BPU modification includes the integration of a second, separate Branch Target Buffer dedicated to the support of conditional and direct branches. Instead of using the actual target address, which is only available in simulation environment, the new dedicated BTB will be probed at instruction fetching stage to check whether a target for this branch is stored. In such case, fetching will redirect at the stored target address, only if the predictor indicates a taken branch prediction. In case there no

[12] Return Address Stack structure is briefly explained on bullet 3

[13] The calculation of target address can be accelerated by the use of a dedicated adder before execution stage.

target address is available through the BTB, fetching will be stalled until the target address is calculated. The old BTB that supports indirect branches will be half size compared to the new BTB supporting conditional and direct branches. Both BTBs will be accessible at Instruction Fetch stage and will be updated at Commit stage.

During validation stage, BTB updating was tested at Execution stage also. Results showed an interesting behavior of excessive BTB pollution which seemed to be caused by non-committed branch instructions. This seemed to have a significant negative impact on performance and therefore updating BTBs at execution stage was rejected without further investigation. Nevertheless this behavior constitutes a remarkable reference.

The updated block diagram of Branch Prediction Unit follows:

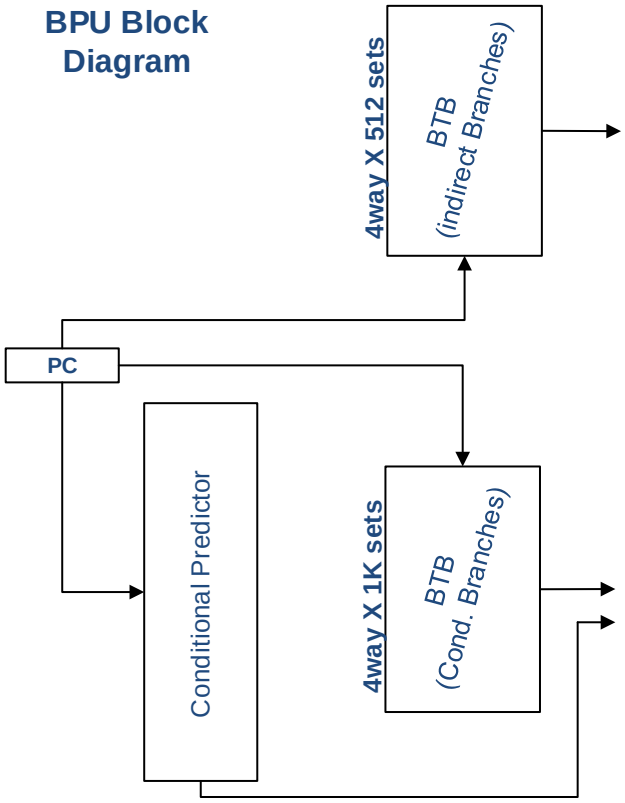


Figure 8 BPU block diagram

An issue that arose with the correction of the BPU model was the long stalls and flush penalties in cases of BTB miss or misprediction (a stored address in BTB does not necessarily mean it is a correct address), since the execution stage is deep within the pipeline. A common technique that resolves this problem is the integration of a dedicated adder for calculating branch target addresses. Since the target is relative to the program counter, using a dedicated circuit that performs addition and calculates the actual address can reduce these long stalls. PTLsim did not have an adder implemented for that purpose because it used the actual target address through the simulation environment and did not need to calculate it again. Our modification adopts a data flow for conditional and direct branches that emulates the integration of a dedicated adder that can supply target addresses within 3 cycles. The new flow defines that verification of branch target address takes place 3 cycles after fetching the instruction. In a case of correct fetching redirection, execution continues normally and the instruction

stream is not disturbed. In contrast, if there was a BTB miss or the stored BTB address was not correct, any instructions that were fetched after the branch instruction are flushed and fetching is redirected to the correct target address. This means that in the updated model, BTB misses or mispredictions will cost 3 cycles delay for the processor.

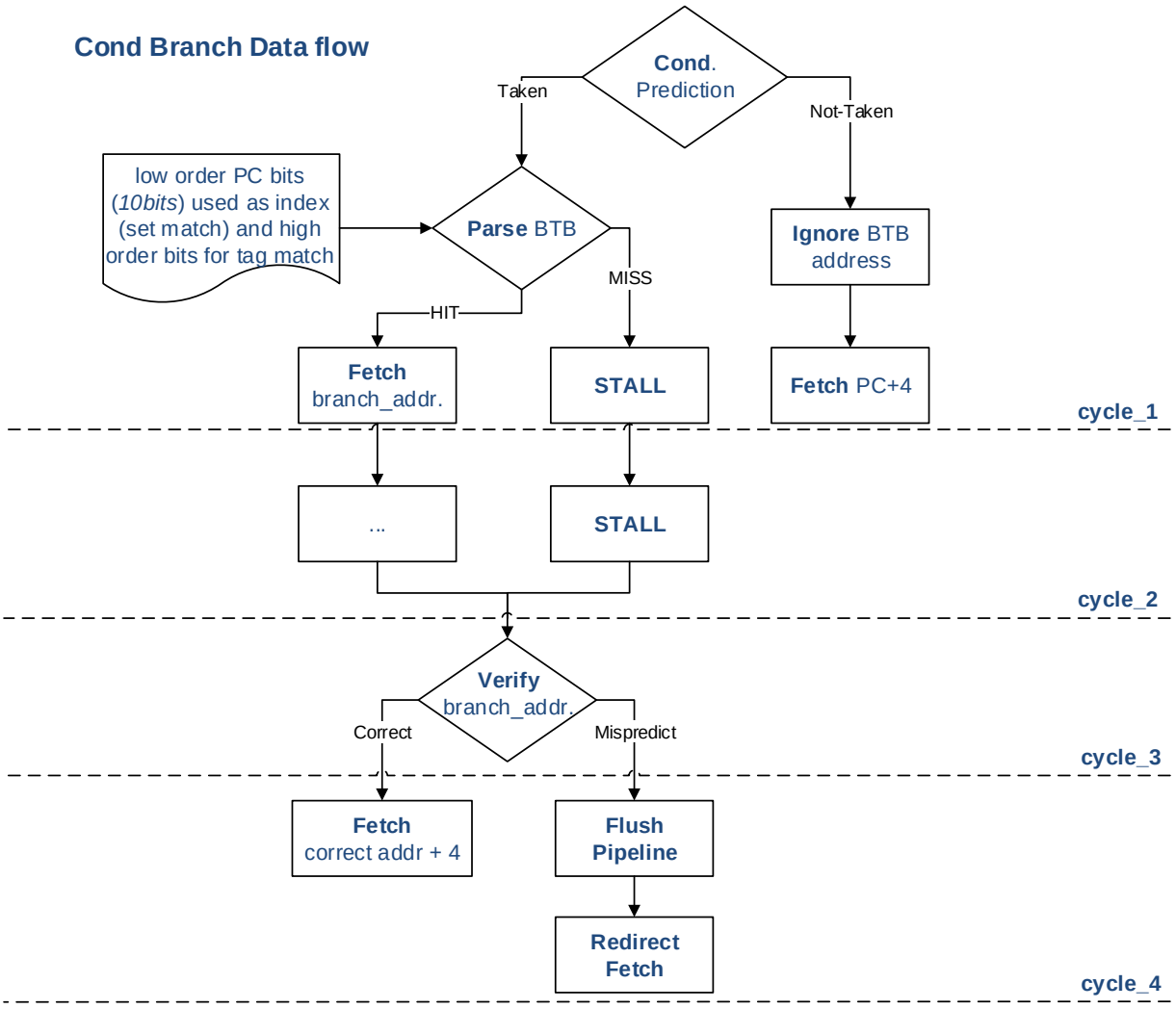


Figure 9 Conditional Branches data flow

Another small inaccuracy of the PTLsim BPU model was in the indirect branch data flow. Indirect branches are branches relative to a register value (different than program counter). This pretty much means that, unlike other types of branches, calculation of target address cannot be available before execution stage, where all data dependencies are solved by the processor core. The only chance to have a target address earlier is through BTB. So, unless there is a BTB hit for that branch instruction, it is pointless to continue fetching any instruction after that point until the target is calculated. PTLsim BPU model however continued fetching, even if the address was unavailable (to a false address) and flushed the pipeline when this branch reached execution stage. Technically there is nothing wrong with that. However some structures inside the processor were wrongly updated and some mistakenly

fetched instructions were triggering events when they should not. This incorrect behavior had almost no impact in any way for the processor. In total, there was minor performance difference which is not worth mentioning. Our modification fixes the data flow by stalling the fetching stage when there is no target address for an indirect branch.

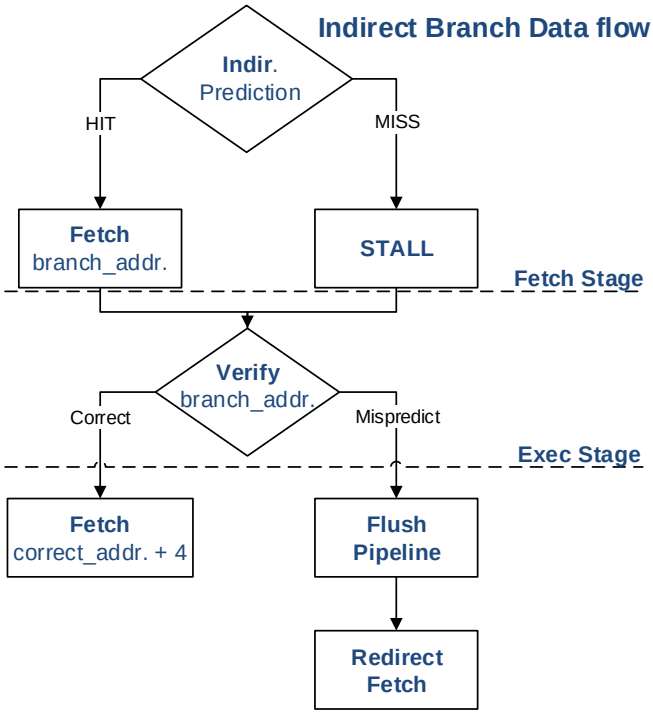


Figure 10 Indirect branches data flow

Compared to the original PTLsim model, our updated BPU design slows down the processor. This is due to the fixing of the incorrect use of unavailable target addresses for conditional branches. This approach is closer to a realistic implementation.

The BPU is closely related to the Out of Order core and more specifically to the Instruction Fetching stage. Modifications that are made to the updated BPU model require adjustment of the core in order to be able to handle the new states. These will be discussed on the following section alongside with all the other amendments of the Out of Order core.

## 4.3 Out-of-Order core modifications

Although the extension/modification of PTLsim is directed to particular entities of the processor, adjustments were required to parts inside the pipeline in order to be able to adapt to the new components and the updated designs. These adjustments are discussed in the current section.

### 4.3.1 Adaptation to the updated memory system

Three major changes took place within the memory subsystem: The addition of prefetching components, the cache access control mechanism and the updated miss address buffer design. Both

prefetchers are independent and totally invisible to the core. They only communicate with structures within the memory system and do not come in direct contact with the out-of-order core in any way. However, pipeline stages require some modification in order to be able to handle the updated Miss Address Buffer and the cache access control mechanism.

### *Miss Address Buffer and Issue stage*

Normally, all modifications in the MAB design should not affect the pipeline. However, some simplified assumptions of the original cache memory subsystem of PTLsim required changes in the Issue stage of the pipeline.

The PTLsim model describes an internal exception with the name LFRQ-MAB-FULL. This exception should be raised when the core tries to allocate a Miss Address Buffer and a Load Fill Request Queue entry and one of these structures is full. Even though this exception is referred inside the source code and is expected to exist by the memory subsystem side, it is actually not implemented or handled. Instead, there is a simplified workaround in order to handle cases of full LFRQ or MAB structures.

The implementation of Load Issue of PTLsim takes the following actions:

1. Generates the physical memory address for the reference.
2. Checks for any exceptions rose during address generation.
3. Searches the Load Store Queue for previous references and handles all cases.
4. Checks if LFRQ or MAB is full and replays in such case.
5. Checks for memory inter-locks (for SMP systems).
6. Checks if it is an internal load (data is available within a pipeline stage or register).
7. Probes the cache and in case of hit, reads the line.
8. In case of miss, allocates a LFRQ and MAB and turns idle.

Important notice: Because we are in a software implementation, these steps are taken with that specific order and each one can terminate the process. This means that if process stops at step 4, everything that follows will not be taken into account.

The workaround we mentioned earlier is at step 4. This checks the LFRQ and MAB for available entries and if there aren't any, the process is terminated and the issue is replayed. However, this is not necessary. Cases where the referenced line is valid on L1 cache do not require a LFRQ of MAB entry at all. The same applies to cases where data is already inside the pipeline and an internal forwarding is enough to complete the execution. Furthermore, misses that are already allocated in the MAB by another source also don't require a new entry. If the LFRQ or MAB were full, the stage would be replayed even if there was no actual reason for it. Especially after our modification, where more than one MAB exist, this is even more complicated. Furthermore the described process is unable to handle a case where a MAB or LFRQ entry cannot be allocated for any other reason (and our updated MAB design describes a case where this can happen).

In order to achieve a more correct approach on Load issue stage, we completely removed step 4 and expanded the logic at step 8. The load issue stage is now able to accept a LFRQ or MAB full exception upon a LFRQ and MAB allocation. In such case, the issue is replayed; otherwise, the load turns idle and waits for wakeup through the LFRQ, similarly to the original design.

This modification can successfully cover all cases where MAB or LFRQ allocation is unable for any reason, including the newly added case of simultaneous data and instruction memory references on the same line, described at section 4.1.1 Coupling dedicated MABs to cache memories .

#### *Cache access contention mechanism*

The new contention mechanism together with the arbiter logic applies additional control on cache memory accessing. Every part of the simulator that accesses cache memories must adapt to it. As already mentioned, three pipeline stages perform memory accesses along with the miss buffer and prefetchers. Issue (load issue) and Commit (store commit) stages access L1 data cache while Instruction Fetch stage accesses L1 instruction cache. The modification includes checking for cache availability and declaration of access when this happens. When cache memory is busy, the stage is replayed. Priorities are implemented through the order of access request inside a clock cycle.

### 4.3.2 Adaptation to the updated BPU

Instruction fetching stage is fully guided by the BPU when it meets control flow instructions. The original design implies that the BPU should always supply a memory address for directing fetching stage. All prediction logic is invisible to the core, which only expects an address to continue fetching instructions. It does not share any other signals to communicate with the BPU. This worked well and without any problems. However, our updated BPU model refutes the simplified target address calculation through simulation environment and describes cases where it is unable to supply any address to the core. Another change is the integration of the dedicated adder which describes that address verification will take place after 3 cycles. Adjustment of Instruction Fetching stage is required to cover such cases.

The original design was calculating the sequential fetching address and compared it with the one that the BPU returned. If those addresses were equal, it was assumed that the prediction was not taken, otherwise it was considered taken. Based on that, the fetch redirection flag was raised or not. To cover the cases where the updated BPU is unable to return an address, the Instruction fetch stage was modified to be able to handle null returns. Null returns indicate that prediction was taken but no address was available at that time. A new “address resolution” state was added to the fetch stage to cover such cases. Fetching of instructions is stalled until the address is resolved by either the dedicated adder for conditional branches (3 cycles) or execution stage for indirect branches.

When fetching stage turns to “address resolution” state, it stops fetching instructions and asks the BPU for a redirection address at each cycle. It remains at this idle state until the BPU supplies an address. When the result is ready (target address), fetching of instructions continues. Execution stage was already able to force a reset and redirection of the fetching stage for mispredictions, which also totally covers the case of indirect branches without further modification.

