Pref-X: a framework to reveal data prefetching in commercial in-order cores

Quentin Huppert, Francky Catthoor, Lionel Torres, David Novo

Pref-X: A Framework to Reveal Data Prefetching in Commercial In-Order Cores

Quentin Huppert[1], Francky Catthoor[2], Lionel Torres[1], and David Novo[1]
[1]*LIRMM, University of Montpellier, CNRS, Montpellier, France* [2]*imec, Leuven, Belgium*
{quentin.huppert, david.novo}@lirmm.fr

***Abstract*—Computer system simulators are major tools used by architecture researchers to develop and evaluate new ideas. Clearly, such evaluations are more conclusive when compared against commercial state-of-the-art architectures. However, the behavior of key components in existing processors is often not disclosed, complicating the construction of faithful reference models. The data prefetching engine is one of such obscured components which can have a significant impact in key metrics like performance and energy. In this paper, we propose Pref-X [1], a framework to analyze functional characteristics of data prefetching in commercial in-order cores. Our framework reveals data prefetches by *X-raying* into the cache memory at per-cycle granularity, which allows linking memory access patterns with changes in the cache content. To demonstrate the power and accuracy of our methodology, we use Pref-X to replicate the data prefetching mechanisms of two representative processors, namely the Arm Cortex-A7 and the Arm Cortex-A53, with a 99.8% and 96.9% average accuracy, respectively.**

***Index Terms*—Computer Architecture Simulation, Data Prefetching, Commercial Processors**

I. Introduction

Computer architectures are in permanent evolution, always pushing for higher performance and energy efficiency, while adapting to new technologies and applications. Cycle-accurate computer architecture simulators are major catalysts of this process, enabling a quick evaluation of new ideas while avoiding the expensive manufacturing process of integrated circuits. However, commercial microarchitectures include unspecified components that compromise the accuracy of their simulation models. This lack of information complicates the fair comparison of new simulated architectures and commercial ones.

Previous works have evaluated the accuracy of cycle-accurate simulators modeling commercial architectures [2]–[5]. They instantiate a parametric simulation model with values directly copied from the datasheet (e.g., cache sizes) or extracted from microbenchmarks executed on the architecture (e.g., cache latencies). However, currently they all either disable or ignore data prefetching due to the lack of information on the underlying mechanisms. But prefetching has a non-negligible impact in performance and energy consumption, and its omission can seriously compromise the accuracy of simulations. For instance, recent works show that data prefetching increases performance by more than 25% in memory intensive workloads [6]–[8].

Our main goal is to provide a general methodology and supporting framework to analyze functional characteristics of data prefetching in commercial in-order cores. To this end, we develop Pref-X, a two-phase framework that enables functional analysis by inspecting the content of the L1 cache at the request granularity. We use Pref-X to replicate the data prefetching mechanisms of two representative processors, namely the Arm Cortex-A7 [9] and the Arm Cortex-A53 [10], with a 99.8% and 96.9% average accuracy, respectively.

II. Background

The memory system of in-order cores in typical mobile SoCs includes multiple levels of on-chip cache memory and off-chip DRAM memory [11]. When executing a memory instruction, the CPU generates a *Virtual Address (VA)* (❶ in Figure 1) which is translated to a *Physical Address (PA)* by the *Memory Management Unit (MMU)* at a page granularity (typically 4kB). The request is then sent to the L1 cache ❷, which manages stored data at a cache line granularity (typically 64B). Thus, a page contains 64 cache lines. When the requested data is already in L1 cache (i.e., a hit), the data is immediately sent to the CPU ❼. However, when the data is not in L1 (i.e., a miss), the request has to travel deeper in the memory hierarchy (higher levels of cache and ultimately DRAM) ❸ until finding the data location. Then, the data is sent to the L1 ❺ and the CPU. Note that the limited size of the L1 can cause the eviction of unrelated data ❻ to make place for the newly requested data.

