Advancing the RISC-V Performance Simulation Ecosystem with Data Prefetching

Luís Crespo[1][0000−0002−1116−8859], Nuno Neves[2][0000−0003−0628−2259], Pedro Tomas[3][0000−0001−8083−4432], and Nuno Roma[4][0000−0003−2491−4977]

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{luis.miguel.crespo[1], nuno.neves[2], pedro.z.tomas[3], nuno.roma[4]}@tecnico.ulisboa.pt

Abstract. As processor speeds continue to outpace memory access times, memory latency remains a critical bottleneck in modern computing systems. Addressing this challenge requires effective memory access optimizations, such as data prefetching, which can anticipate memory requests and reduce stalls. Meanwhile, RISC-V has emerged as a compelling open-source alternative to proprietary ISAs with increasing adoption in domains ranging from embedded systems to high-performance computing. However, as RISC-V systems scale in complexity, there is a growing need for accurate and efficient performance modeling to guide architectural optimizations. Accordingly, this paper enhances the RISC-V Olympia trace-based performance simulator by extending it to support prefetching mechanisms. Additional contributions include modifications to the Spike simulator and the development of a complementary parser for generating an open-source simulation trace format. The effectiveness of these prefetching strategies is evaluated, demonstrating their potential to reduce memory access latency and enhance RISC-V system performance.

Keywords: RISC-V · Performance Modeling · Data Prefetching · Processor Simulation · High-Performance Computing

1 Introduction

The emergence of RISC-V as a royalty-free and versatile instruction set architecture (ISA) has significantly influenced processor design, offering an open-source alternative to proprietary ISAs like x86 and ARM [5,24,25,26]. With growing adoption in academia and industry, performance optimization is increasingly critical. To address this, the RISC-V Performance Modelling (and Simulation) Special Interest Group (SIG) was established to coordinate efforts toward a unified performance modelling and cycle-accurate simulation framework[1]. One key outcome is *Olympia*[2], a trace-driven simulator under development for evaluating

[1] https://lists.riscv.org/g/sig-perf-modeling
[2] https://github.com/riscv-software-src/riscv-perf-model

RISC-V architectural optimizations, including branch prediction, caching, and prefetching.

However, *Olympia*'s current trace generation relies on an emulator limited to the RV64GC ISA subset, excluding extensions relevant to performance modelling. A more suitable approach is leveraging *Spike*[3], the golden reference RISC-V simulator, given its broad adoption and role in validating new instructions and features. Additionally, *Olympia* lacks support for modelling data prefetching, a crucial technique for mitigating the "memory wall"[22]—the growing disparity between processor speed and memory latency. Prefetching improves memory efficiency, making its integration vital for modern performance models[21,14].

This paper extends *Olympia* with data prefetching mechanisms and introduces a complementary parser to generate traces from *Spike*. These enhancements strengthen RISC-V's performance modelling capabilities, advancing its competitiveness for high-performance computing. The contributions of this work include: (i) the integration of data prefetching mechanisms into *Olympia*; (ii) the development of a parser to convert *Spike*-generated traces into *Olympia*'s format; (iii) the implementation of multiple prefetching algorithms within *Olympia*; (iv) the evaluation of prefetching strategies based on performance impact; and (v) the identification of memory hierarchy bottlenecks and guidelines for future improvements.

2 Background and Related Work

Architectural simulation is essential for evaluating processor designs and optimizing performance. This section reviews key performance modelling approaches, introduces *Olympia*, a RISC-V out-of-order (OoO) CPU performance model, and discusses data prefetching as a way to mitigate memory access latency.

2.1 Performance Modeling

Processor simulation is the standard method for evaluating design choices, testing research ideas, and analyzing performance and power consumption [2,18]. Timing simulators, which capture microarchitectural behaviour to generate performance statistics such as Instructions Per Cycle (IPC), are commonly used to identify bottlenecks.

Timing simulators fall into three main categories [2]: (1) *Cycle-level simulators*[17,6] model every processor cycle with high accuracy but significant computational cost. (2) *Event-driven simulators*[9,10] improve efficiency by focusing on discrete events rather than every cycle. (3) *Interval simulators* [8] increase simulation speed by processing instruction flow in intervals based on key miss events (e.g., cache misses, branch mispredictions).

Among widely used simulators, *gem5*[6] supports RISC-V[16,20] and offers full-system functional simulation. Other RISC-V-compatible simulators include

[3] https://github.com/riscv-software-src/riscv-isa-sim

Spike, the official functional simulator, Sniper [12], and *Olympia*, developed within the RISC-V foundation. While *gem5* is well-suited for functional simulation, *Olympia* provides a more precise framework for performance modeling and design space exploration, making it particularly relevant for RISC-V.