The way this is achieved is by adding a counter inside the BPU which indicates how many cycles remain until address resolution. This way, resolution time is configurable. Indirect branches lead to infinite stall and only execution stage can resume fetching. The address that is being returned is known since the beginning (through the simulation environment). However, it can only be recovered after the specified time that it would be normally calculated. This property gives us the opportunity to simplify address verification that is described to take place after 3 cycles. Since we already know the address, we can verify the result at the beginning and emulate the described behavior at expected time. This way, we can cause a 3-cycle stall instead of continuing fetching and flushing in cases where the stored address in BTB is not the correct target address.

This modification fixes also a rare case where the target address of a branch is equal to the sequential fetching address. In the original design this would be considered as a not-taken branch and

fetching would not be redirected or stopped. Even if this behavior is correct, still the target address is in a relative format and requires time to be calculated. Unless the target is already available through the BTB, this case should also be stalled until resolution.

# 5 Experimental results

## 5.1 Configuration and experimental models

This section presents the results of experimentation in our effort is evaluate the updated PTLsim processor model. The original PTLsim model along with our modifications is highly adjustable. In our effort to reach a more realistic approach, modifications have also been made to processor attributes. The final configuration of the microprocessor (regarding our updated PTLsim model) came after consultation with partners who specialize in the site of microprocessor design. These modifications by themselves do affect the processor performance and experimentation includes measurement of these changes individually.

The following table presents the benchmarks used to evaluate the performance of the simulated microprocessor model.

| Suite | Optimization Level | Input dataset | Benchmark |
|---|---|---|---|
| SPEC2000 | O3 | reference, test | ammp, applu, apsi, art, bzip, crafty, eon, equake, facerec, fma3d, galgel, gap, gcc, gzip, lucas, mcf, mesa, mgrid, parser, perlbmk, sixtrack, swim, twolf, vortex, vpr, wupwise |
| SPEC2006 | O2 | reference, test | perlbench, bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, h264ref, omnetpp, astar, xalancbmk, bwaves, games, milc, zeusmp, gromacs, cactusADM, leslie3d, namd, dealII, soplex, povray, calculix, GemsFDTD, lbm, wrf, tonto, sphinx3 |

Table 9 List of benchmark programs

SPEC2000 and SPEC2006 benchmark suites were simulated for 1 billion committed x86 instructions with a warm-up period of 100 million x86 committed instructions for the following microprocessor model configurations:

1. **Golden Model**: Unmodified version of PTLsim simulator.
   The original PTLsim version was used for comparison of results in total.

| Parameter description | Setting |
|---|---|
| Pipeline depth | 16 (max branch in-flight) |
| Fetch//Issue/Commit | 4/4/4 instr. per cycle |
| RAS | 1024 entries |
| Branch Target Buffer | 4-way set associative, 1024 sets |
| Combined Predictor | 16KB (65536 entries, 2 bits per entry, 16 bits history) |
| Issue Queue | 16 entries (one per cluster) |
| Reorder Buffer | 128 entries |
| Functional Units | 4 clusters (2 INT ALUs, 2 FPU ALUs) |
| Miss Buffer | 64 slots (shared) |
| L1 instruction cache | 32KB (64B cache line, 128 sets, 4-ways, 2 cycles latency) |

| | |
|---|---|
| L1 data cache | 16KB (64B cache line, 64 sets, 4-ways, 2 cycles latency) |
| L2 cache | 256KB (64B cache line, 256 sets, 16-ways, 7 cycles latency) |
| L3 cache | 4000KB (64B cache line, 2K sets, 32-ways, 15 cycles latency) |
| Main memory | Infinite size (155 cycles latency) |

Table 10 Golden configuration model

2. **BPU settings**: Golden model of PTLsim simulator with the settings of the branch predictors units updated.
   This model can is used to measure the impact of the updated BPU attributes alone.

| **Parameter description** | **Setting** |
|---|---|
| RAS | 16 slots |
| Branch Target Buffer | 4-way set associative, 512 sets |
| Combined Predictor | 8KB (32768 entries, 2 bits per entry, 15 bits history) |

Table 11 BPU settings configuration model

3. **Memory System configuration**: Golden model of PTLsim simulator with the updated memory system.
   This configuration is used to measure the impact of the updated memory system alone.

| **Parameter description** | **Setting** |
|---|---|
| L1 instruction cache | 32KB (64B cache line, 128 sets, 4-ways, 2 cycles latency, miss buffer: 8 slots) |
| L1 data cache | 16KB (64B cache line, 64 sets, 4-ways, 2 cycles latency, miss buffer: 32 slots) |
| L2 cache | 256KB (64B cache line, 256 sets, 16-ways, 12 cycles latency, miss buffer: 40 slots) |
| L3 cache | 4000KB (64B cache line, 2K sets, 32-ways, 40 cycles latency, miss buffer: 40 slots) |

Table 12 Memory system configuration model

4. **Baseline Model**: Integration of configurations (1) + (2) + (3) and BTB extension in PTLsim golden model.
   Our aim doing the modifications in the original PTLsim model is to make this baseline model as realistic as possible to evaluate the obtained speedup when the prefetchers are added (the "Experimental Model" that follows). This last model will be used eventually for the fault injections campaign.

5. **Experimental Model**: The baseline model updated with the instruction and data prefetcher.

| Parameter description | Setting |
|---|---|
| Pipeline depth | 16 (max branch in-flight) |
| Fetch//Issue/Commit | 4/4/4 instr. per cycle |
| RAS | 16 entries |
| Branch Target Buffer | 4-way set associative, 512 sets |
| Combined Predictor | 8KB (32768 entries, 2 bits per entry, 15 bits history) |
| Issue Queue | 16 entries (one per cluster) |
| Reorder Buffer | 128 entries |
| Functional Units | 4 clusters (2 INT ALUs, 2 FPU ALUs) |
| L1 instruction cache | 32KB (64B cache line, 128 sets, 4-ways, 2 cycles latency, miss buffer: 8 slots) |
| L1 data cache | 16KB (64B cache line, 64 sets, 4-ways, 2 cycles latency, miss buffer: 32 slots) |
| L2 cache | 256KB (64B cache line, 256 sets, 16-ways, 12 cycles latency, miss buffer: 40 slots) |
| L3 cache | 4000KB (64B cache line, 2K sets, 32-ways, 40 cycles latency, miss buffer: 40 slots) |
| Main memory | Infinite size (200 cycles latency) |
| Input buffer | 8 entries |
| Prefetch Table | 4-way set associative, 64 sets |
| Confidence size | 3 bits |
| Confidence Threshold | 4 |
| Stride size | 5 bits |
| Prefetch degree | 1 (single step) |
| Prefetch Queue | 8 slots |
| Cache lock penalty | 1 cycles |

Table 13 Experimental configuration model

## 5.2 Simulation results

### *IPC Graphs*

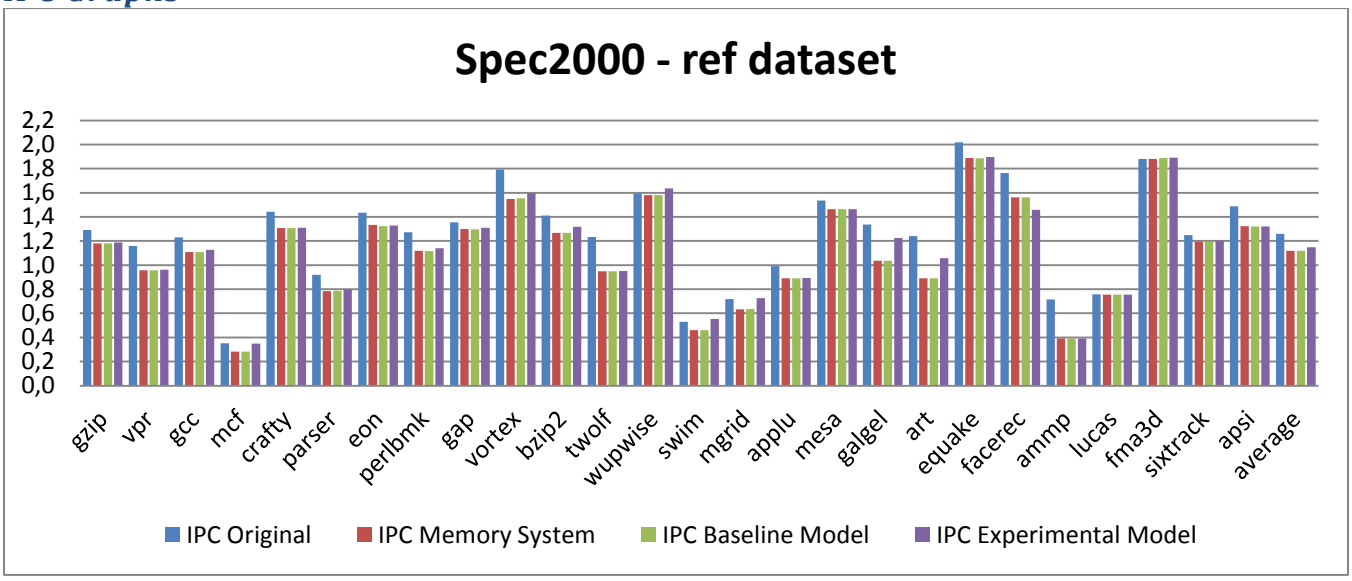


Figure 11: IPC graph for Spec2000 benchmark suite using ref dataset

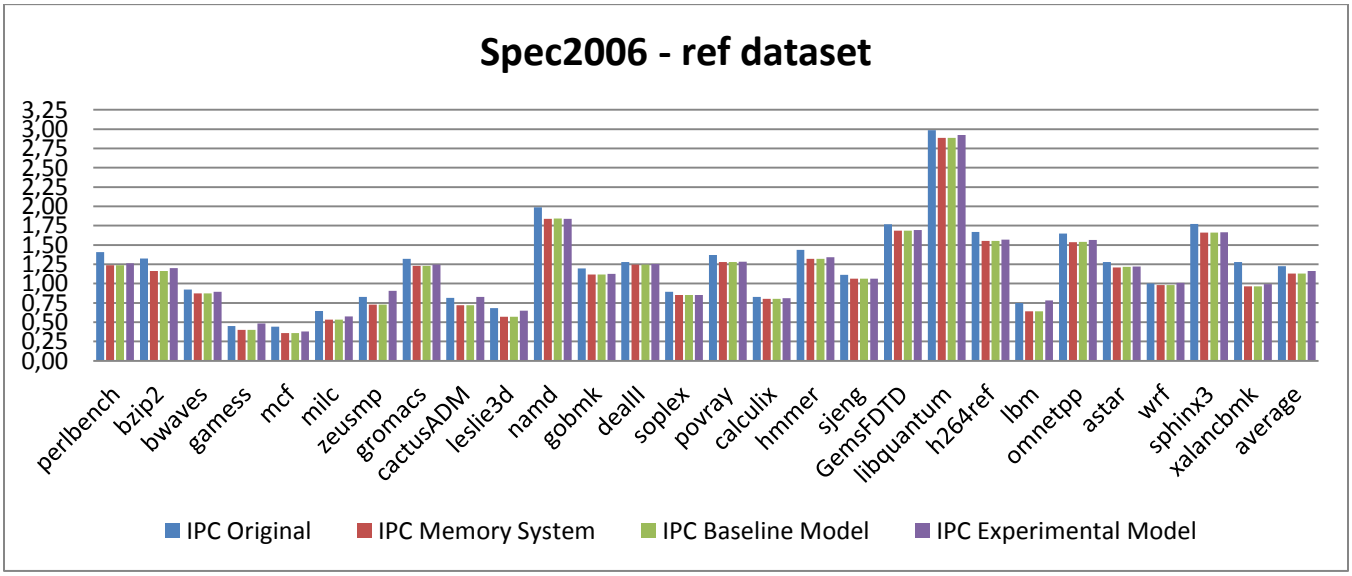


Figure 12: IPC graph for Spec2006 benchmark suite using ref dataset

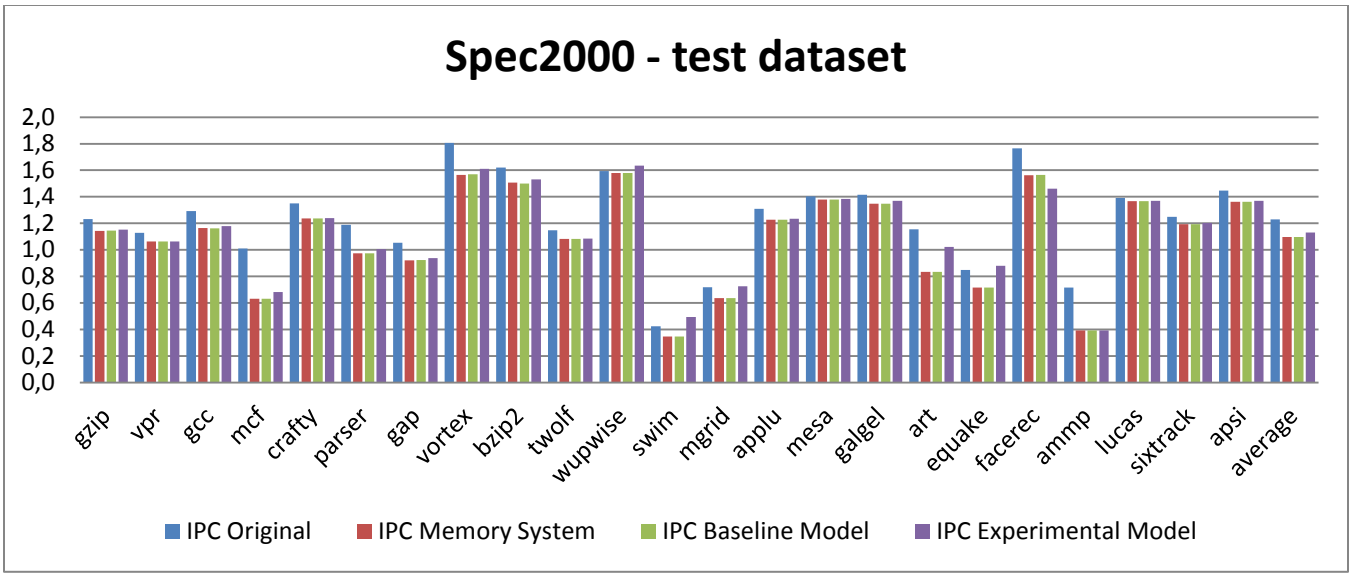


Figure 13: IPC graph for Spec2000 benchmark suite using test dataset

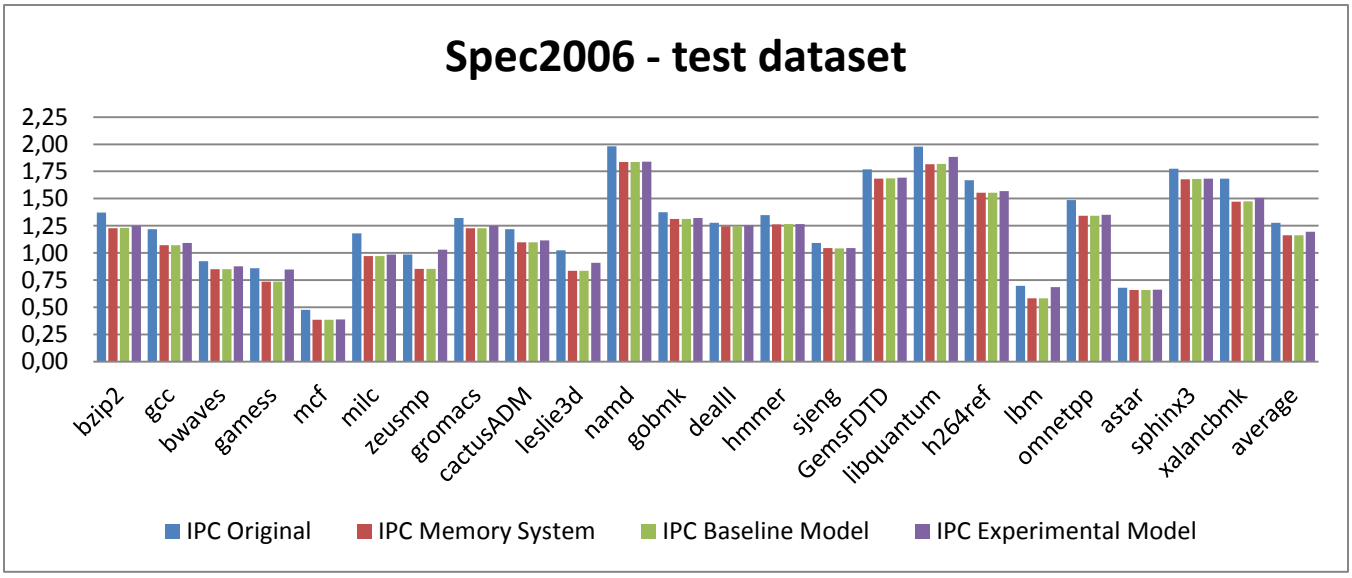


Figure 14: IPC graph for Spec2006 benchmark suite using test dataset

## *Summary*

The presented IPC graphs present the performance behavior of our benchmarks on Instructions Per Cycle (IPC) basis. The impact of the increased memory latencies is visible in almost every single case, compared to the Golden PTLsim version. The average IPC of each configuration is presented on the following table (Table 14 Average IPC). Almost in every case, the Experimental model shows a marginal improvement compared to the Baseline model.