Modern CPUs use hardware data prefetching to hide the cache miss latency, which is several hundred cycles for off-chip DRAM accesses [5]. Prefetching is a well-studied speculation mechanism that predicts the addresses of future memory requests and fetches the corresponding data before the CPU requests them ❹. Among the different types of prefetching techniques, stride prefetching is very popular in mobile CPUs because of its combination of good performance and moderate hardware complexity [12]–[14]. The stride prefetcher detects *streams* from requests sequences with a constant stride in the access pattern and prefetches cache lines following that stride. For instance, the figure shows a stream with a stride of two triggering a 2-line prefetch: 0, 2, 4 → 6, 8. Hence, we will focus on this type of prefetcher in this paper.

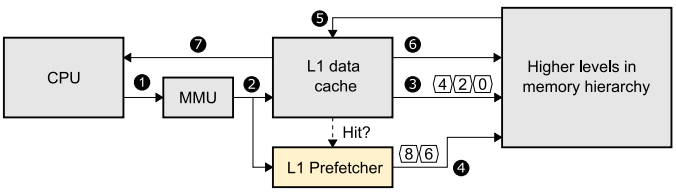

Fig. 1. Memory system organization of a typical mobile CPU.

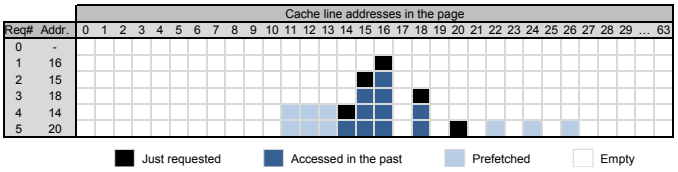

Fig. 2. Example of using our L1 cache inspection methodology on the address pattern $\{16, 15, 18, 14, 20\}$. Black and dark blue lines indicate that the data is present in the L1 cache after having been accessed. Instead, light blue lines indicate that the data is present despite not having been accessed (i.e., prefetched data). We can identify two 3-line prefetches: at request 4 (i.e., $\{11, 12, 13\}$) and at request 5 (i.e., $\{22, 24, 26\}$).

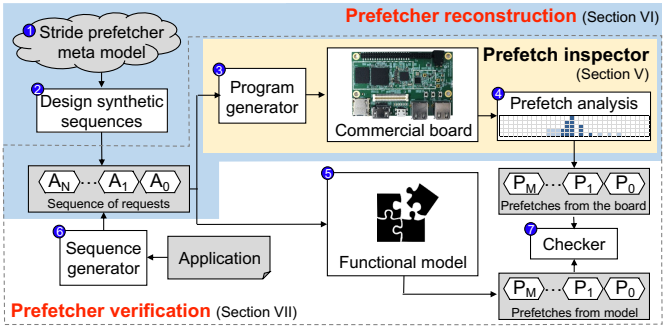

Fig. 3. Pref-X framework.

III. Main Challenges and Contributions

Studying the prefetcher functionality in commercial cores is a challenging task. Ideally, we would generate a sequence of memory requests (❶ in Figure 1) and just *observe* the extra requests generated by the prefetcher ❹. However, a *first challenge* comes from the fact that the output of the prefetcher is not visible in commercial architectures. Accordingly, we propose a method to deduce the prefetcher activity from the changes in the content of the L1 cache. For instance, Figure 2 shows how the content of the cache changes with every new request. Based on this information, we can infer that the fourth request has triggered a prefetch burst of three cache lines: 16, 15, 18, 14 → 13, 12, 11, and the fifth request has triggered a second prefetch burst: 16, 15, 18, 14, 20 → 22, 24, 26.

A *second challenge* is controlling the prefetcher behaviour exclusively from the input request sequence while isolating it from other concurrent events in the microarchitecture. For example, the response of the prefetcher can be affected by the state of previous pendent memory requests. Thus, it is important to infer a controlled microarchitectural state before and after receiving each memory request. Accordingly, we propose a method to construct the input sequence of memory requests that enables the request-by-request analysis shown in Figure 2.