2.2 Olympia

Olympia is a recent RISC-V OoO CPU performance model built on the Sparta Modeling Framework[4]. It functions as a trace-driven simulator, processing instruction streams in JavaScript Object Notation (JSON) or Simulation Trace Format (STF)[5]. While JSON is human-readable but limited in scope, STF is a more efficient binary format for full application traces. Currently, STF traces in *Olympia* are generated using an instrumented version of *Dromajo*[6], an emulator restricted to RV64GC, limiting flexibility.

Olympia models an OoO superscalar CPU with five pipeline stages: Fetch, Decode, Rename, Issue/Execute/Write Back, and Commit. It includes an L1 instruction and data cache and a shared L2 cache. Each component communicates through latency-defined ports, enabling modular design and ease of extension.

2.3 Data Prefetching

Prefetching in particular is a widely studied technique [21,14,3,13,4,23] that proactively anticipates and loads data into caches before it is requested by the processor, thereby reducing stall cycles and improving system throughput [7,14]. This helps reducing cache misses and improves the overall performance, since it minimizes the time the processor spends waiting for data, by overlapping the memory accesses with instruction execution. This latency-hiding effect makes prefetching an essential optimization mechanism for modern memory hierarchies.

Prefetching can be implemented in two primary ways: software prefetching [11,1] and hardware prefetching [3,13,4,23]. In software prefetching, the compiler (or a programmer) inserts explicit instructions to anticipate the load of data into the cache before it is accessed by the program. On the other hand, hardware prefetching is dynamically managed by a dedicated hardware component that predicts memory access patterns and issues prefetch requests based on run-time information. Because of its larger adoption, this work will be focused on hardware prefetching, although it could be later extended to support software prefetching.

Independently of the prefetcher type, prefetching can be applied at various cache levels, each with its own tradeoffs, resulting in different prefetching mechanisms [13]. Specifically, an L1 prefetcher makes use of information that would be costly to propagate to the L2-Cache, such as the Program Counter (PC) [3].

[4] https://github.com/sparcians/map
[5] https://github.com/sparcians/stf_spec
[6] https://github.com/chipsalliance/dromajo

In contrast, lower-level caches (e.g., L2 and L3) are characterized by a larger capacity and bandwidth, making them more tolerant to inaccurate and aggressive prefetching.

Several parameters and metrics are essential for a convenient characterization of prefetchers. Particular parameters include *prefetch distance* and *prefetch degree*, which control the *aggressiveness* of the prefetching strategy. *Prefetch distance* refers to the number of instructions or memory accesses anticipated ahead of the current instruction. Conversely, *prefetch degree* denotes the number of data blocks fetched into the cache. Although aggressive prefetching can help cover cache misses, it can also lead to reduced accuracy and inefficient memory bandwidth utilization.

The most relevant metrics related to prefetching are: (i) the *Issued prefetches*, i.e., the number of prefetch requests issued to the memory hierarchy after being filtered by the cache and the Miss Status Holding Registers (MSHR); (ii) the *Demand misses* which evaluates the number of cache accesses that result in misses not solved by the prefetcher (cache misses that are in the MSHR do not count as demand misses); (iii) *Useful prefetches*, corresponding to the number of issued prefetches that eliminate original misses and whose data (fetched by the prefetcher) is completed and placed in the cache before it is accessed; (iv) *Useless prefetches* indicating the number of prefetch requests whose fetched data (in the cache) is replaced before being used; (v) *Late prefetches* corresponding to the number of prefetch requests that are already in the cache or MSHR; (vi) *Accuracy* that measures how effectively a prefetcher predicts memory access patterns of applications and is calculated as the ratio of the number of useful prefetches to the total number of issued prefetches; and (vii) *Coverage*, indicating how prefetches can mitigate cache misses, and is calculated as the ratio of useful prefetches to the sum of useful prefetches and demand misses.

3 Proposed Simulation flow

This section details the integration of prefetching mechanisms into the *Olympia* RISC-V performance simulator, ensuring the completion of the simulation flow entirely within the RISC-V environment (as summarized in Fig. 1). Specifically, it describes *i)* the adopted methodology for trace generation using *Spike*; *ii)* the implementation of prefetching in the *Olympia* simulator as a module and its integration within the simulator memory hierarchy.