| Configuration | Average IPC Spec2000 ref | Average IPC Spec2000 test | Average IPC Spec2006 ref | Average IPC Spec2006 test |
|---|---|---|---|---|
| Original (Golden) | 1.2582 | 1.2287 | 1.2252 | 1.2773 |

| Model | | | | |
|---|---|---|---|---|
| Memory System Mod. | 1.1188 | 1.0969 | 1.1277 | 1.1624 |
| Baseline Model | 1.1186 | 1.0968 | 1.1285 | 1.1633 |
| Experimental Model | 1.1480 | 1.1313 | 1.1635 | 1.1954 |

Table 14 Average IPC

### *Performance improvement graphs (percentage)*

These graphs present the performance deviation between Baseline and Experimental Model.

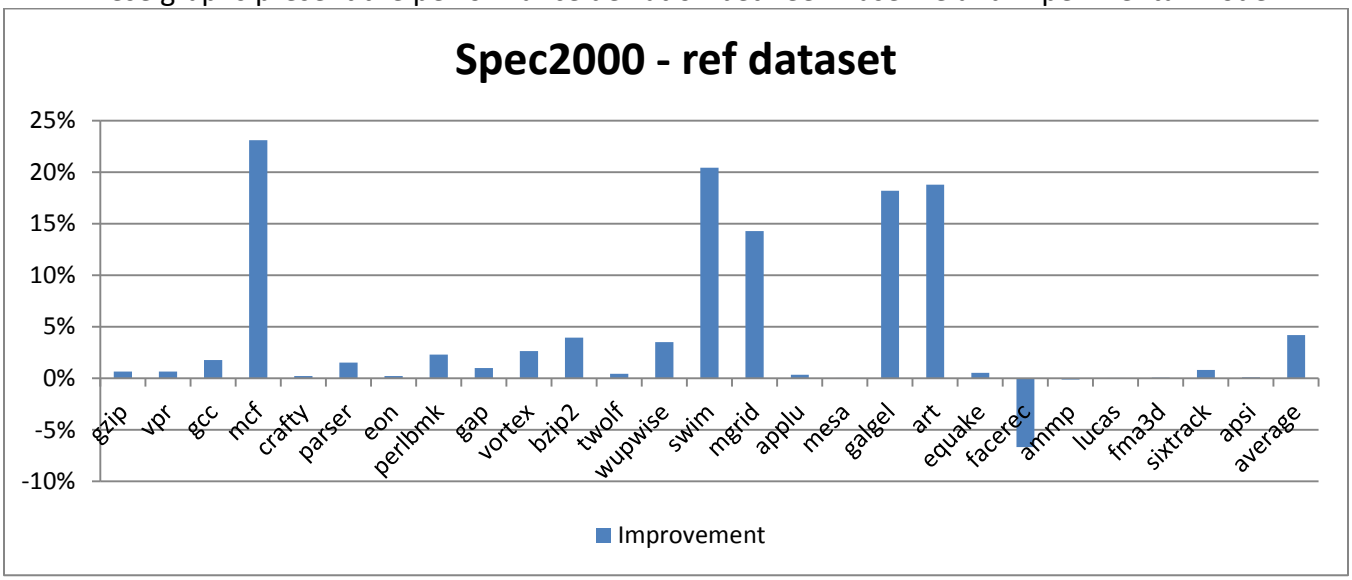


Figure 15: Deviation between Baseline and Experimental models for Spec2000 using ref dataset

Average Experimental model improvement for Spec2000 using ref dataset is 4.18%.

The observed performance degradation for facerec benchmark is discussed at section 5.4 Design evolution through simulation results along with all result evaluation and architectural design decisions during development.

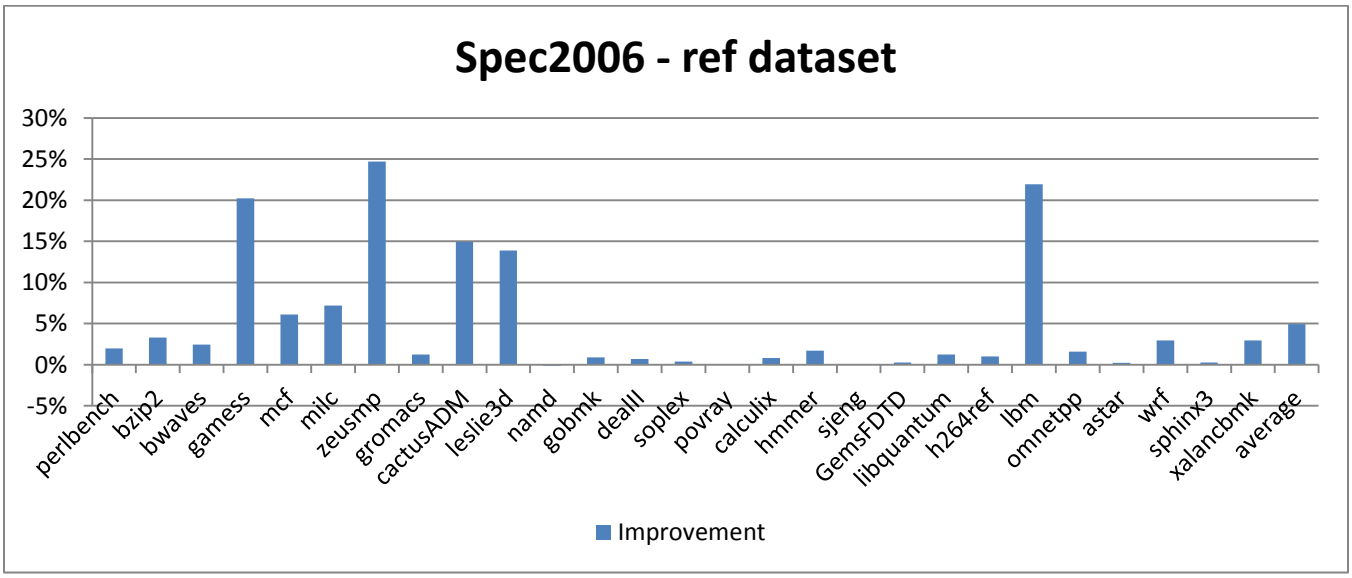


Figure 16: Deviation between Baseline and Experimental models for Spec2006 using ref dataset

Average Experimental model improvement for Spec2006 using ref dataset is 4.92%.

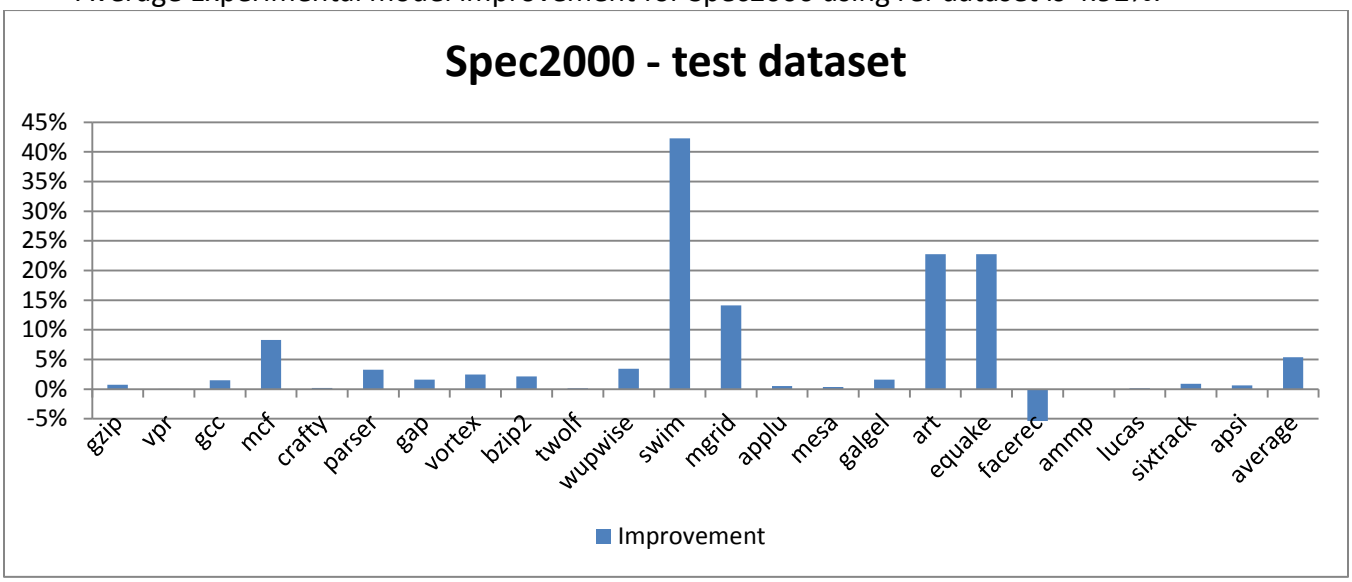


Figure 17: Deviation between Baseline and Experimental models for Spec2000 using test dataset

Average Experimental model improvement for Spec2000 using test dataset is 5.36%.

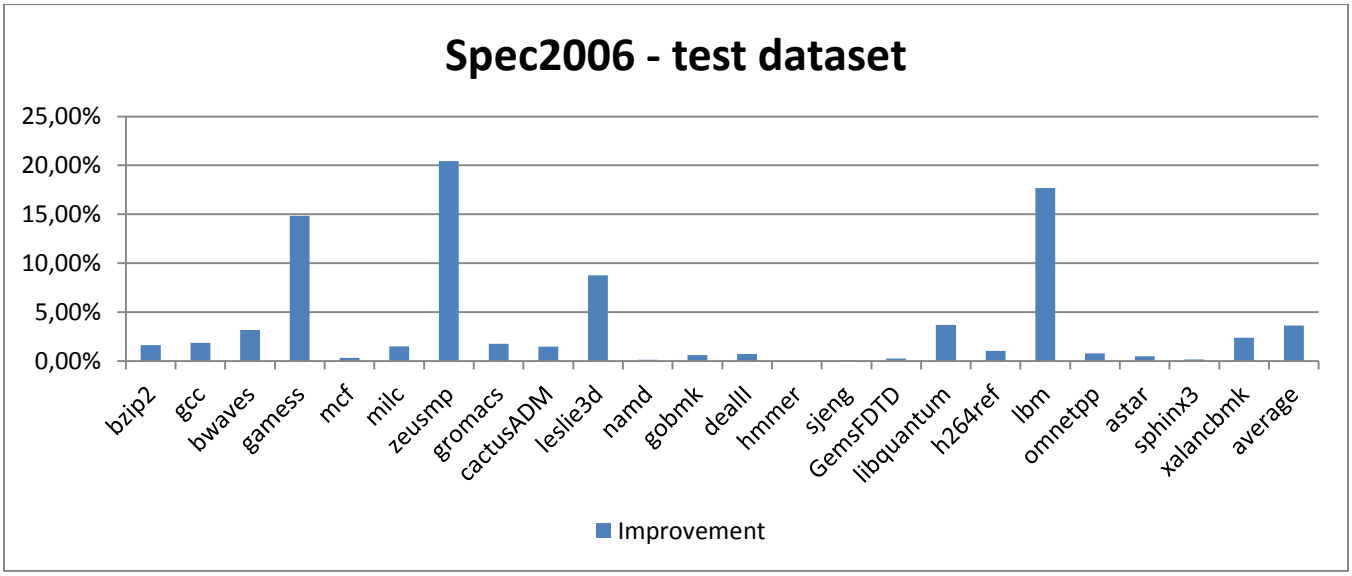


Figure 18: Deviation between Baseline and Experimental models for Spec2000 using test dataset

Average Experimental model improvement for Spec2006 using test dataset is 3.63%.

### *Prefetcher accuracy graphs (percentage)*

The following graphs present events regarding data prefetcher component. Accuracy is calculated by the number of the prefetched items that were referenced (actually used) by the processor core, divided by the total number of prefetches. These results only concern the Experimental model, since it is the only configuration model that includes a data prefetcher.

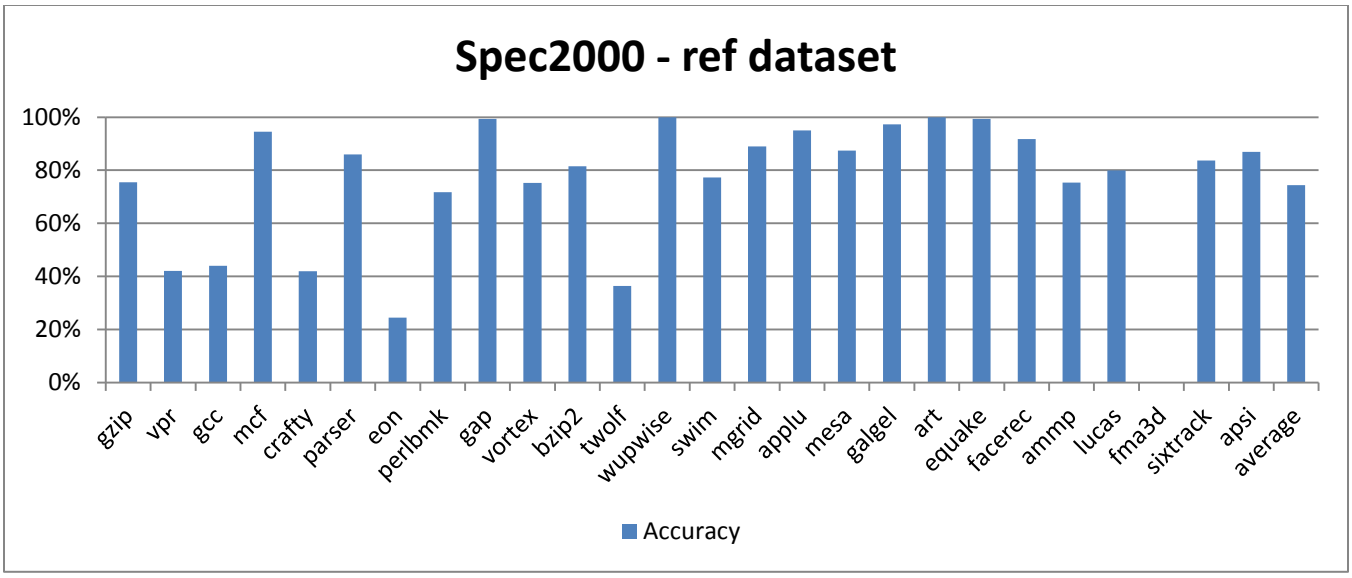


Figure 19: Data prefetcher accuracy graph for Spec2000 benchmark suite using ref dataset

Average data prefetcher accuracy for Spec2000 using ref dataset is 74%.