Finally, a *third challenge* is how to build a functional model of the prefetcher from the control and observation capabilities gained after solving the previous challenges. We propose a method to construct that model with a limited number of experiments for stride prefetchers. In this paper, this is demonstrated for the L1 data prefetcher of in-order cores.

IV. The Pref-X mechanism

Figure 3 illustrates the basic Pref-X mechanism, which includes two distinct phases: (1) the reconstruction of the prefetching algorithm from synthetic access patterns, and (2) the verification of the reconstructed algorithm with access patterns from real applications.

Prefetcher reconstruction. To reconstruct the prefetching algorithm, we first define a meta model (❶ in Figure 3) that captures the common features of stride prefetchers. We circumscribe the model as a block that reacts to CPU memory requests by emitting prefetching requests. Any prefetching request is triggered by a single memory request and exclusively depends on three variables: (1) the address of the memory request, (2) whether that requests hits or misses L1, and (3) the internal state of the prefetcher. It is relatively straightforward to realize that only those variables make sense to exploit. Note that a prefetcher needs to store metadata for internal bookkeeping (e.g., managing active streams), which becomes a variable we need to control in our experiments to produce a repeatable prefetching behaviour. The prefetcher constantly monitors CPU memory requests to detect the beginning and the end of streams. An active stream is bounded by these two detection processes. A prefetcher can simultaneously host a limited number of streams. Finally, the output of a stride prefetcher can be characterized by a base address, a stride, and a burst length [12]–[14].

Based on the described meta-model, we craft synthetic sequences of addresses ❷ to expose the main parameters regulating the key prefetching mechanisms. For instance, the minimum number of accesses in a stream needed to trigger a prefetch, the maximum stride length, etc. Each sequence is embedded in a program ❸ designed to iteratively execute the sequence of memory requests in a way that allows the detection of prefetches ❹. We call *prefetch inspector* to the combo of program generation from a sequence of addresses, execution of the program in the board, and postexecution analysis for prefetch detection (refer to Section V for details).

We iterate the described flow to build a functional model of the prefetcher ❺, whose objective is to generate the same prefetches as the real board for any input sequence of addresses. Section VI provides further details on this phase.

Prefetcher verification. The second phase of Pref-X targets the functional verification of the model built in the previous phase. This time, we extract sequences of addresses from real applications ❻ that we feed to both the real board using the prefetch inspector and to our functional model. Then, we compare the prefetches from the board and the model ❼. If we detect relevant differences, we use the new insights to polish the functional model. The more complete the meta-model of the prefetcher, the less number of iterations will be needed between the two phases. Section VII provides further details on this phase.

V. Prefetch inspector

The prefetch inspector takes an input sequence of addresses and reconstructs a request-by-request representation of the content of the L1 cache (see Figure 2). From that representa-

tion, we can directly identify each prefetched cache line and the input request triggering the prefetch. Figure 4 illustrates the different steps involved in the process. The incremental build-up step is essential to achieve the request-by-request analysis. It takes the input sequence of X addresses and generates $X+1$ subsequences starting from the empty subsequence and incorporates one address at a time. Each subsequence goes separately through the succeeding steps.

For each subsequence, we build a program that executes in the target core. An input parameter indicates the number of pages to inspect during execution (e.g., one page, which corresponds to 64 cache lines, in Figure 4). The objective is to measure which cache lines reside in the L1 after the execution of the subsequence. For each `k` line in the inspected pages, the program (1) initializes a set of *Hardware Performance Counters (HPC)* to count L1 hits, (2) iterates `N` times over a four-stage process, and (3) stores the HPC value in a file for later inspection. We conclude that the `k` cache line has been prefetched if it resides in L1 despite not belonging to the executed subsequence. We use $\texttt{N} = 1000$ in our experiments as it shows a good signal-to-noise ratio in measurements. Note that we run the program in commercial architectures including an operating system, which produces residual system activity that we attenuate by increasing the value of `N`. That is crucial to expose the actual prefetching impact on the relevant application code.