3.1 Trace Generation

Trace files serve as inputs for trace-driven simulators. These files contain prerecorded instruction streams from executed applications, capturing details such as PC, instruction opcodes, data memory addresses, and branch target addresses.

As mentioned above (see Section 2.2), the current trace generation method used by Olympia involves instrumenting the *Dromajo* functional model, which is not ideal, as it implements a specific configuration (RV64GC), leaving out

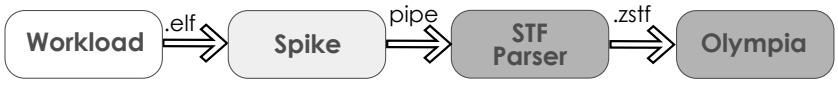

Fig. 1. Overview of the proposed simulation pipeline. A workload with the capture zone inserted in the code (using predefined MACROs) is compiled with a RISC-V-compatible compiler. It is then executed in *Spike* with the `--log-commits` flag. The output of *Spike* is parsed using the proposed STF parser through a pipe, capturing the intended code section and generating a trace binary file following the STF specification. The traces can then be fed into *Olympia* for performance modeling.

extensions that would benefit from performance modeling. Accordingly, a new STF parser (see Algorithm 1) was specifically developed that receives the execution log produced by *Spike* (`--log-commits`) and outputs the required STF trace file (`.zstf`) (see Fig. 1). Following a similar approach used by the *Dromajo* implementation, the parser only captures the delimited code section of the application under consideration through a START and END macro. The `--log-commits` flag must be used in *Spike* (instead of the default log (`-l`)) since the STF specification requires the memory access addresses that are only made available within this specific log. Furthermore, it was necessary to extend it to include the data size. The instruction parameters are also obtained from the commit log using pattern matching with *regular expressions*. The required parameters to produce the STF log include: *i)* PC, which is directly exposed; *ii)* next PC (in case of a branch) – a buffer of one instruction is maintained to keep track of non-sequential PCs; *iii)* instruction encoding, which is also directly exposed; *iv)* instruction type (load/store), obtainable with a simple analysis; *v)* data size, directly exposed after instrument spike; and *vi)* instruction width, obtainable with a simple analysis of the instruction binary.

3.2 Prefetcher Module

As mentioned earlier (Section 2.2), the simulator features a two-level cache. Prefetchers can be deployed at different (or multiple) cache levels [14]. Accordingly, the implemented data prefetcher operates as a single module (see Fig. 2.D) that is instantiated in two distinct units: L1D-Cache and L2-Cache (see Fig. 2.A). Each prefetcher monitors its corresponding cache requests and fill responses, generating prefetch requests for the subsequent lower-level cache. Before being issued, these requests pass through the associated cache to squash unnecessary prefetches – either because the requested cache line is already in the cache or allocated in the MSHR.

The internal prefetching engine (`Pf engine`) is the component that handles the demand and fill requests to generate the prefetch requests according to the prefetch algorithm under evaluation. Leveraging a prefetch engine allows the model to rapidly customize the prefetchers by selecting a given prefetch algorithm, being only necessary to implement each prefetcher algorithm once as an independent module. As a consequence, the latency of the engine in generating

Algorithm 1 STF Parser

```
Initialize STF writer
current_line = getline(pipe)
while next_line = getline(pipe) do                ▷ Go one inst. ahead to detect jumps
    if trace_regex = regex(current_line, PATTERN) then
        Extract PC, instruction (and width), and operation type from trace_regex
        if Instruction == END then                ▷ Trace until finding the END macro
            Exit loop
        end if
        if Setup then                             ▷ STF needs the starting PC for the header
            Write to STF the Setup Header with the starting PC
        end if
        if Tracing enabled then
            Extract next PC from next_line
            if next PC not expected PC then       ▷ Checked with the inst. width
                Write to STF the next PC target
            end if
            if Memory operation detected then
                Extract and log memory details in STF
            end if
            Write instruction record in STF       ▷ Depends on inst. width
        end if
        if Instruction is START then ▷ Ensure START is excluded from the trace
            Enable tracing
        end if
    end if
    current_line = next_line
end while
Close STF writer and exit
```

prefetch requests depends on the selected algorithm. The requests are stored in a prefetch queue (`Pf Queue`), and later sent to the cache. To prevent redundant requests, addresses already present in the queue are dropped.

In addition to the set of parameters specific to each prefetcher, they are configurable with *degree* and *distance* parameters to control the prefetch *aggressiveness*. Each prefetcher can also be set to generate prefetch requests triggered by store operations (turned off by default). Additionally, as a design decision, to maintain security while avoiding the extension of the Memory Management Unit (MMU), prefetch requests that would cross page boundaries are discarded.