Regarding 0% accuracy of fma3d benchmark, all prefetch requests that were calculated and initiated by the data prefetcher were already valid memory blocks in L1 cache and therefore, they were dropped. This is possible since we have excluded a warm-up period and all data required by the program (and adopt a stride reference pattern) could be already in L1 cache memory. The benchmark reports a hit rate of 99.85%, which tends to confirm this assumption.

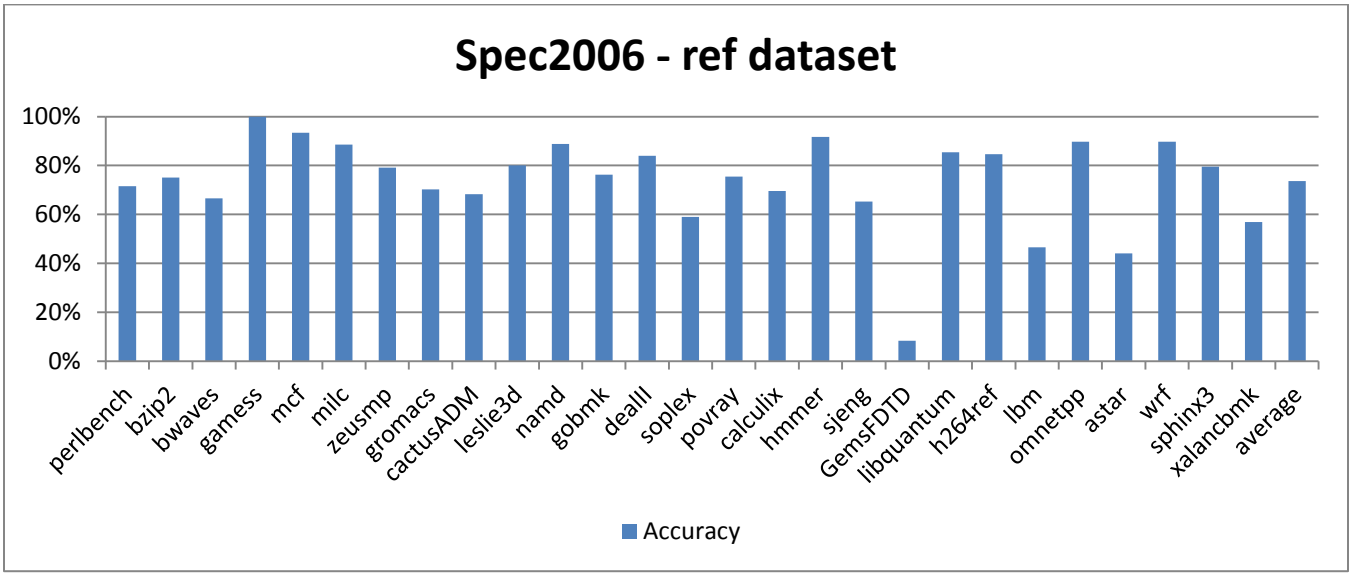


Figure 20: Data prefetcher accuracy graph for Spec2006 benchmark suite using ref dataset

Average data prefetcher accuracy for Spec2006 using ref dataset is 74%.

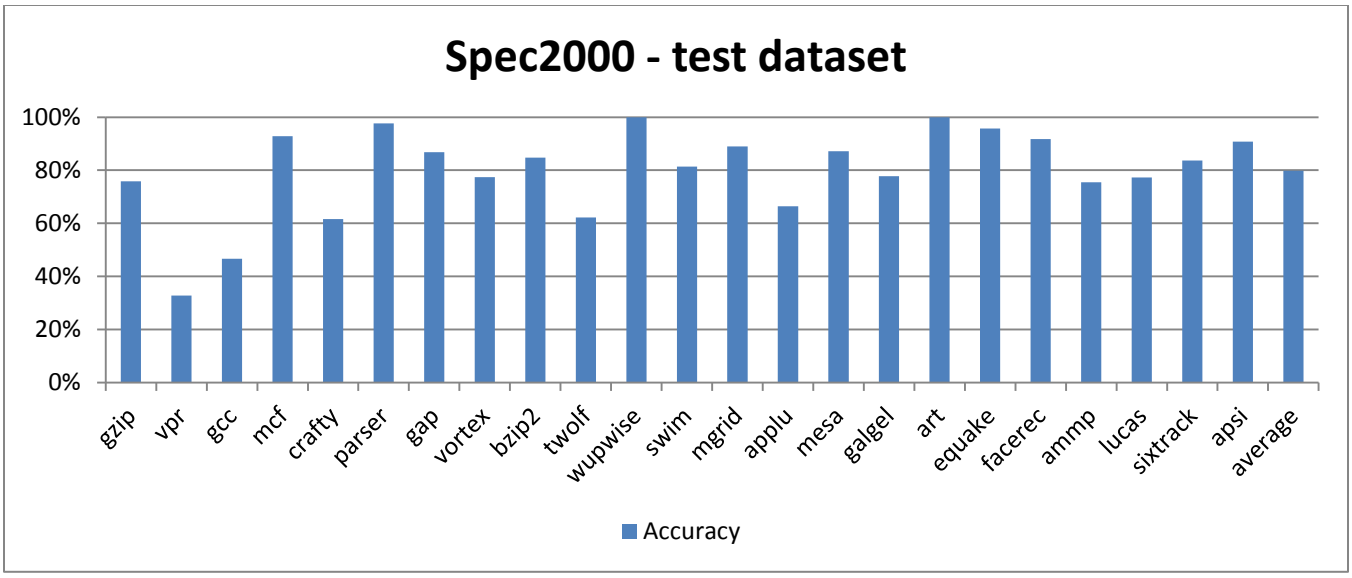


Figure 21: Data prefetcher accuracy graph for Spec2000 benchmark suite using test dataset

Average data prefetcher accuracy for Spec2000 using test dataset is 80%.

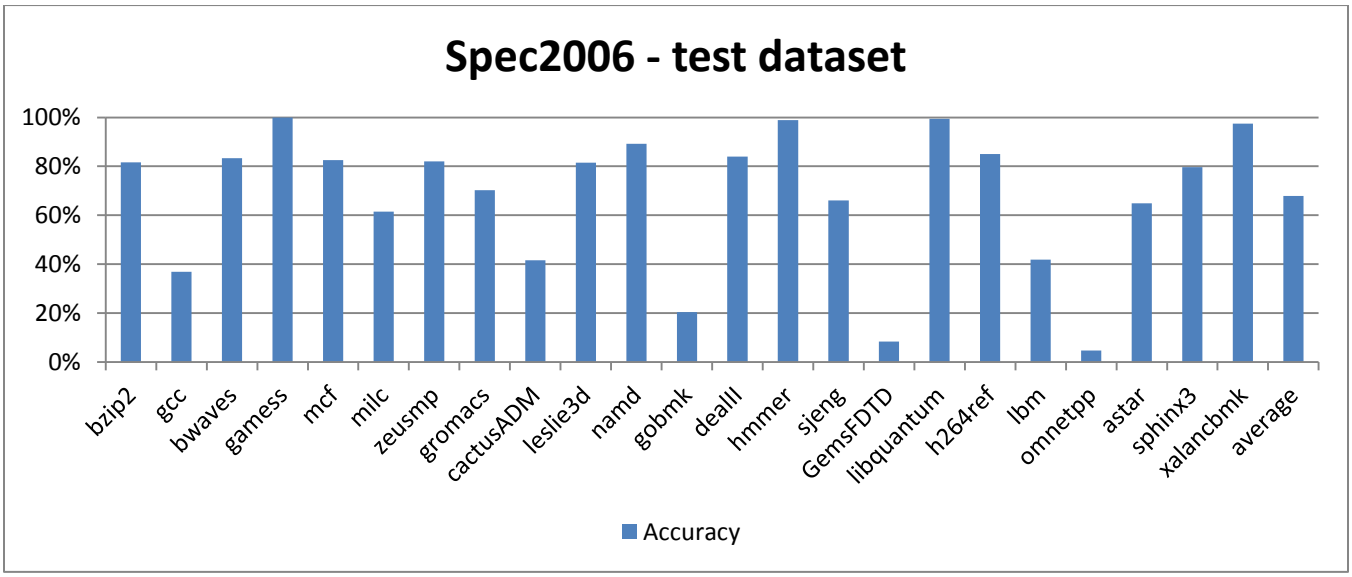


Figure 22: Data prefetcher accuracy graph for Spec2006 benchmark suite using test dataset

Average data prefetcher accuracy for Spec2006 using test dataset is 68%.

### *Prefetcher coverage graphs (percentage)*

Another metric that is used to evaluate prefetcher's performance is data coverage. It measures the portion of data miss rate covered by the operation of the prefetcher. Data coverage is calculated approximately by comparison of cache miss rate in a no-prefetcher execution. Accurate measurement is almost impossible because we are unable to track down all data transactions between memory levels and keep recording their trail. The operation of a prefetcher does however affect the overall simulation and therefore, some marginal cases can record a misleading coverage value.

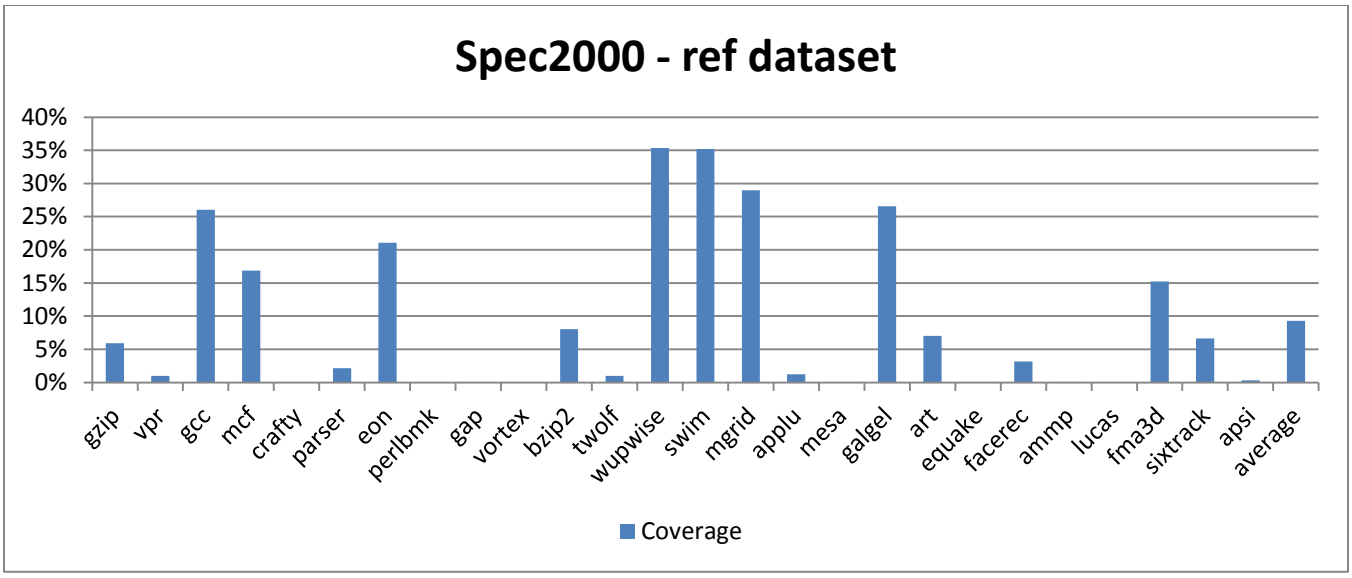


Figure 23: Data prefetcher coverage graph for Spec2000 benchmark suite using ref dataset

Average data prefetcher coverage for Spec2000 using ref dataset is 9%.

Notice that many benchmarks report high accuracy and poor coverage at the same time. This is normal while those two metrics are not directly related. A prefetcher that produces only a small number of successful prefetches (high accuracy) might not affect a high cache miss rate (small coverage).

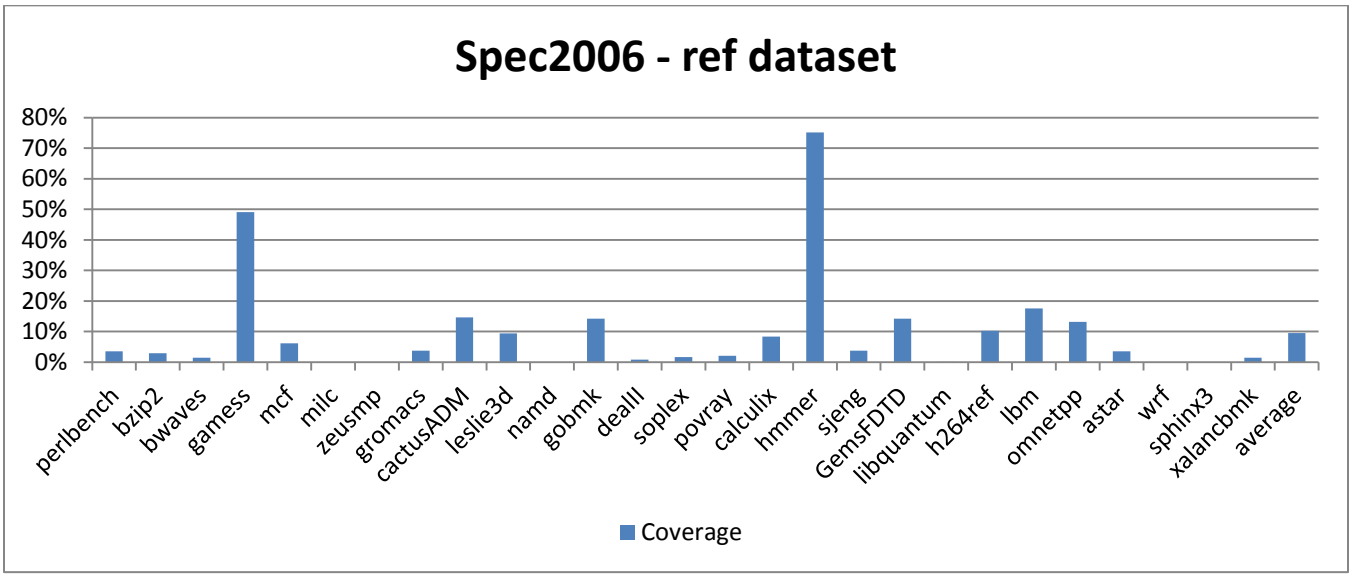


Figure 24: Data prefetcher coverage graph for Spec2006 benchmark suite using ref dataset

Average data prefetcher coverage for Spec2006 using ref dataset is 9.52%.