The four-stage process includes a first stage where we execute the subsequence of memory requests ❶. Typically, we insert a large number of NOPs in between each memory request to give enough time for the memory system to serve all previous memory request before executing the new one. Note that serving a memory request from DRAM can take several hundred cycles. Thus, having the possibility to avoid transient states is very important to isolate the prefetching functionality from the rest of the microarchitecture and to facilitate its analysis. After executing the subsequence, we wait for the L1 to receive all pending requests ❷. While for some analysis one could play with the number of NOPs in the first stage (e.g., detect time out mechanisms), here we always have to wait until the L1 stabilizes before *reading* its content: an L1 hit of the request to the k cache line ❸ will reveal a prefetch when k does not belong to the subsequence. Finally, the reset stage ❹ avoids that the L1 cache and prefetch state are carried out across iterations. We achieve that by keeping the relative subsequence but changing the working page at every iteration. Thereby, the prefetcher cannot accumulate state information across iterations and every iteration starts with a cold L1 cache.

After the execution of all the subsequences, the prefetch analysis step processes the file containing a number of L1 hits for each cache line in each subsequence and it produces the graphical representation shown in Figure 2.

VI. Prefetcher Reconstruction

In this section, we want the illustrate the general methodology on a specific representative use case. For this purpose, we use the Arm Cortex-A53 core to showcase how to build a functional model of a data prefetch engine. We suggest different synthetic sequences that help reveal the key underlying prefetching mechanisms.

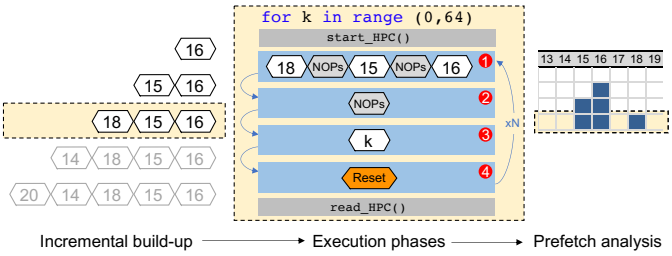

Fig. 4. Prefetch inspector process on one page.

Triggering prefetches. We use a sequential sequence as an example of an ideal stream that should be detected by the simplest stride prefetcher. As expected, sequence #1 in Figure 5 shows a high prefetching activity. From this experiment, we conclude the following:

- Three accesses of a stream trigger a prefetch burst.
- A prefetch burst includes three cache lines.
- A hit on a prefetched line triggers a new prefetch burst.

Missing after a prefetched burst. With the previous sequence we have discovered that hitting a prefetched line triggers further prefetching. Now, we want to understand the effect of missing the line that follows the stream after the prefetched burst. To do that, we repeat the previous sequence until the prefetch is triggered and then we provoke a miss instead of the hit on the prefetched burst. Sequence #2 in Figure 5 shows that the miss in the fourth request triggers a 1-line prefetch. We confirm this behaviour by continuing the sequence with a fifth request that misses after the newly prefetched line and generates another 1-line prefetch.

Maximum distance between requests in a stream. Practical prefetching engines have a limited detection scope. In this experiment, we want to discover the number of unrelated requests that can be accommodated between stream requests while still triggering a prefetch. Sequence #3 in Figure 5 shows that at least six unrelated misses can be included between the first and eight request and between the eight and the fifteenth request while still triggering a prefetch. However, no prefetch is triggered when repeating the same experiment with seven unrelated misses between requests. Therefore, the maximum distance between request in a stream is seven.

Burst hitting in L1. So far, we have seen that the prefetched burst corresponds to the three next lines following the stream sequence. However, what would happen if any of these lines were already present in L1? Sequence #4 in Figure 5 starts with a request to cache line 10 and continues with sequence #1. As usual, the sixth request triggers a three-line prefetch burst after hitting in a prefetched line 4. However, the burst skips line 10, which is already in L1, and continues with the stream sequence. Thus, the prefetch burst includes the three next lines following the stream sequence that are not in L1.