By default, both the L1 and L2 prefetchers are disabled. Users can activate a prefetcher by modifying the architecture configuration file or passing it as an argument when running the simulator. In this work, three prefetch algorithms were implemented:

- Next-Line [19]: Assumes that accessing a memory location will likely lead to an access to the next one. It always fetches the next sequential memory block and can be applied at any cache level.
- Stride [3]: Predicts future data requests based on stride access patterns, like R[i], R[i+Q], R[i+2Q], R[i+3Q], where Q is the stride. It tracks these patterns for memory accesses triggered by the same PC and is best suited for implementation at the L1 data cache level.
- Best-Offset[13]: Dynamically selects the prefetch offset based on the application behavior. It has a learning phase, where it tests multiple offsets to

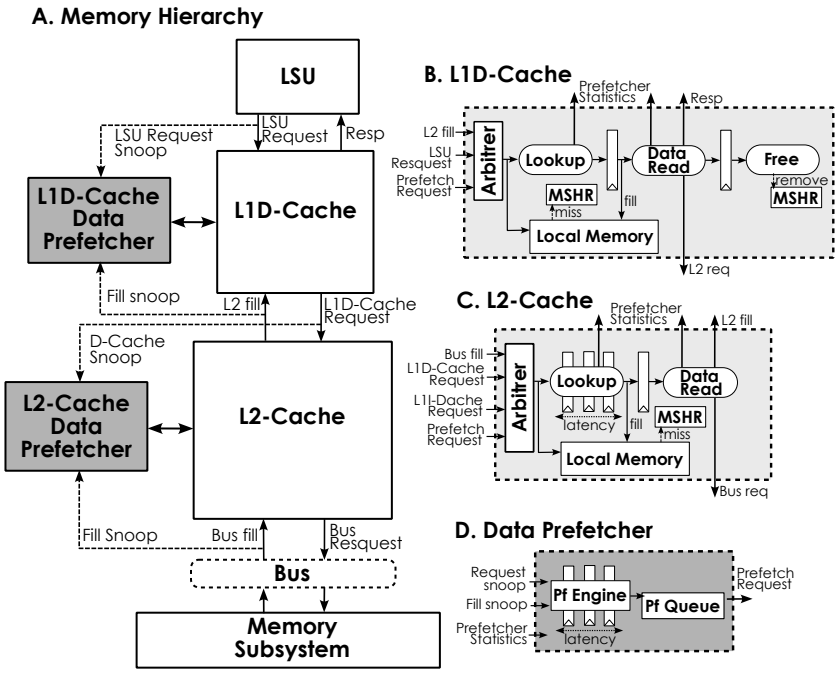

Fig. 2. Integration of prefetchers in the *Olympia* simulator: (A) High-level view of the memory hierarchy with data prefetchers; (B) D-Cache, (C) L2-Cache, and (C) Data Prefetcher architectures.

determine the most effective. By recording the base addresses of completed prefetch requests in a *recent requests table*, it infers the optimal timeliness for prefetch candidates. After the learning phase, it generates one request for each demand. This algorithm is usually more suited for L2 Caches and is the only implemented prefetcher that snoops both demand and refill requests.

Prefetch Statistics: The developed prefetcher module includes the gathering infrastructure to collect all the metrics described in Section 2.3, which are incremented as the corresponding events occur in the respective cache. In particular, an *issued prefetch* is counted when the cache issues it, a *demand access* is counted when a request is not found in the cache or the MSHR, and a *late prefetch* is recorded when a requested prefetch is found in the cache or MSHR. To identify *useful* and *useless prefetches*, a prefetch request is sent with the prefetch flag set, such that when the request arrives at the issuing cache, the corresponding cache line is marked as prefetched. If a subsequent memory request accesses that line, the cache notifies the prefetcher, which then increments the *useful prefetches* counter. The prefetch flag is subsequently disabled. Additionally, if a cache refill evicts a cache line with the prefetch flag set, the cache notifies the prefetcher to increment the *useless prefetches* counter.

3.3 Prefetcher Integration in the Olympia Simulator

Since *Olympia* is a trace-based simulator, it accepts as input the trace generated by a functional simulator. In the proposed solution, the *Spike* simulator is adopted (see Fig. 1). The following paragraphs outline the architecture of the *Olympia* L1D-Cache and L2-Cache, highlighting the required modifications for the prefetcher integration. Both caches transmit the statistical data to the respective prefetcher according to the events described above.