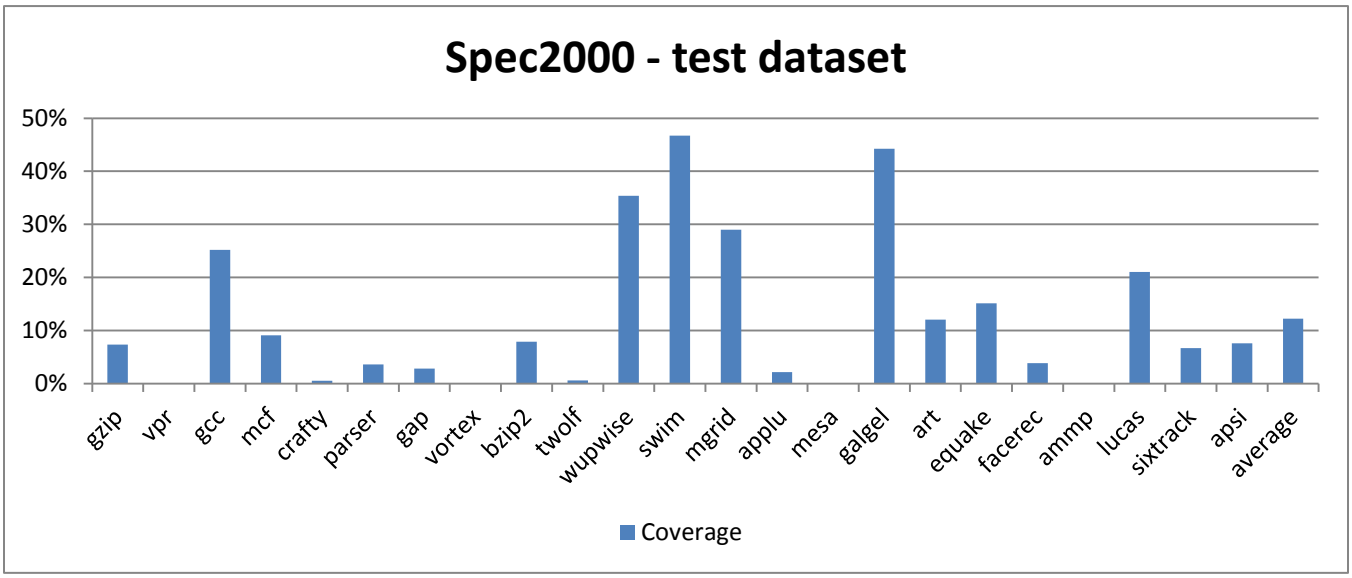


Figure 25: Data prefetcher coverage graph for Spec2000 benchmark suite using test dataset

Average data prefetcher coverage for Spec2000 using test dataset is 12%.

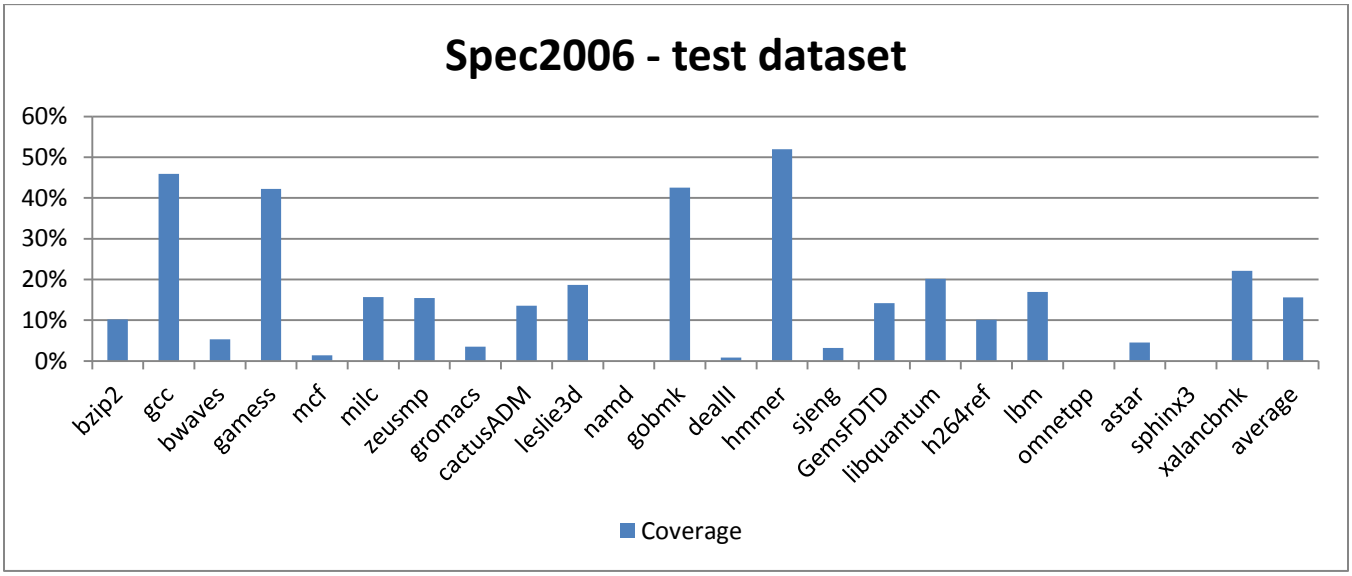


Figure 26: Data prefetcher coverage graph for Spec2006 benchmark suite using test dataset

Average data prefetcher coverage for Spec2006 using test dataset is 16%.

Experimentation reported a few cases where data miss rate increased. Data prefetches polluted L1 cache by replacing useful memory blocks. Most of these cases however also reported performance improvement. This can happen because the replaced blocks still remain valid in lower cache memory levels (L2 and L3) with relatively small miss penalty while some useful blocks are prefetched from main memory giving a significant boost on program execution time.

Pollution graphs will not be presented because they concern only a few benchmark cases and do not offer much useful information.

### *Prefetch-request service sources (percentage)*

The acceleration of a successful prefetch depends on many factors. One of the most important is in which memory level was the prefetched block found. For instance, a prefetch request that validates a block that was originally in the main memory will cause significantly more acceleration than a prefetch request that was brought by L2 memory. Although this is a quite clear metric, it does not necessarily mean that a prefetch from main memory always offers greater acceleration than a request from L2. The following graphs present the prefetch request hit distribution in the memory hierarchy.

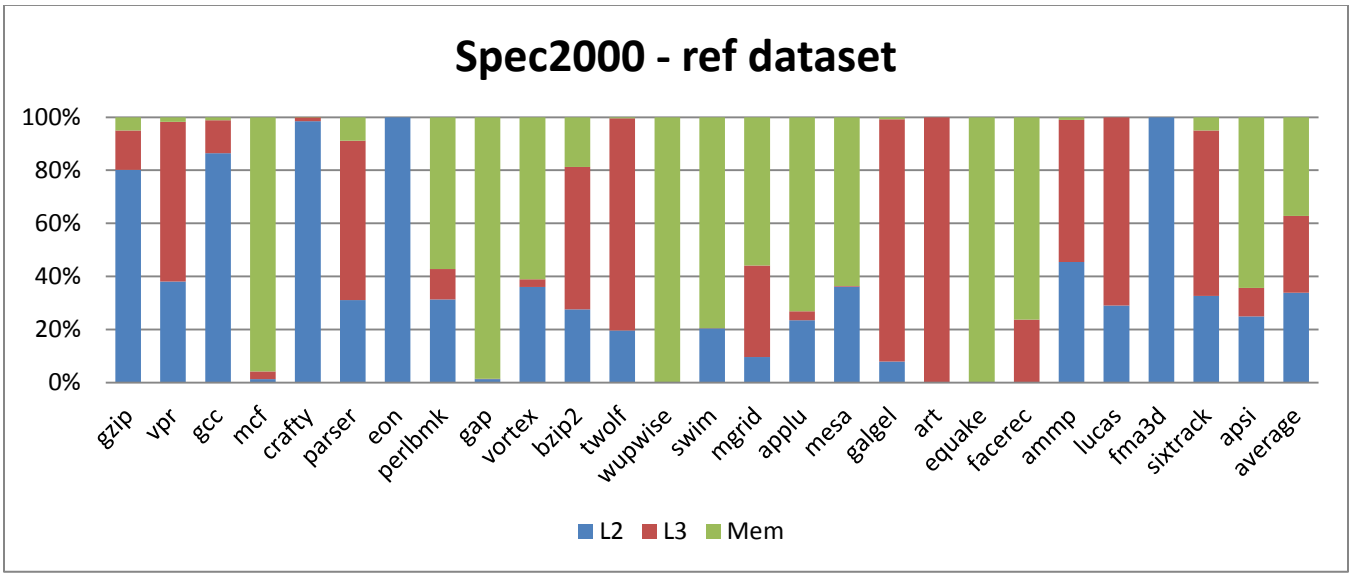


Figure 27: Prefetch request sources for Spec2000 benchmark suite using ref dataset

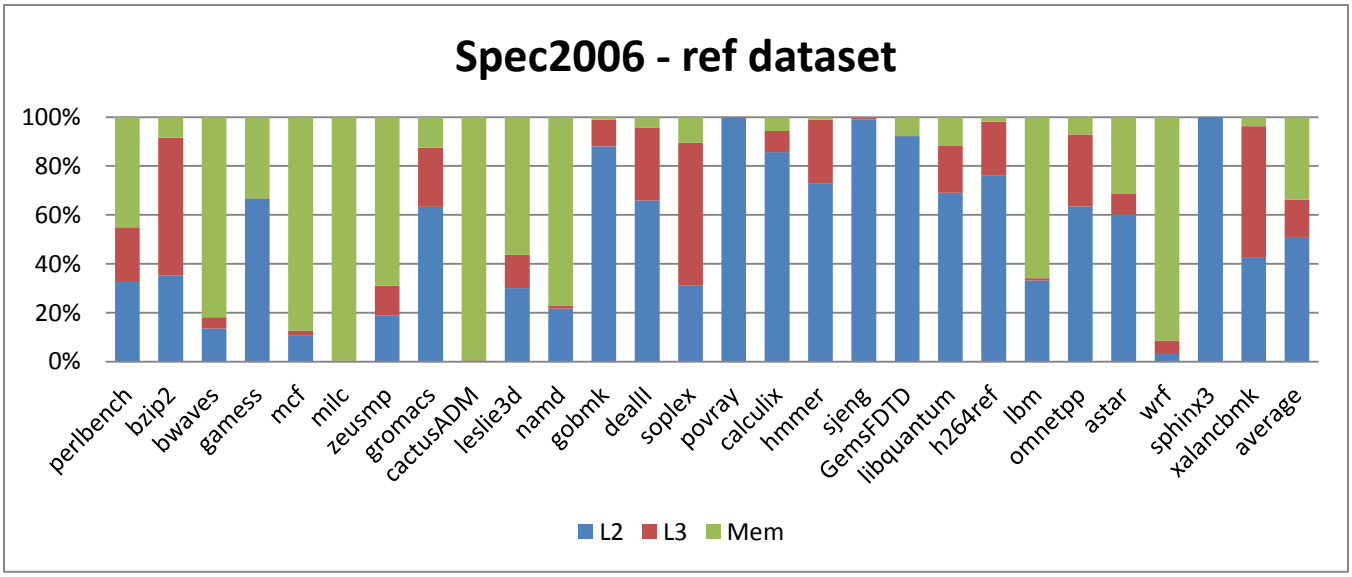


Figure 28: Prefetch request sources for Spec2006 benchmark suite using ref dataset

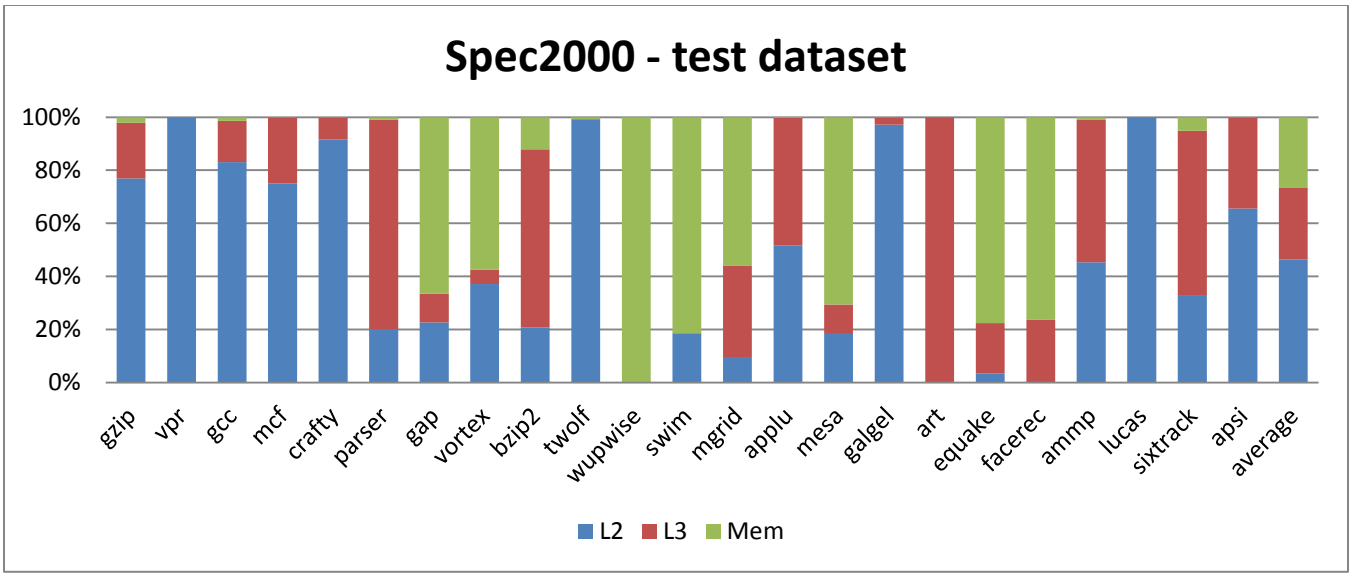


Figure 29: Prefetch request sources for Spec2000 benchmark suite using test dataset

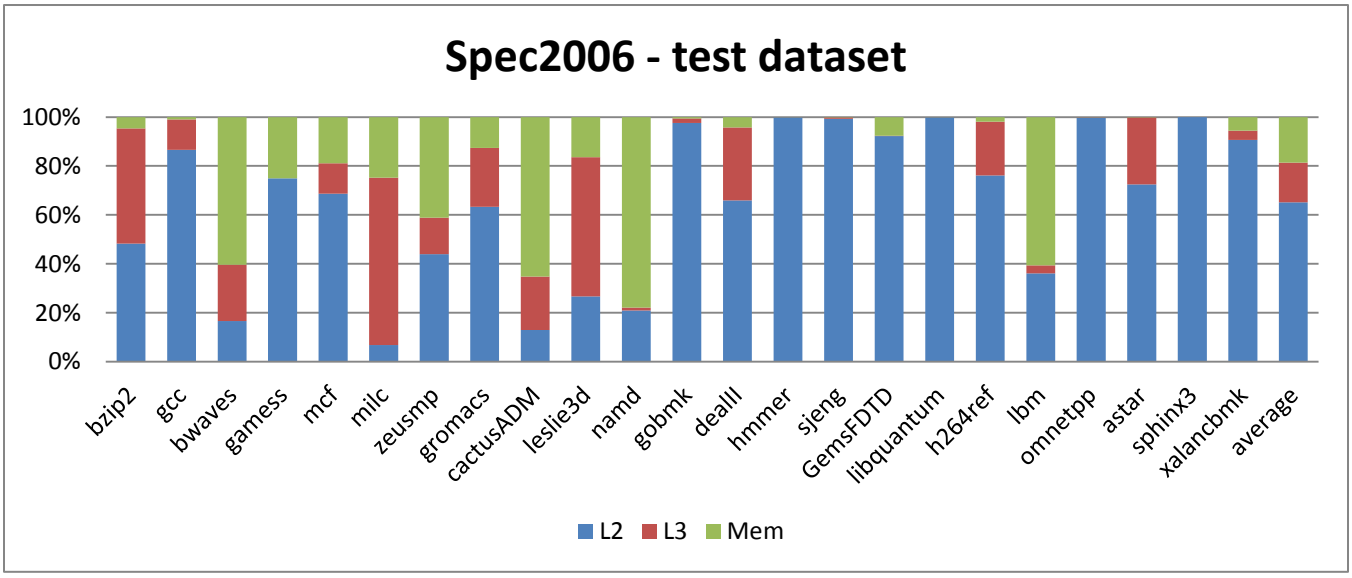


Figure 30: Prefetch request sources for Spec2006 benchmark suite using test dataset