Prefetching across the page boundary. Often, stride prefetchers are bounded to act within the page boundary

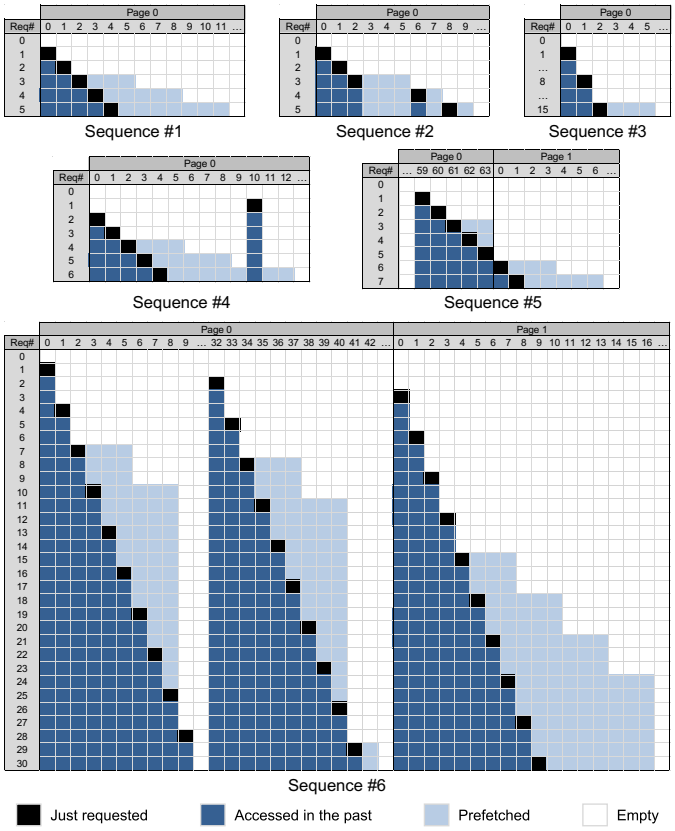

Fig. 5. Synthetic sequences used to reconstruct the Cortex-A53 prefetching algorithm.

as streams in the virtual address space are not necessarily maintained after physical address translation. Accordingly, sequence #5 in Figure 5 includes a stream of requests that expands over two pages crossing the page boundary. We can see that the hits in the prefetched lines do not generate prefetches beyond the page boundary. However, the first request on the second page that follows the stream immediately results in a prefetch. This indicates that the prefetcher maintains the stream metadata across pages but does not resume prefetching until a request that follows the stream misses in L1 and validates the new page.

Multi-stream prefetching. So far, we have seen sequences that include a single stream. Sequence #6 in Figure 5 challenges the prefetcher with three perfectly interleaved streams. We observe that requests 7 and 8 trigger two consecutive prefetches of different streams, which indicates that the triggering process has run in parallel. Then, new hits on the prefetched lines generate new prefetches for the two streams until request 13. Request 13 (14) does not generate prefetches because the prefetcher has forgotten that line 4 (36) was once prefetched (signaling the limit of the internal request buffer). In the meanwhile, stream 3 (in page 1) has accumulated the last three L1 misses, which triggers a prefetch at request 15. The following requests keep on generating stream 3 prefetches until request 27, indicating that the prefetcher has forgotten that line 8 of page 1 was once prefetched. After that, request 28 in stream 1 misses L1, which should generate a 1-line prefetch. However, the lack of the prefetching action indicates that stream 1 has been forgotten. Instead, request 29 in stream 2 generates the expected 1-line prefetch. Thus, we can conclude that the prefetcher can only track up to 2 streams.

TABLE I
COMPARISON OF THE A7 AND THE A53 STRIDE PREFETCHER VARIANTS.

Parameter	A7	A53
Initial trigger cond.	3 misses	3 misses
Trigger input	L1 misses	L1 misses + hit on prefetch
Burst length	3	3 / 1
Max. stride length	4	4
Max. dist. in requests	1	7
Cond. to continue	Miss after prefetched	Hit on prefetch: burst of 3 / miss after prefetch: burst of 1
Burst hitting in L1	Stop burst	Keep burst skipping lines in L1
Tracking across pages	No	Yes (see seq. #4 in Fig. 5)
Max. num. of streams	1	2
Max. inter stream dist.	-	8: from 3rd miss to any prefetch

Due to the lack of space, we cannot describe all the synthetic sequences used during reconstruction[1]. However, Table I summarizes the main characteristics of the two stride prefetches evaluated in the experimental section.