L1D-Cache: The L1D-Cache aggregated to the OoO processor model simulated by the *Olympia* simulator was implemented as a 3-stage pipelined unit, capable of accepting one request per cycle (see Fig. 2.B):

i) In the first stage (`Lookup`), the pipeline arbitrates between refill requests from L2 (`L2 fill`) and Load–Store Unit (LSU) requests (`LSU Request`), giving priority to refill requests. This stage also performs a memory lookup, allocating a new entry in the MSHR on a cache miss.

ii) In the second stage (`Data Read`), the L1D-Cache acts based on the lookup result: on a hit, it forwards data to the LSU; on a miss, it sends a request to the L2. If processing a refill request occurs, it updates the cache memory.

iii) The third stage (`Free`) deallocates the MSHR entries upon cache refills.

Required Modifications: The current implementation of the LSU keeps making requests to the L1D-Cache until it receives a hit. While it manages the arbitration among memory accesses within the LSU, it frequently repeats requests to the same memory access (that are still being serviced by the L2-Cache in case of a miss). This repetition floods the L1D-Cache with redundant memory requests. In a simulation system without prefetchers, this behavior is less problematic, as the L1D-Cache can simply drop the repeated LSU requests. However, since the L1D-Cache must prioritize memory requests over prefetch requests, the latter may not be serviced effectively. Accordingly, the L1D-Cache was adapted so that it notifies the prefetcher when it can accept prefetch requests. This is performed by the L1D-Cache at the data read stage (see Fig. 2.B). Then, prefetch requests are issued to the L2-Cache if its address is not in the cache or the MSHR.

Another limitation arises from the fact that, despite featuring a MSHR, the L1D-Cache could only support one outstanding request at a time, stalling until the L2-Cache responded. This effectively nullified the benefits of the prefetcher. To address this issue, the existing (but unused) credit-based communication system between the L1D-Cache and L2 was used to make the L1D-Cache non-blocking. In this scheme, the L2-Cache signals the L1D-Cache when it is ready to receive new requests, similar to a hardware *ready* signal. This mechanism allows the L1D-Cache to issue new requests as long as it has available credits, blocking further requests when credits are exhausted.

L2-Cache: The L2-Cache is implemented within the simulator with a configurable size pipeline unit, capable of accepting one request per cycle, either

from the Bus (`Bus fill`), the L1D-Cache (`L1D-Cache Request`), or the I-Cache (`I-Cache Request`) (see Fig. 2.C). Requests from the Bus have the highest priority, while L1D-Cache and I-Cache requests are arbitrated using a round-robin approach. During the first stage (`Lookup`), which spans several cycles (`latency`), the cache memory is accessed. In case of a cache refill, the respective cache line is updated. In the second stage (`Data Read`), the L2-Cache responds based on the lookup result: on a hit, it forwards data to the respective cache; on a miss, it sends a request to the Bus and allocates an entry on the MSHR. Misses already present in the MSHR are squashed, and the entry is deallocated upon a refill.

The integration of the prefetcher in the L2-Cache is slightly different from the L1D-Cache as it is not flooded with repeated requests. It sends an *acknowledge* signal to the prefetcher when it can receive prefetch requests. These requests are then arbitrated alongside the other requests, having the least priority. They follow the normal pipeline (see Fig. 2.C), being issued to the Bus if their addresses are not in the memory or the MSHR, being squashed otherwise. Prefetch requests generated by the L2-Cache prefetcher are not sent to the L1D-Cache when refilled.

4 Evaluation

The following paragraphs evaluate the proposed prefetching features introduced in the *Olympia* simulation ecosystem.

The experimental setup includes a selection of 8 applications from the MachSuite [15] benchmark suite, providing a diverse collection of computational kernels. These benchmarks were compiled using Clang with the following optimization and architecture-specific flags: `-O3`, and `-march=rv64imafdc`. The corresponding binaries were then executed in *Spike*, and the execution traces were generated using the proposed STF parser. The obtained traces were then used as inputs for the *Olympia* simulator, configured with its reference architectural parameterization for an out-of-order RISC-V processor (`big_arch`), publicly available in the *Olympia* repository. Table 1 summarizes the parameters of the modeled baseline architecture. Finally, the three implemented prefetcher models are validated by considering different prefetching setups, mixing the utilization of the next-line, stride, and best-offset prefetchers in the L1 and L2 caches.