The average distribution of each configuration is presented on the following table. Cache hit distribution, combined with prefetcher accuracy and prefetch request count can lead to useful conclusions regarding prefetcher's performance.

| Prefetch request hit | Spec 2000 ref | Spec 2000 test | Spec 2006 ref | Spec 2006 test |
|---|---|---|---|---|
| L2 | 34% | 46 % | 51 % | 65 % |
| L3 | 29% | 27 % | 15 % | 16 % |
| Memory | 37% | 26 % | 34 % | 19 % |

Table 15 Average prefetch request hit distribution

### *Cache miss rates*

| | Baseline Model | Experimental Model | Deviation |
|---|---|---|---|
| spec2000-ref dataset | 14.75% | 13.38% | -9.25% |
| spec2006-ref dataset | 11.24% | 10.56% | -6.02% |
| spec2000-test dataset | 17.30% | 15.53% | -10.23% |
| spec2006-test dataset | 15.16% | 12.43% | -17.98% |

Table 16 Average data cache miss rate

| | Baseline Model | Experimental Model | Deviation |
|---|---|---|---|
| spec2000-ref dataset | 0.28% | 0.28% | -0.43% |
| spec2006-ref dataset | 0.23% | 0.22% | -5.84% |
| spec2000-test dataset | 0.20% | 0.18% | -8.36% |
| spec2006-test dataset | 0.21% | 0.20% | -1.46% |

Table 17 Average instruction cache miss rate

There are some benchmarks, which are not included in the presented results. Specifically, for the spec2000 suite and the test dataset, benchmark eon, fma3d and perlbmk don't run for 1B committed x86 instructions, while for the reference dataset all benchmarks executed. On the contrary, for the spec2006 suite and test dataset, benchmarks perlbench, soplex, povray, calculix, tonot and wrf don't run for the specified simulation window, while for the reference dataset gcc and tonto fail to execute for 1B committed x86 instructions. Therefore, averages are calculated only for the benchmarks that are fully executed.

## 5.3 Expanded 10-billion simulation results

Spec2000 and Spec2006 benchmark suites include a fairly large variety of benchmark programs for evaluating processor performance. Those programs are planned for a significantly long execution period with many phases and are accompanied by two different input datasets. A simulation can only observe a really small portion of this execution while the ratio between simulation time and execution time is greater than 1.000 to 1 (approximately the simulator achieves simulation clock of 100 KHz to 1.0 MHz on a host system clocked at 4.0 GHz). Reference dataset is a representative dataset that matches a real-world scenario while test dataset is more of a short summary for that. Test dataset aims to go through all execution phases in a shorter period of time.

According to the results in the previous subsection, the dataset itself seems to be a critical factor of a benchmark performance in the same execution period. This is due to two main reasons. First, as already stated, test datasets aim for a shorter execution and therefore are likely to go through more execution stages in the same period of time and second, input data may vary on size and patterns that somehow can significantly affect memory dependencies (cache miss rates have different values between two datasets on the same benchmark).

In our effort to achieve a better and more objective image for our expansion project, we decided to add samples of expanded simulation and observe the behavior of certain benchmarks going through more phases of the executed datasets. The expanded simulation period would be a ten-billion x86 commits simulation with one billion commits warm-up period. Selection of the benchmarks was based

on many factors and finally, programs with somehow strange or unexpected behaviors in their first simulation were chosen. This subsection presents the results of these benchmarks executed for 10 billion x86 instruction commits for comparison with the one-billion benchmarking results.

### *IPC*

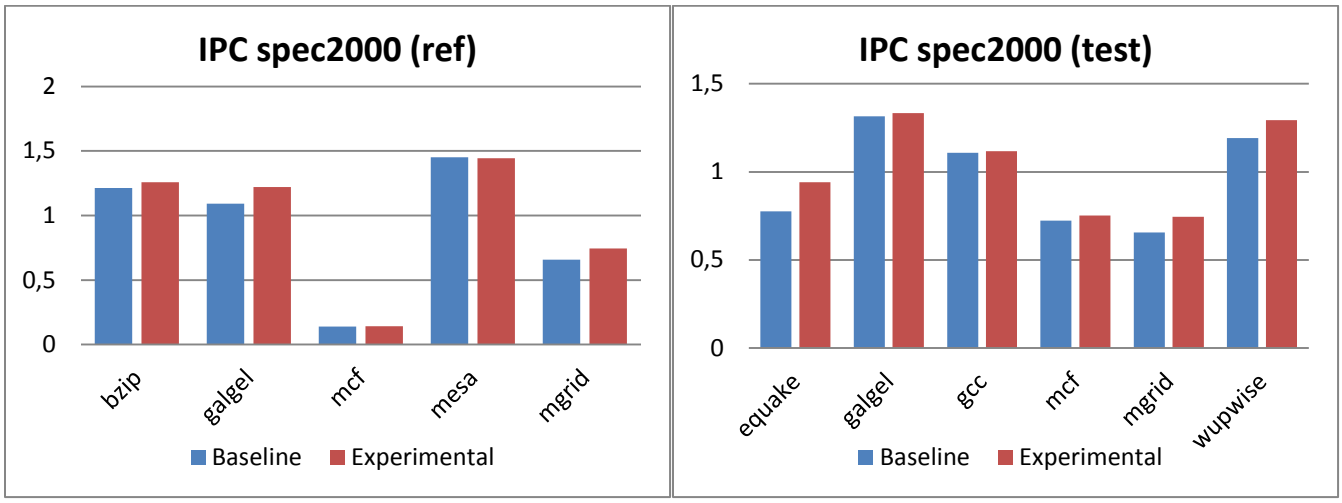


Figure 31 IPC for Spec2000 10-billion x86 commit simulation

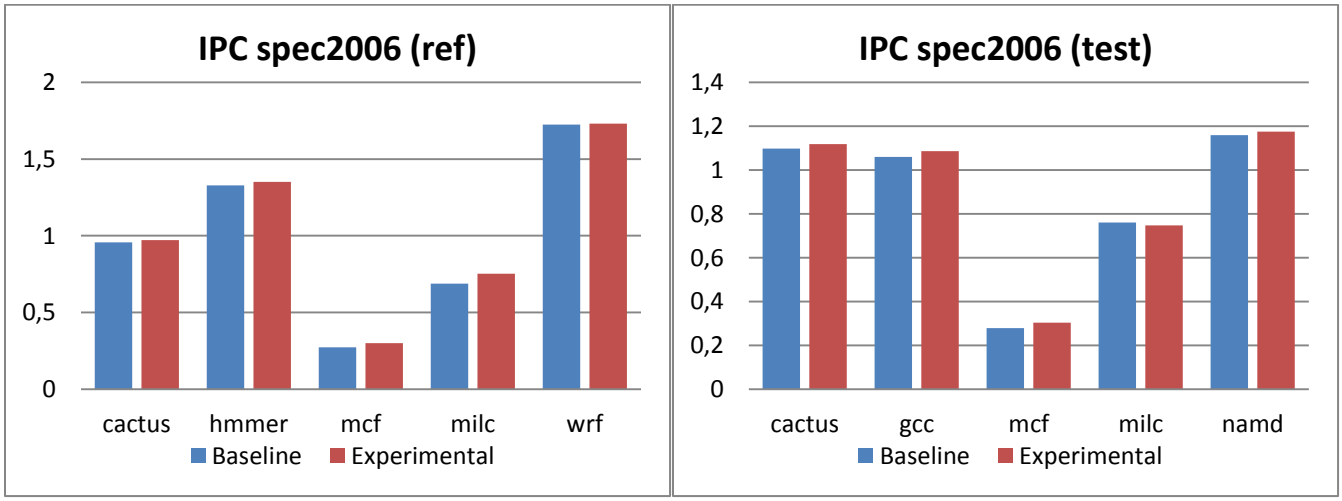


Figure 32 IPC for spec2006 10-billion x86 commit simulation

Regarding the test dataset, the following benchmarks: equake, galgel,mcf for Spec2000 suite and cactus, gcc, mcf for Spec2006 suite had a complete execution before reaching 10 billion instruction commits. In most of these cases, warm-up period covered a large or even the largest portion of the simulation time. For this reason it was not excluded like the rest cases but instead, statistics during the first billion instructions were recorded and are included in the following results.

### *Deviation*

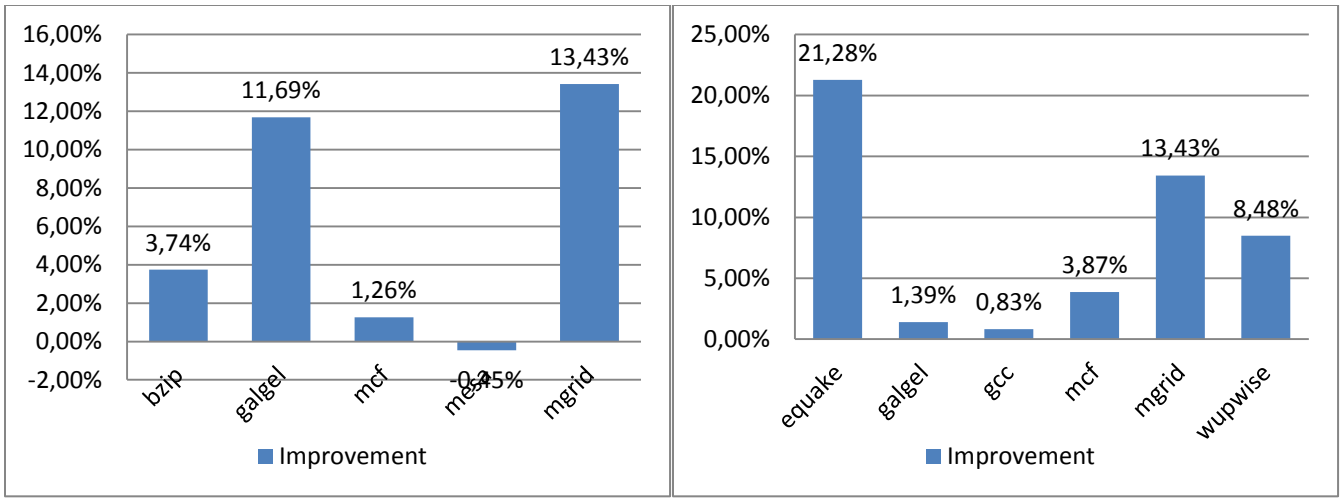


Figure 33 Deviation for Spec2000 10-billion x86 commits simulation

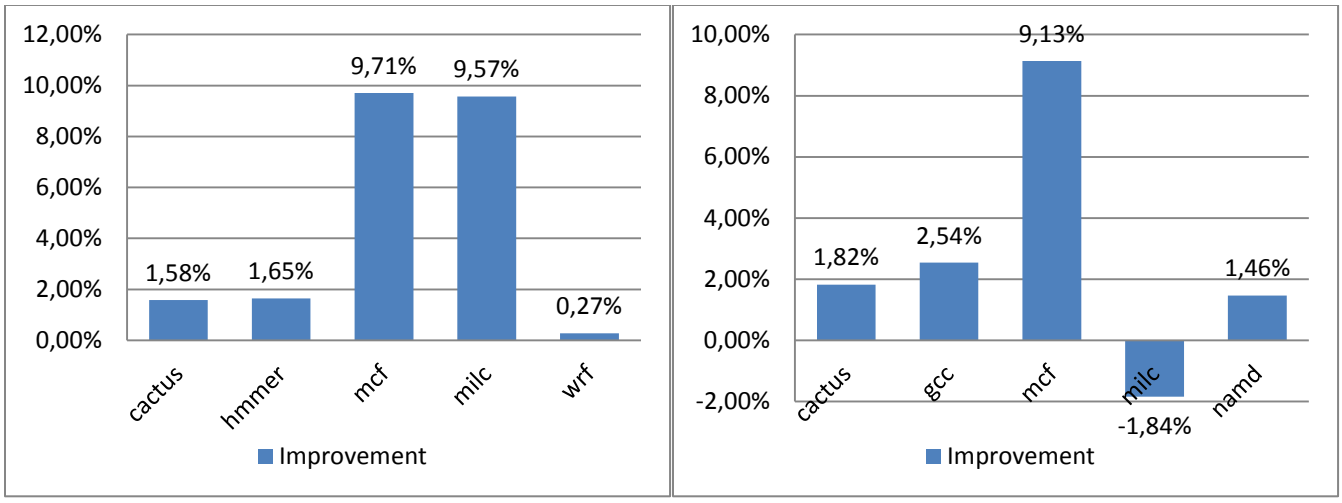


Figure 34 Deviation for Spec2006 10-billion x86 commits simulation

## 5.4 Design evolution through simulation results

The initial expansion design only included the addition of both prefetcher components and BPU modifications. Experimentation revealed cases where hardware limitations were not taken into account and indicated architectural design misstatements. This led to new design decisions which included attribute adjustments and unit modifications in both existing and newly added components. The final design, as described in this report, is a result of evolution through this process. This subsection refers to some discrete stages during development along with some simulation results and analysis.

Something that should be noted is that all results presented in this subsection concern alpha testing and no standard simulation patterns were used. Unlike previous sections, simulation time is much shorter, different host machines with different operating systems were used, shorter number of benchmarks was simulated and in most cases only one dataset was used as input. In addition, more configuration models were used for covering isolated components. These results should not be used for evaluation but only for understanding the components' behavior.

### *Initial design*

After completion of the prefetcher components and the cache access control mechanism, the initial model was used to simulate 13 benchmarks of Spec2000 suite for 70 million instruction commits in order to start evaluating the changes. At this point, prefetchers did not have an input queue and prefetcher notifications led directly to training. Memory latencies also had the default values.