VII. PREFETCHER VERIFICATION

After building a functional model of the target prefetcher from the information extracted from the synthetic sequences, we propose to challenge the model with realistic sequences derived from real applications. However, the prefetch inspector process, described in Section V, suffers from key limitations that need to be considered in the generation of sequences from real applications.

The prefetch inspector mechanism relies on the assumption that the prefetched data remains in L1 cache after the sequence execution. Thus, the sequence should be executed while avoiding L1 evictions. This requires a careful examination of the cache architecture and replacement policy. For instance, both architectures considered in the experimental evaluation include a 32kB 4-way set-associative L1 data cache which uses a pseudo-random cache replacement policy. From this information, we can deduce that the 32kB are divided in 4 ways of 8kB and each way is divided in 128 sets (8kB divided by 64B cache line). Due to the random replacement policy, the only way to avoid evictions is to distribute the sequence memory requests across the 128 sets without repetitions. Figure 6 shows how the address bits are segmented to address the L1 cache: the 6 *Least-Significant Bits (LSBs)* to address the byte within the 64B cache line, the next 7 bits to index the cache set and the remaining bits compose the tag. At the same time, we need to consider the physical address translation process, which maintains the 12 LSBs (i.e., page offset) while modifying the remaining bits (i.e., page number). To control this process, we configure the memory allocator of the operating system to maintain page adjacency after address translation. Thereby, we can guarantee that two consecutive pages will have a different

[1]All synthetic sequences and prefetcher models will be made public upon paper acceptance.

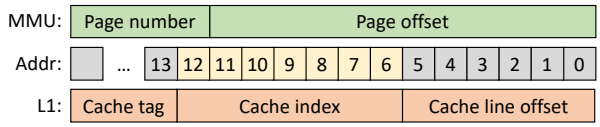

Fig. 6. Bit segmentation of an address in a 32kB 4-way set-associative cache.

LSB in the physical page number field. Figure 6 shows that only the *Most Significant Bit (MSB)* of the cache index is included in the page number address field, however, the two MSBs would be included in a 64kB L1 cache. This implies that the prefetch inspector can only analyze up to two pages at the time to avoid L1 evictions in architectures targeted in the experimental section.

Furthermore, the prefetch inspector processing time grows $\mathcal{O}(S^2P)$, where S is the sequence length and P the number of pages to inspect. Thus, even if the sequence does not produce L1 evictions, we would still need to enforce an S_{max} to keep processing time within limits[2]. Accordingly, the sequence generator preprocesses the memory sequences extracted from the applications to be within the discussed limitations. Specifically, the original sequences are segmented in chunks of S_{max} elements. Then for each chunk, we enable a configurable selection of the two pages to inspect. In practice, we rank the pages accessed in the chunk based on the number of accesses and we observe that the top-4 pages already account for most of the prefetches in the chunk.

VIII. Experimental Results

In this section, we present results on two representative Arm in-order CPUs: the 32-bit Cortex-A7 [9] and its successor the popular 64-bit Cortex-A53 [10].

A. Experimental methodology

To evaluate the accuracy of the functional models built using Pref-X, we compare the number of prefetches generated by the same sequences of memory requests on the commercial board and on the functional model. We use 24 benchmarks from the SPEC CPU 2006 suite as realistic applications and the gem5 simulator [15] to instrument the execution and extract the memory request sequences. To reduce the simulation time, we follow a standard SimPoint methodology [16] with one million instructions per slice and max K set to 30. Each simpoint is simulated to generate a memory trace of L1 d-cache requests that we further split in sequences of 1000 requests. In each sequence, we remove repeated requests to the same cache line as only the L1 misses are important for prefetching. To reduce the memory footprint, we compact the page mapping to avoid unused intermediate pages. We also discard the sequences with more than 100 pages as such a disperse access pattern will generate very low prefetching activity in the considered architectures. Overall, we evaluate 149 thousand sequences that include over 16 million requests.