4.1 L1 Data Prefetcher Validation

Fig. 3 presents the obtained metrics for the implemented next-line and stride prefetchers in the L1 data cache, across the considered benchmarks, including speedup (over the baseline), accuracy, and coverage of the prefetching algorithm. A perfect cache scenario is also included, representing the theoretical upper bound where every request results in a cache hit. To adjust the prefetcher *aggressiveness*, we select the prefetch *degree* to its best configuration for each benchmark (obtained experimentally), with values ranging from 1 to 8.

Table 1. CPU Architecture Specifications

CPU	Functional Units	Caches
RV64GC 1 OoO core 8-wide fetch/issue/commit 8 LSQ, 30 ROB	5 Int ALU (1 cycle) 2 Int Mult (3 cycles) 1 Int Div (23 cycles) 2 FP ALU (2 cycles) 2 FP MUL/DIV (4/63 cycles)	**L1-I:** - 16KB / 2-way - 8 MSHRs / 2-cycle latency **L1-D:** - 64KB / 2-way - 8 MSHRs / 2-cycle latency **L2:** - 512KB / 16-way - 16 MSHRs / 10-cycle latency

As could be expected, the performance impact for each model varies significantly across different workloads and prefetching methods. The highest speedups are observed in `bfs_bulk`, followed closely by `bfs_queue`, `spmv_ellpack`, and `spmv_crs`. These algorithms exhibit sparse memory access patterns, meaning they do not access data sequentially in the cache and therefore do not naturally benefit from its spatial locality. As a result, they experience the largest improvements with prefetching. Conversely, `gemm_ncubed` is an algorithm where both next-line and stride prefetchers should perform well. The accuracy results confirm this expectation. However, the low coverage values indicate that the *Olympia* memory hierarchy is overloaded with requests spanning multiple cache lines, limiting the overall benefits (discussed in Section 4.3). This is further verified with the coverage in another benchmark that saturates the cache (i.e., `gemm_ncubed`), having high accuracy but lower coverage. The `gemm_blocked` on the other hand, implements a version of `gemm` with tiling dimensioned to the cache configuration. As would be expected, a next-line prefetcher cannot accurately predict such a pattern, which is successfully identified by the stride prefetcher.

Overall, both prefetching algorithms exhibit comparable performance to that reported in the literature [19,3], validating the implemented prefetcher models.

4.2 L2 Data Prefetcher Validation

To validate the integration of the implemented prefetcher in the L2 cache, four different setups are considered: *i)* L2 next-line; *ii)* L2 best-offset; *iii)* L1 next-line + L2 best-offset; and *iv)* L1 stride + L2 best-offset. Similarly to the previous validation, a perfect cache scenario is also considered as a baseline reference. Fig. 4 presents the obtained evaluation metrics for the four setups.

As expected, contrarily to its deployment on the L1 data cache, it can be observed that coupling a next-line prefetcher in the L2 is insufficient to predict the memory access patterns given the lower amount of cache requests reaching the L2, hence showing inferior performance improvements. Nonetheless, it accurately predicts the patterns of the considered benchmarks. On the other hand, given the long training latency necessary to start up the best-offset prefetcher, its effects are only visible on benchmarks with large datasets (i.e., `gemm_blocked` and `gemm_ncubed`). In these cases, once the best-offset prefetcher has acquired

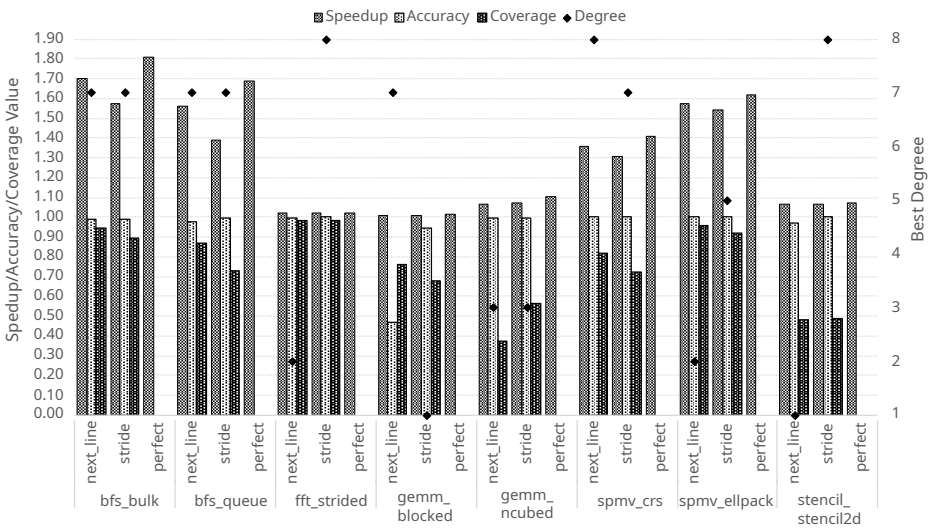

Fig. 3. Evaluation results for the next-line and stride prefetchers, deployed in the L1 data cache, for the considered benchmarks. Metrics include speedup, accuracy, and coverage. A perfect cache scenario is included as a theoretical upper bound. The prefetching degree is tuned per benchmark (within the range of 1–8).