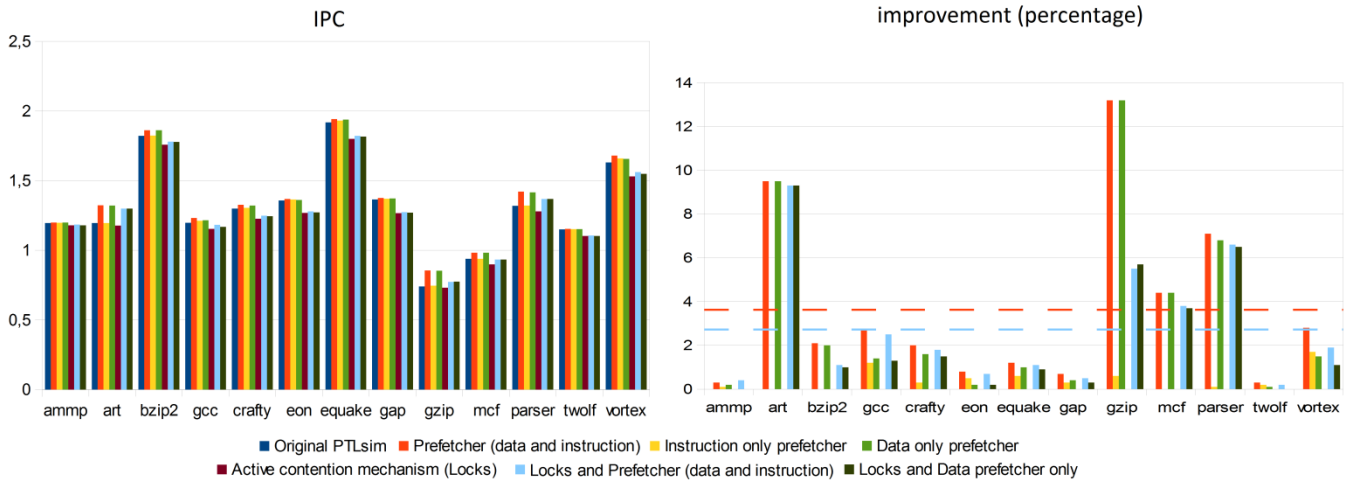


Figure 35 Initial design stage simulation results

Simulation involved 8 different configuration models separated in two main setups: Active cache access control and Deactivated cache access control (also referred as “Locks”). Each includes the original PTLsim model, a configuration with data prefetcher, a configuration with instruction prefetcher and a configuration with both prefetchers active. The Locks + Instruction prefetcher results were excluded from the graphs while they did not offer useful information. The two horizontal lines in the Improvement graph represent the mean values.

### *Miss-buffer coupling, memory latencies fix and prefetcher input queue*

The next stage involved the Miss Address Buffer coupling to each cache memory (as described in section 4.1.1 Coupling dedicated MABs to cache memories) along with the addition of input queues to hold prefetcher notifications (since there are more than one address generation units in the model and multiple notifications can occur on the same clock cycle). The initial MAB sizes were: 8 entries for L1 instruction cache, 16 entries for L1 data cache, 32 for L2 cache and 32 for L3 cache. Also, after consultation, memory latencies were fixed to simulate a total access delay of: 12 cycles for L2, 40 cycles for L3 and 200 cycles for main memory.

After performing the same 70-million commits simulation and with some promising results, we moved to a more complete simulation pattern with full Spec2000 and Spec2006 suites and 100 million commits simulation time. Surprisingly, the use of ref dataset revealed some interesting issues in our design which were not observable with the test dataset and the small number of benchmarks used. The following figure shows the performance deviation compared to the baseline model (without prefetcher) for the spec2006 suite and ref dataset for a 100 million commit simulation.

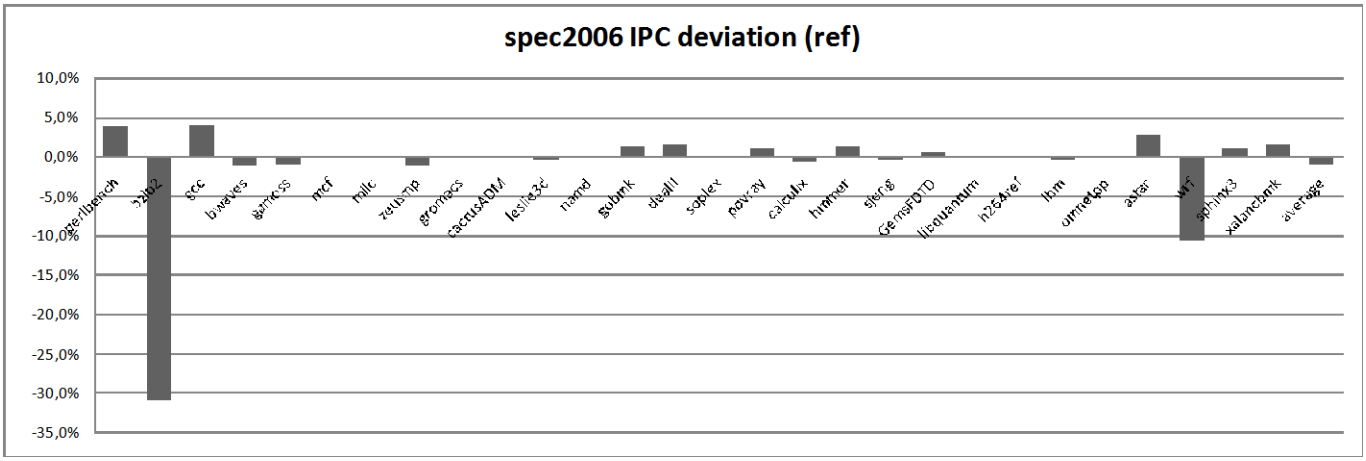


Figure 36 Stage 2 Deviation results for Spec2006 and ref dataset

Despite the fact that the overall impact of the design was far worse than expected, the cases of bzip and wrf benchmarks are far from normal results. Both benchmarks reported better cache hit rate with active prefetchers which means that the components did not harm by polluting the cache memory. Investigation revealed that bzip reported a seriously high amount of MAB-full events inside the core. The larger memory latencies in combination with the nature of bzip (which requests a lot of data from main memory) and ref dataset led to a high utilization of the MAB during execution, which in many cases reached 100%. The activation of prefetcher pushed this already fragile situation beyond the model's capabilities. Even though the core has higher priority than the prefetcher on the memory accessing, the prefetcher could still get idle cycles and perform its requests, which held MAB entries for long time (most requests were served by the main memory). Finally, all these events led to this extreme performance degradation. Similar situations were also reported in other benchmarks but in a smaller degree. At this point, the decision to double the L1 data cache MAB size was taken to fix such behaviors. In addition, L2 and L3 MAB sizes were incremented in order to be able to fit both L1 caches and avoid full-MAB events in lower levels. In addition, the input queue for the prefetchers was doubled to 8 entries in order to reduce the relatively high number of notification drops. These changes fixed the described behavior and bzip.

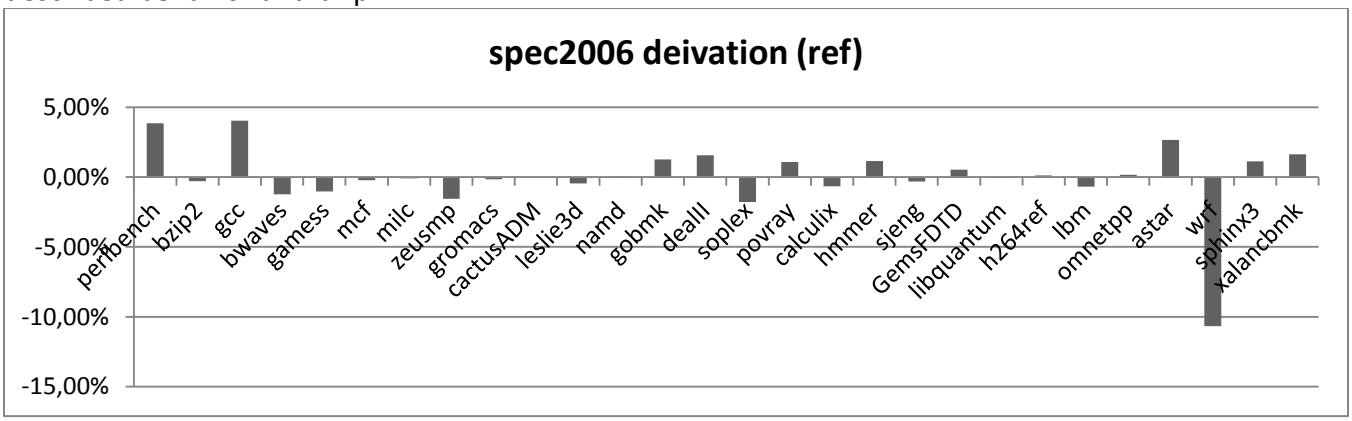


Figure 37 Stage 2 results for spec2006 ref dataset, after increasing L1 data cache MAB size

Regarding the overall results, a test dataset simulation of 100 million was performed for both suites. The results were again back to normal and improvement was within the expected limits. This large difference between the two datasets could not be easily explained and after further

experimentation we concluded that the observable window of the initial 100 million instruction commits was not objective for evaluation. Our testing patterns moved to longer simulations of one billion commits with a 100 million commits warm-up period.

### *Complete prioritization*

The updated design fixed most abnormal behaviors. However wrf benchmark continued reporting extensive performance degradation. The special about this specific benchmark was that it was producing a very large amount of prefetch requests but without polluting cache memory. This was the only case where the prefetcher was actually blocking the processor core from accessing the cache and caused pipeline stalls. Even with the cache access arbiter giving higher priority to the core, the prefetcher could still indirectly block out the core through the Miss-buffer. The excessive amount of prefetch requests led to blocking of the core's demand requests when those (prefetches) arrived and the Miss buffer was enabling them to L1 data cache. This rare case led to the decision of separating MAB entries to prefetch and demand. Prioritization was applied to all components related to the prefetcher. As expected, wrf's degradation was fixed and in addition there was a noticeable impact on other benchmarks as well. At this point, the model was close to the final design regarding the cache subsystem.

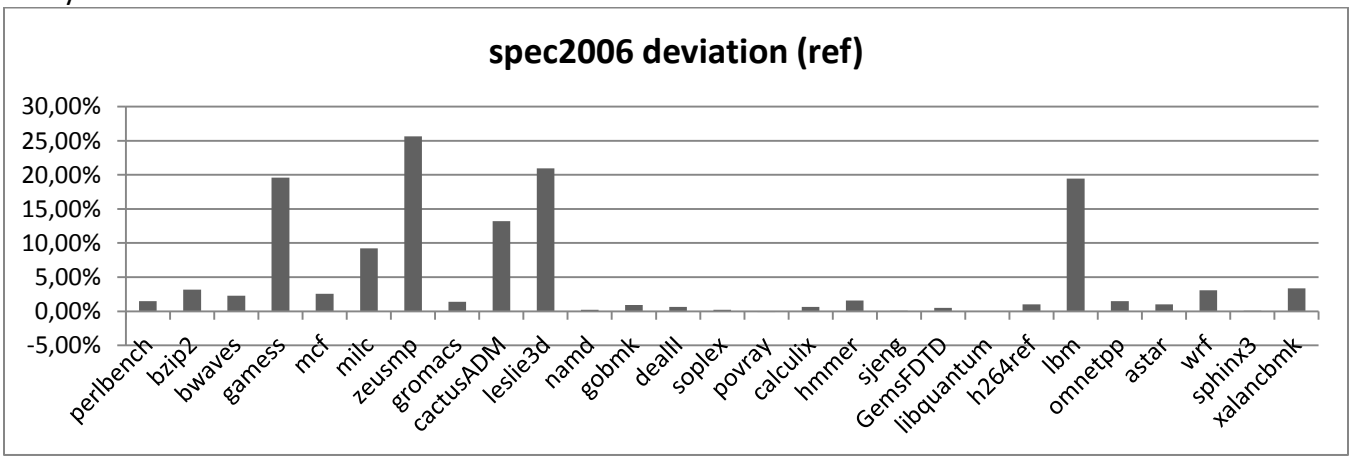


Figure 38 Stage 3 results for spec2006 ref dataset after complete prioritization – fix of wrf's performance degradation

### *BPU modifications and commit-store bug fix – Final design*

The final stage included the Branch Prediction Unit modifications and the commit-store bug fix. Not many changes were initiated during this stage and the final result was completed as planned. The final results are marginally different regarding performance compared to the previous stage. However, there are significant differences on the memory subsystem statistics, mostly due to the bug fix, as already referred on the bug's impact at Appendix A - I.

The only remaining alarming result is the performance degradation for the facerec benchmark. This benchmark reports a cache hit rate improvement, healthy memory traffic, no drops or stalls due to the memory subsystem and a total performance improvement at memory accessing. In contrast, there is 7% performance degradation. Our extensive investigation of statistics and events during execution has given a likely answer: the benefits gained by the prefetcher's operation significantly increase the aliasing and re-dispatch events inside the core (more than 6 times), causing a major pile-up on the core's frontend. A combination of instructions is possible to cause such behavior when some

of the memory data are accessed faster (due to the prefetcher). However, we cannot know if facerec actually has such combination of instructions and we are unable to confirm this assumption.

PTLsim maintains a Load Store Queue for resolving load/store dependencies. In some cases, the addresses of one or more prior stores that a load may depend on may not have been resolved by the time the load issues. Instead of stalling the load until all prior addresses are known, PTLSim aggressively issues loads as soon as possible. In addition it implements load/store aliasing prediction technique to avoid issuing loads that frequently alias to another store inside the pipeline. Compared to stalling, this method has an overall performance improvement.

Disabling this feature and modifying the core to cause stalls instead of issuing unresolved loads could enlighten us in the case of facerec. However, this requires extensive modification inside the core and since it is related to a pipeline feature, it is considered out of scope and was rejected.

# 6 Conclusions

We modified a widely used x86 cycle accurate architectural simulator and enhanced it with additional components. During development, our primary concern was to keep the design within realistic boundaries and adjust all the modified parts to replicate commercial implementations. Goal of the resulting model was to reduce the gap that existed regarding performance units between the PTLsim processor model and modern x86 implementations.

We successfully integrated two L1 prefetcher components (data and instruction) along with an arbiter for cache memory accessing. In order to capture the impact of the additions, changes were required inside the cache memory subsystem. A dedicated Miss Address Buffer was coupled to each cache memory to resemble a real hardware model. Finally, we corrected some misstatements regarding the branch prediction unit and added an extra branch target buffer for supporting conditional and direct branch operations.

Our updated model expands one of the few tools that correspond to this widespread commercial computer architecture (x86). A short summary of the updated design's results:

- The average performance speedup (experimental model over baseline model) in SPEC2000 for the test dataset is 5.36%, while for the SPEC2006 is 3.63%. Regarding the reference dataset in SPEC2000 the average IPC improvement is equal to 4.18%, while for the SPEC2006 is 4.92%.
- The integration of the data and instruction prefetchers in the x86 microprocessor model reduce the average miss rate in both the instruction and data caches.
- The accuracy of the data prefetcher on average among all benchmarks is 73.97%.
- The coverage of the data prefetcher on average among all benchmarks is 11.65%.

The impact of our expansion matches real hardware implementations and is close to our expectations. Both prefetcher components can be easily modified to implement different prefetching schemes and could also be used for experimentation in this area. All modifications were made in a way that can support future expansion and additional components (for instance, Level 2 prefetching components). In addition, some design and implementation misstatements were corrected and the overall reliability of the simulator was increased.

# 7 References


1. Jerry Banks, J. S. C. B. L. N. D. M. N., *Discrete-Event System Simulation* (Prentice Hall).
2. Yourst, M. T., *PTLsim User's Guide and Reference*, 2nd ed. (2007).
3. Gupta, A. . H. J. . G. K. . M. T. a. W. W.- , *Comparative Evaluation of Latency Reducing and Tolerating Techniques*, presented at 18th International Symposium on Computer Architecture, Toronto, Ont., Canada, 1991.
4. Smith, A. J., Cache Memories. *Computing Surveys* (1982).
5. Mowry, T. C. . L. S. a. G. A., *Design and Evaluation of a Compiler Algorithm for Prefetching*, presented at Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, 1992.
6. Kyle J. Nesbit, A. S. D. J. E. S., *AC/DC: An Adaptive Data Cache Prefetcher*, presented at 13th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA , 2004.
7. Kroft, D., *Lockup-free Instruction Fetch/prefetch Cache Organization*, presented at 8th International Symposium on Computer Architecture, Minneapolis, 1981.
8. Mowry, T. a. A. G., Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors. *Journal of Parallel and Distributed Computing* **12** (2), 87 - 106 (1991).
9. Steven P. Vanderwiel, D. J. L., Data prefetch mechanisms. *ACM Computing Surveys* **32** (2), 174-199 (2000).
10. Dahlgren, F. . M. D. a. P. S., *Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors*, presented at International Conference on Parallel Processing, St. Charles, 1993.
11. John L. Hennessy, D. A. P., *Computer Architecture: A Quantitative Approach, 4th Edition* (Morgan Kaufmann, 2006).
12. Santhosh Srinath, O. M. H. K. Y. N. P., *Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers*, presented at 13th International Symposium on High Performance Computer Architecture, Washington, DC, USA, 2007.