As we are only evaluating the number of prefetches, we execute a simpler program than the prefetcher inspector described in Section V, which is designed to identify all prefetched lines. Instead, we directly use the HPC to count the number of prefetches generated by 1000 executions of the full sequence. Then, we feed the same sequences to our prefetch functional models, which are implemented in Python as a state machine that reacts to incoming memory requests.

[2]We use $S_{max} = 100$ in our experiments.

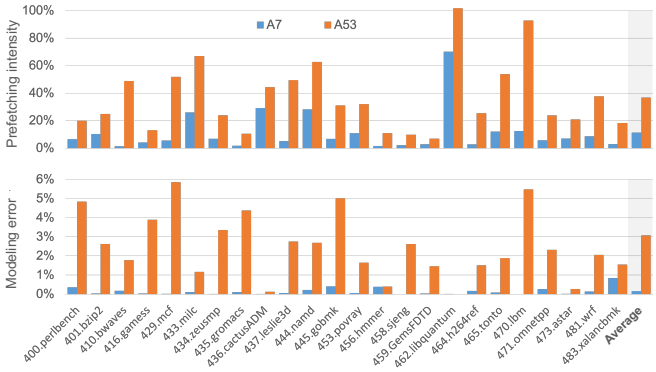

Fig. 7. Prefetching error in 24 programs of the SPEC CPU 2006 suite.

B. Results

Figure 7 shows the prefetching intensity and the modeling error of each evaluated program. The prefetching intensity corresponds to the ratio between prefetches and requests in the input sequence. The modeling error is the absolute difference in prefetches between the real execution and the model, normalized to the real execution. We make three observations. First, the A53 achieves a close to 40% average prefetching intensity, which is 3× higher than in the prior generation A7. This indicates that the requests generated by the prefetcher can represent a sizable share of the data movement in the memory system, which motivates the need for accurate prefetcher models. Second, the average (maximum) error of the A7 and the A53 prefetcher models are 0.2% (0.8%) and 3.1% (5.9%), respectively. Although the higher complexity of the A53 prefetching mechanisms results in a higher modeling error, this is well below the error tolerance of the architecture simulations motivating this work [5]. Third, there is no apparent correlation between the modelling error and the prefetching intensity.

To gain further insights in our last observation, Figure 8 shows the average error and the distribution of the analyzed sequences grouped by the number of prefetches. Each sequence is assigned to a bin, which is defined by a minimum and a maximum number of prefetches. We make two new observations. First, most sequences trigger less than 20 (10) prefetches in the A53 (A7) prefetcher and accumulate a slightly larger error than the other bins. This is due to the higher influence of single mispredictions in corner cases (e.g., end of the sequence). Second, the modeling error remains largely constant across all bins, which shows that our functional models maintain the accuracy in sequences with low, medium, and extremely high prefetching activity.

In summary, our results show that Pref-X takes an important step forward in enabling accurate models of data prefetching for the simulation of commercial in-order architectures.

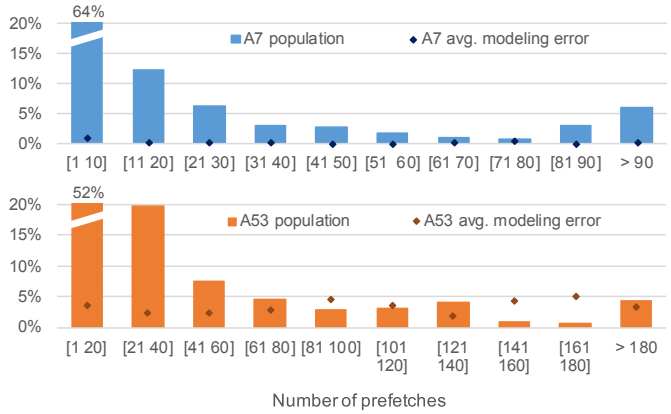

Fig. 8. Prefetching error and distribution of memory sequences grouped by number of prefetches.