sufficient history it issues more prefetch requests. To further validate this behavior, all benchmarks were executed in the gem5 [6] simulator (which also deploys an equivalent model for the best-offset prefetcher). As expected, the gem5 version also only generated prefetch requests for the `gemm` benchmarks (i.e., `gemm_blocked` and `gemm_ncubed`).

Overall, the considered setups for the L2 cache further validate the implemented prefetcher models and allow the identification of bottlenecks and limitations in the *Olympia* simulator, which are discussed in the next section.

4.3 Limitations and Future Development Guidelines

While the integration of data prefetching mechanisms in *Olympia* enhances its modeling capabilities, several limitations can still be found in the underlying processor and memory hierarchy models.

One key challenge is the limited flexibility of the memory hierarchy model in handling aggressive prefetching strategies. As observed in the experimental results, certain workloads expose bottlenecks in the cache and bus structures, as they cannot handle multiple outstanding requests, reducing the effectiveness of the prefetching strategy. In particular, the bus model that handles the communication of the L2 with the memory is blocking, causing the L2 to stop sending memory requests, eventually causing several pipeline bubbles in the processor.

Another significant bottleneck in *Olympia* is its load-store unit, which continuously generates memory requests (even for outstanding ones already sent

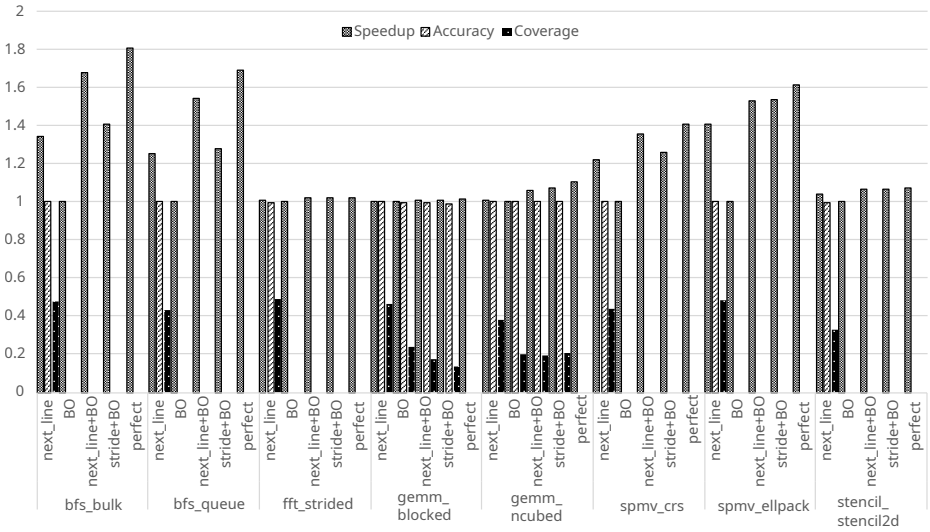

Fig. 4. Evaluation of prefetchers in the L2 cache.

to the memory hierarchy), contributing to increased congestion in the memory system. Also, since store operations must be processed immediately, they may delay or block demand loads and prefetches, leading to suboptimal performance. Addressing this issue requires implementing a store queue to decouple store operations from the LSU, allowing more efficient handling of memory requests and reducing pressure on the cache hierarchy.

5 Conclusion

This work extends the RISC-V *Olympia* simulator with data prefetching mechanisms and a new trace generation process using *Spike*, improving simulation flexibility and robustness. Experimental results confirm performance gains across workloads, validating the next-line, stride, and best-offset prefetchers. Future work will address the identified bottlenecks, explore software prefetching, refine page boundary handling, and integrate additional prefetching techniques to further enhance accuracy and efficiency.

Acknowledgments. Work supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC). We also acknowledge the contributions from projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), 2022.11626.BD, and from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland.