# Appendix A - Miscellaneous simulator modifications

## I. Commit-store bug

Extensive experimentation revealed a critical bug in the original PTLsim source code with major impact on the data cache subsystem. Due to the nature of this bug, technical details are included in order to make a comprehensive documentation.

### *Cache memory implementation in PTLsim*

PTLsim implements a concept where the cache subsystem only maintains information about cache memory structures. The actual data is not stored. Instead, it (data) is stored on the host machine's main memory. This means that a possible error in the memory design will not fail the execution correctness, but will only affect the performance. Each implemented cache memory is consisted of a set-associative array of Cache-Line entries. Line selection is made as in a normal cache memory (using block offset for selection, pseudo-LRU for way etc.). Cache-Line structure maintains statistical information and a valid bit vector, while the tag information and LRU state bits are implemented on the set-associative array structure. The valid bitmap assigns a bit for each byte within the cache line, indicating whether a byte is valid or not.

Upon a memory reference, the cache memory subsystem is checked for the corresponding memory block. L1 data cache is probed first and upon a hit, the valid bit mask is checked to identify whether the requested bytes within the block are valid[14]. If they are, the request is normally served. Otherwise, the subsystem emulates the reading of the main memory by seeking the requested line to lower cache levels through the miss address buffer.

Memory transactions are done in block size. However, this does not apply to store instructions because the upper size limit of a store instruction is a quad-word (equals to 1/8 of a memory block). When a store instruction commits, it enables the referred cache line and validates the bits that correspond to the offset inside the block. If the line has any invalid bytes, it is requested through the Miss Address Buffer in order to complement the remaining missing bytes. In the meanwhile, only those valid bytes (that were committed by store) can be accessed through a load instruction.

The practical meaning of this is that when a store instruction references a memory line which leads to a cache miss, it should partially enable it and request this line from lower memory levels in order to complement all the remaining missing bytes.

### *Bug description*

The way this is implemented requires a newly allocated memory line to be reset zero-initialized by the store commit side. However, the associative array implementation does not satisfy this requirement. It does not reset the Cache Line structure upon new allocation or replacement; it only updates the tag and LRU state bits instead. Therefore, dirty data remain in the structure from the previous entry. This means that when a new line is allocated and replaces another, it will still keep the old valid bitmap along with the statistical data. In the most likely case, this bitmap will be fully valid (since design states that partially valid lines are requested through the memory system to complement remaining data).

[14] A cache line may not be fully validated.

When a store instruction commits, it accesses the L1 cache memory to validate its data. If the corresponding cache line is not already valid, it is allocated (replaces an old entry) and store-commit validates the bits on the bitmap that correspond to the bytes of the store address and size. It then checks if the bitmap is fully valid and in opposite case, it initiates a miss in the MAB to recover all remaining data. However, because of the remaining dirty valid bits that we described in the previous paragraph, this almost never happens. Store-commits act like cache hits almost in every single case. This can be observed at the statistics tree /dcache/prefetches, which counts how many times a store instruction requested a prefetch of a line in order to complement its missing data. We have noticed that this count is almost identical in every simulation, no matter how long the simulation was or what program was simulated.

### *Bug impact*

The described bug has a critical impact mainly on the memory subsystem. A significant part of the memory traffic is skipped due to the lack of store cache-miss events. This applies to all levels of cache memory and the Miss Address buffer. It also affects the data consistency between cache levels. The cache hit rates and miss rates that are recorded relate only to load instructions.

Performance is also affected but to a lesser extent. Store misses do allocate a MAB entry to recover the remaining cache line, which implies that in the worst case, all data will be valid within 200 cycles (main memory latency). Requests to mistakenly validated bytes are incorrectly served only within this short period. This means that these very rare cases can lead to 200-cycle acceleration each at most, against the correct approach.

### *Fix*

The fix for this bug is either to prevent dirty-bit dependences for the store commit or reset of the structure in cases of replacement at the associative array implementation. The first approach fixes the described behavior only, while the second fixes the associative array implementation and may also affect other instances of associative arrays.

The second approach (fixing the associative array implementation) was designated hazardous since many instances of this structure exist and it is unknown if any of these actually takes advantage of this dirty-data property. In order to avoid possible side effects, we decided to choose the safe approach of fixing the commit store part.

The referenced line is probed at the beginning, before validating or choosing any data and decision of whether it should be prefetched or not depends on whether the line is active on L1 data cache or not, instead of examining the valid bitmap. In a case of cache miss and MAB allocation, the newly activated line resets its valid bitmap before validating the store's corresponding bytes. This successfully fixes both memory system impact (by initiating prefetches when required) and performance impact (by invalidating mistakenly valid bytes).

## II. Statistics tree and new command line options

The updated simulator model required some expansion of the statistical tree in order to observe new events and fix a few old. Furthermore, some additional command line options were added for achieving desired simulation scenarios. These changes are presented on this subsection, beginning with the newly added simulation options that are passed to the simulator through the command line.

A brief reference on the PTLsim statistics and analysis is required in order to successfully understand our additions. PTLsim implements a statistics data store mechanism for recording statistical events and data points during simulation process. The file format supports storing multiple regular or triggered snapshots of all counters, which can be later be subtracted, averaged and extensively manipulated. Statistics are maintained as a tree of nested structures. The root holds statistical information concerning the simulator and there are main branches/subtrees for the instruction decoder, the out-of-order core, the data cache subsystem etc. The stats file is parsed by the separate ptlstats executable. To select a specific snapshot, the –snapshot <name-or-number> option is used. To select a specific subtree of interest, the –collect <path> option must be used. Path must be absolute and its syntax is similar to a filesystem path format, using slash "/" for branch separator. An example for reading statistics of final snapshot concerning the out-of-order core's fetch stage follows:

ptlstats –snapshot final –collect /ooocore/total/fetch testfile.stats

**Simulation option: -snapshot-warmup <warmup>:**

This option takes a statistical snapshot by the name "warmup" after <warmup> clock cycles of execution. Simulation of a program execution goes through several stages. The initial stage where cache memories are empty and program is performing initialization is not representative on how it behaves. Observations of this stage are not objective and may be misleading. In order to avoid recording events at this stage, we added this new command line option that enables us to isolate this initial "warm up" stage.

**Simulation option: -snapshot-commits <snapshot>:**

This option takes a statistical snapshot and reset every <snapshot> number of instruction commits. The corresponding option in clock cycles was already implemented in PTLsim. However, since different configuration or enabling/disabling of components in the updated simulator design can affect execution, comparison of statistical snapshots that are separated on cycle-count may not be objective, while these may refer to different parts of the program. For this reason, this new option was added.

### *Updated statistics tree*

As already stated at section 3.1 Data and Instruction prefetcher, the component hierarchically belongs to the data cache subsystem. This applies to its statistical node as well. The statistics tree concerning both prefetcher can be found at /dcache/dataprefetch/. The recorded events along with a short explanation follow. Detailed reference on the described events can be found at section 3.1

```
|- /dcache/dataprefetch
    |- notifications          : Counts the number of prefetcher notifications[15].
    |    |- data              : Notifications concerning data prefetcher.
    |    |- instruction       : Notifications concerning instruction prefetcher.
    |
    |- in_data_queue_full     : Counts the number of a full input data queue.
    |- in_instr_queue_full    : Counts the number of a full input instruction queue.
    |- train                  : Number of notifications that reached the prefetcher (led to training).
    |- calculate              : Training events.
```

[15] Prefetcher notification is defined at section 3.1.2

```
|    |- instr                 : Instruction training events.
|    |    |- page_dropped     : Instruction notifications that led to a drop due to page crossing.
|    |    |- requests         : Notifications that successfully led to a prefetch request.
|    |
|    |- Data                  : Data training events concerning stride prefetching logic.
|         |- miss             : Prefetch Table misses.
|         |- hits             : Prefetch Table hits.
|              |- dropped_stride_size     : Notifications that dropped due to off-range stride.
|              |- dropped_page            : Notifications that dropped due to page crossing.
|              |- dropped_same_line       : Requests in the same block with notification and dropped
|              |- ignored_null_stride     : Notifications that dropped due to null stride calculation.
|              |- stride_replace          : Stride mismatch that lead to stride replacement.
|              |- conf_decrease           : Stride mismatch that lead to confidence decrement.
|              |- fwd_stride_hit          : Stride match that successfully lead to prefetch request.
|
|- drop_hit_on_pref_queue                 : Prefetch requests that were already in a prefetch queue.
|- data_queue_full                        : Number of full data prefetch queue events.
|- Instr_queue_full                       : Number of full instruction prefetch queue events.
|- requests              : Prefetch requests that were successfully placed to a prefetch queue.
|    |- requests_data    : Number of requests placed in the data prefetch queue.
|    |- requests_instr   : Number of requests placed in the instruction prefetch queue.
|
|- deliver               : Prefetch requests that were forwarded from prefetch queue to MAB for
                           allocation – Important notice: those may or may not allocate a MAB
                           entry, they can be dropped because are already allocated in the MAB or
                           are valid at L1 cache.
|
|    |- deliver_data_request      : Data prefetch requests .
|    |- deliver_instr_request     : Instruction prefetch requests.
|
|- dhit                  : Filled by the MAB, records where data prefetch requests hit.
|    |- L1               : L1 data cache and dropped
|    |- L2               : L2 cache
|    |- L3               : L3 cache
|    |- mem              : Main memory
|    |- missbuffer       : Already in MAB – dropped.
|
|- ihit                  : Filled by the MAB, records where instruction prefetch requests hit.
|    |- L1               : L1 instruction cache and dropped
|    |- L2               : L2 cache
|    |- L3               : L3 cache
|    |- mem              : Main memory
|    |- missbuffer       : Already in MAB – dropped
|
```

|- dsuccess : Number of data prefetch requests that were referenced by the core
| |- L1 : Were valid on L1 when requested by core (recorded as cache hit)
| |- missbuffer : Were referenced before reaching L1 (recorded as cache miss)
|
|- isuccess : Number of instruction prefetch requests that were referenced by the core
|- L1 : Were valid on L1 when requested by core (recorded as cache hit)
|- missbuffer : Were referenced before reaching L1 (recorded as cache miss)

The statistics are described in data-flow order. All events that are being recorded before MAB delivery are internal prefetcher events and until that point, they do not affect the memory subsystem. Prefetch requests that are forwarded to the Miss Address Buffer (deliver counter), leave the prefetcher component and are probed to cache memory in order to allocate a MAB entry or not. The total summary of prefetch requests that affect in some way the cache subsystem is counted at ihit & dhit counters. However, the number of prefetches that actually bring data to a higher memory level and are counted as prefetches, is the summary of L2, L3 and mem counters within ihit and dhit. L1 and missbuffer count requests that are already valid in the L1 or MAB and therefore, are dropped. Prefetcher accuracy can be calculated by the summary of successful prefetches (dsuccess or isuccess) divided by the number of prefetches, as dedcribed above.

Cache access control mechanism also belongs to the cache memory subsystem. The subtree for its statistics can be found at /dcache/cachelocks/. Arbiter events are recorded concerning which source gained cache access and which was blocked.

|- /dcache/cachelocks/
|- locked_by : Counts cache accesses given by arbiter.
| |- data_lock_by_missbuffer : Miss buffer L1 data cache accesses.
| |- data_lock_by_load : Load issue stage data cache accesses.
| |- data_lock_by_store : Sore commit stage data cache accesses.
| |- data_lock_by_prefetcher : Prefetcher data cache accesses.
| |- instr_lock_by_missbuffer : Miss buffer L1 instruction cache accesses.
| |- instr_lock_by_fetch : Fetch stage instruction cache accesses.
| |- instr_lock_by_prefetcher : Prefetcher instruction cache accesses.
|
|- found_locked : Counts cache access requests that were blocked by the arbiter.
|- data_lock_missbuffer : L1 data cache access denial for Miss Address Buffer.
|- data_lock_load : L1 data cache access denial for Load issue stage.
|- data_lock_store : L1 data cache access denial for Store commit stage.
|- data_lock_prefetcher : L1 data cache access denial for Prefetcher.
|- instr_lock_missbuffer : L1 instruction cache access denial for Miss Address Buffer.
|- instr_lock_fetch : L1 instruction cache access denial for Instruction fetch stage.
|- Instr_lock_prefetcher : L1 instruction cache access denial for Prefetcher.

Our design states that L1 instruction cache is a multi-port memory and therefore the mechanism is disabled. The “found_locked” events concerning instruction cache will be zero in this configuration. However, the mechanism can be switched on at any time and blocking events will be recorded normally.

Regarding the Miss Address Buffer, new counters for recording full-MAB events were added for each memory level.

```
|- /dcache/missbuffer/
    |- full_L1          : Counts L1 data cache full-MAB events.
    |- full_L1I         : Counts L1 instruction cache full-MAB events.
    |- full_L2          : Counts L2 cache full-MAB events.
    |- full_L3          : Counts L3 cache full-MAB events.
    |- full_pref: Counts events where maximum number of prefetches was allocated in MAB.
```

MAB structure is responsible for updating another series of statistics as well. The paths /dcache/<vcpu>/load and /dcache/<vcpu>/fetch hold statistics that show in which memory level did the demand requests hit. Those should agree with the number of loads that are recorded on Out-of-order issue load statistics. However, a load that was already allocated inside the MAB is not counted. In order to successfully count all cache accesses through this statistic subtree, we added a “missbuffer” counter to count the events where demand requests were already allocated. This happens quite often in our updated simulator model because of the prefetcher components, especially on good stride prefetching patterns.

Regarding the BPU, statistics were added for recording events for both Branch Target Buffers. The events are separated for 3 different branch categories that access BTBs (direct, indirect and conditional). The BPU main subtree is located at /ooocore/branchpred. Our expansions is descripted bellow:

```
|- /ooocore/branchpred/
    |- conditional              : Events regarding conditional branches (BTB 1).
    |    |-hit                  : BTB 1 hit for conditional branches.
    |    |    |- correct        : Correct target address.
    |    |    |-mispredict      : Mispredicted target address.
    |    |
    |    |- Miss                : BTB 1 miss for conditional branches.
    |
    |- indirect                 : Events regarding indirect branches (BTB 2)
    |    |- hit                 : BTB 2 hit (validation cannot be measured at this point)
    |    |- miss                : BTB 2 miss
    |
    |- direct                   : Events regarding direct branches (BTB 1)
        |- hit                  : BTB 1 hit for direct branches.
        |    |- correct         : Correct target address.
        |    |- mispredict      : Mispredicted target address.
        |
        |- Miss                 : BTB 1 miss for direct branches.
```

The subtree also includes its original event statistics. It should be noticed that validation for indirect branches is not possible at the BPU and mispredictions are recorded as Core events. An extra counter was added at /ooocore/total/fetch/stop/verification_stall in order to record BPU verification stalls of the instruction fetching stage.