IX. Related work

To our knowledge, this is the first work to propose a general methodology and framework to analyze functional characteristics of data prefetching in existing in-order processor architectures. In this section, we identify two areas of related work that are particularly relevant to this work.

On the one hand, previous works have evaluated the accuracy of cycle-accurate simulators modeling commercial architectures. However, they either disable the prefetcher on the reference board and on the simulator during calibration [2], avoid prefetches by generating random accesses [5], or simply ignore the effect of prefetching [3], [4]. Our work is complementary to these works as it provides a systematic way to include data prefetching in the calibration of simulators.

On the other hand, several prior works study prefetching to find security vulnerabilities, such as covert channels [17] or side-channels timing attacks [18], [19]. For instance, Rohan et al. [17] examine the *streamer* prefetcher, which is an L2 hardware prefetcher present in commercially available Intel machines, to construct a cross-thread covert channel. They all use basic microbenchmarks to discover some hidden properties (e.g., if the prefetcher is shared between SMT threads) necessary to expose and exploit the targeted vulnerability. However, their ad hoc approach cannot be leveraged to reveal the complete set of properties needed to faithfully simulate the functionality of the stride prefetchers targeted in this work. Instead, we propose a more complete and systematic approach.

X. Conclusion

We introduce Pref-X, a new framework to analyze functional characteristics of data prefetching in commercial in-order cores by inspecting the content of the L1 cache at the request granularity. We demonstrate how to use synthetic sequences of memory requests to reconstruct data prefetching algorithms. We use Pref-X to replicate the data prefetching mechanisms of two representative processors, namely the Arm Cortex-A7 and the Arm Cortex-A53 cores with a 99.8% and 96.8% average accuracy, respectively. We believe that Pref-X takes an important step forward to enable accurate simulation models of commercial architectures.

References

[1] "Pref-X GitLab repository," https://gite.lirmm.fr/adac/pref-x.

[2] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *Proceedings of ISPASS*, 2014.

[3] A. Butko, R. Garibotti, L. Ost, and S. Gilles, "Accuracy evaluation of gem5 simulator system," in *Proceedings of ReCoSoC*, 2012.

[4] A. Akram and L. Sawalha, "Validation of the gem5 simulator for x86 architectures," in *Proceedings of PMBS*, 2019.

[5] Q. Huppert, T. Evenblij, M. Perumkunnil, F. Catthoor, L. Torres, and D. Novo, "Memory hierarchy calibration based on real hardware in-order cores for accurate simulation," in *Proceedings of DATE*, 2021.

[6] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of ISCA*, 2019.

[7] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *Proceedings of MICRO*, 2021.

[8] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of ASPLOS*, 2020.

[9] "Arm Cortex-A7 MPCore processor techn ref. manual," https://developer.arm.com/documentation/ddi0464/latest, [Apr-22].

[10] "Arm Cortex-A53 MPCore processor techn ref. manual," https://developer.arm.com/documentation/ddi0500/j/, [Apr-22].

[11] "MT6797 LTE-A smartphone application processor functional specification for development board," https://www.96boards.org/documentation/consumer/mediatekx20/additional-docs/docs/MT6797_Functional_Specification_V1_0.pdf, [Apr-22].

[12] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of ICS*, 1991.

[13] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of ICS*, 2004.

[14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, 1990.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[16] T. Sherwood, E. Perelman, H. Greg, and B. Calder, "Automatically characterizing large scale program behavior," *Proceedings of ASPLOS*, 2002.

[17] A. Rohan, B. Panda, and P. Agarwal, "Reverse engineering the stream prefetcher for profit," in *Proceedings of EuroS&P*, 2020.

[18] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "A formal security analysis of even-odd sequential prefetching in profiled cache-timing attacks," in *Proceedings of HASP*, 2016.

[19] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proceedings of CCS*, 2018.