References

1. Ainsworth, S., Jones, T.M.: Software prefetching for indirect memory accesses. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 305–317. IEEE (2017)
2. Akram, A., Sawalha, L.: A survey of computer architecture simulation techniques and tools. Ieee Access **7**, 78120–78145 (2019)
3. Baer, J.L., Chen, T.F.: An effective on-chip preloading scheme to reduce data access penalty. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 176–186 (1991)
4. Bakhshalipour, M., Shakerinava, M., Lotfi-Kamran, P., Sarbazi-Azad, H.: Bingo spatial data prefetcher. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 399–411. IEEE (2019)
5. Balkind, J., McKeown, M., Fu, Y., Nguyen, T., Zhou, Y., Lavrov, A., Shahrad, M., Fuchs, A., Payne, S., Liang, X., et al.: Openpiton: An open source manycore research framework. ACM SIGPLAN Notices **51**(4), 217–232 (2016)
6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH computer architecture news **39**(2), 1–7 (2011)
7. Falsafi, B., Wenisch, T.F.: A primer on hardware prefetching. Springer Nature (2022)
8. Genbrugge, D., Eyerman, S., Eeckhout, L.: Interval simulation: Raising the level of abstraction in architectural simulation. In: HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture. pp. 1–12. IEEE (2010)
9. Hardavellas, N., Somogyi, S., Wenisch, T.F., Wunderlich, R.E., Chen, S., Kim, J., Falsafi, B., Hoe, J.C., Nowatzyk, A.G.: Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. ACM SIGMETRICS Performance Evaluation Review **31**(4), 31–34 (2004)
10. Hughes, C.J., Pai, V.S., Ranganathan, P., Adve, S.V.: Rsim: simulating shared-memory multiprocessors with ilp processors. Computer **35**(2), 40–49 (2002)
11. Khan, M., Sandberg, A., Hagersten, E.: A case for resource efficient prefetching in multicores. In: 2014 43rd International Conference on Parallel Processing. pp. 101–110. IEEE (2014)
12. Mallya, N.B., Gonzalez-Alvarez, C., Carlson, T.E.: Flexible timing simulation of risc-v processors with sniper. Simulation **4**(1) (2018)
13. Michaud, P.: Best-offset hardware prefetching. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 469–480. IEEE (2016)
14. Mittal, S.: A survey of recent prefetching techniques for processor caches. ACM Computing Surveys (CSUR) **49**(2), 1–35 (2016)
15. Reagen, B., Adolf, R., Shao, Y.S., Wei, G.Y., Brooks, D.: Machsuite: Benchmarks for accelerator design and customized architectures. In: 2014 IEEE International Symposium on Workload Characterization (IISWC). pp. 110–119. IEEE (2014)
16. Roelke, A., Stan, M.R.: Risc5: Implementing the risc-v isa in gem5. In: First Workshop on Computer Architecture Research with RISC-V (CARRV) (2017)
17. Sharkey, J., Ponomarev, D., Ghose, K.: M-sim: a flexible, multithreaded architectural simulation environment. Techenical report, Department of Computer Science, State University of New York at Binghamton (2005)

18. Skadron, K., Martonosi, M., August, D.I., Hill, M.D., Lilja, D.J., Pai, V.S.: Challenges in computer architecture evaluation. Computer **36**(8), 30–36 (2003)
19. Smith, A.J.: Sequential program prefetching in memory hierarchies. Computer **11**(12), 7–21 (1978)
20. Ta, T., Cheng, L., Batten, C.: Simulating multi-core risc-v systems in gem5. In: Workshop on Computer Architecture Research with RISC-V (2018)
21. Vanderwiel, S.P., Lilja, D.J.: Data prefetch mechanisms. ACM Computing Surveys (CSUR) **32**(2), 174–199 (2000)
22. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. ACM SIGARCH computer architecture news **23**(1), 20–24 (1995)
23. Yu, X., Hughes, C.J., Satish, N., Devadas, S.: IMP: Indirect memory prefetcher. In: Proceedings of the 48th International Symposium on Microarchitecture. pp. 178–190 (2015)
24. Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(11), 2629–2640 (2019)
25. Zaruba, F., Schuiki, F., Benini, L.: Manticore: A 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing. IEEE Micro **41**(2), 36–42 (2020)
26. Zhao, J., Korpan, B., Gonzalez, A., Asanovic, K.: SonicBOOM: The 3rd generation berkeley out-of-order machine. In: Fourth Workshop on Computer Architecture Research with RISC-V. vol. 5, pp. 1–7 (2020)