DEER: Deep Runahead for Instruction Prefetching *

Parmida Vahdatniya, Julian Humecki, Henry Kao, Tony Li, Ali Sedaghati, Fang Su*, Ruoyu Zhou*, Alex Bi*, Reza Azimi, Maziar Goudarzi

Huawei Technologies Canada Huawei, China*

***Abstract*—Modern mobile workloads face significant frontend stalls primarily due to their increasingly large code footprints and long repeat cycles. Existing instruction-prefetching methods often struggle with low coverage, poor timeliness, or high costs. We propose DEER, a software/hardware co-designed instruction prefetcher. It leverages profile analysis to extract metadata, enabling the hardware to prefetch the most likely future instruction cache-lines much earlier, hundreds of instructions ahead. The profile analysis is designed to skip over loops and recursions to look deeper into the future instruction stream. On the hardware side, a return-address stack helps prefetch on the return path from deep call stacks. The generated metadata table is stored in DRAM, requiring nearly no on-chip metadata storage because DEER's large lookahead depth allows metadata to be preloaded in time. Evaluation using gem5 on real-world modern mobile workloads shows up to a 45% reduction in L2 instruction-miss rate (19.6% on average), leading to speedups of up to 8% (4.7% on average). These performance improvements are up to 4 times larger than those from full-hardware record-and-replay prefetchers, while needing two orders of magnitude less on-chip storage.**

Keywords*—*Instruction prefetching, SW/HW co-design, path prediction, profile-guided optimization.

I. Introduction

Frontend stalls are a critical performance bottleneck because code footprint is growing faster than hardware tables can be economically scaled [1, 2]. Modern mobile applications are complex, using hundreds of libraries with thousands of functions and deep cross-library calls, along with JITted code. Unlike conventional benchmarks such as SPEC CPU [3] and GeekBench, [4] modern mobile workloads exhibit a long tail in PC repeat distance, meaning many unique instructions are executed between successive occurrences of the same PC. This results in frequent capacity misses in CPU frontend structures such as the instruction cache, TLB, and branch predictors, often leading to backend starvation where IPC is significantly lower than the ideal value. This paper specifically focuses on addressing I-cache misses, though the concept can extend to other structures.

Software instruction-prefetching, whether compiler-based [2, 5, 6] or co-designed [7], attempts to solve this but often incurs substantial and wasteful instruction overheads, particularly for prefetching on the return path from call stacks. Mobile SoCs often use big-little structures, which complicate static tuning of software prefetching due to differing cache sizes and sharing structures among the cores. Frequent cross-library calls occurring every few hundred instructions are another challenge, as compilers can only optimize *within a single* library. Fetch-directed instruction prefetching techniques [8, 9], relying on branch prediction, also perform poorly on these workloads because branch prediction accuracy drops significantly due to large branch footprints and long repeat distances. Record-and-replay (RnR) prefetchers [10, 11, 12, 13] achieve partial success but require hundreds of kilobytes of on-chip storage and the associated energy consumption.

DEER proposes a profile-guided self-correcting path-prediction mechanism for prefetching upcoming instruction cache lines via hardware-software collaboration. Software (compiler or binary analyzer) creates metadata from workload profile data, encoding stable execution paths both within and across libraries. DEER includes optimizations to make metadata concise, accurate, and representative of key execution flow aspects like control flow, loops, recursions, calls, and returns. A hardware engine uses this metadata and runtime information, such as a Return Address Stack (RAS), to look far ahead and do a timely prefetch of instruction cache lines. DEER can correct itself by reacting to runahead mispredictions and getting metadata for the new committed path. Compared to hardware-only RnR prefetchers, DEER offers higher agility, accuracy, and timeliness, while dramatically reducing on-chip storage and energy needed for training. The key contributions of DEER include a SW/HW co-designed prefetcher delegating training and analysis to software profiling (eliminating large hardware storage), profile analysis that bypasses loops/recursions for deeper future lookahead, a simple SW/HW interface with low context-switch overhead (single system register), choice of trigger points and metadata granule (Hyper Blocks) balancing accuracy and overhead, and evaluation on real-world mobile workloads.

II. Design

DEER is a software-hardware co-designed prefetcher. The software component, a compiler or binary optimizer, analyzes the program's control flow and call graph, augmented with profile data (execution frequency of basic blocks, branch/call target probabilities). From this, Hyperblocks (HBs) are formed (Fig. 1). An HB is a stable execution unit within a function, composed of basic blocks likely to execute sequentially with a probability threshold based on profile data. The HB information is then encoded into a Metadata Table stored in the program's address space in DRAM. Metadata generation involves a branch profiler using CPU PMU (like ARM BRBE) to get branch probabilities. A Path Profiler combines this with CFG and call graph info to form HBs along highly probable paths. We further detect loops/recursions for cycle skipping, and form HB metadata by chaining HBs within and across functions. The metadata is loaded into a non-cacheable memory area,

The starting address of this table is loaded into a hardware register, `HBT_PTR`. Each HB is characterized in the Metadata table with a trigger PC (the starting PC of the HB, in Fig. 1 the

* This manuscript is a summary of our paper, DEER: Deep Runahead for Instruction Prefetching on Modern Mobile Workloads, is available at https://arxiv.org/abs/2504.20387; this 3-page summary is initially auto-generated by notebooklm.google.com and then reviewed for correctness and consistency by the corresponding author.

trigger PC of HB1 is the first instruction of BB1). And the enclosing cacheline addresses of the path of HBs most likely to execute after it.

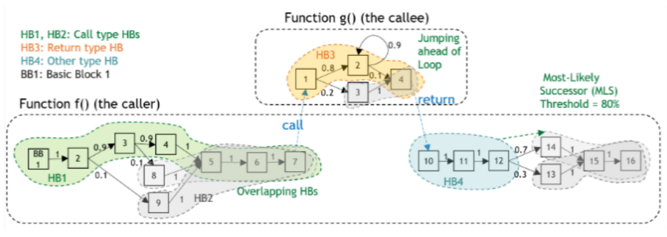

Figure. 1. DEER Hyperblock formation flow.

Figure. 2 shows the hardware component for the Deep Runahead Unit within the CPU core. The Deep Runahead Unit tracks program execution and uses the metadata table stored in memory. At specific trigger points, such as the commit of a call/return instruction, the Deep Runahead Unit identifies a set of upcoming HBs and passes their enclosing cacheline addresses to the Instruction-Fetch Unit or Load/Store Unit for prefetching into L1 I-cache or the unified L2 cache. We define trigger PCs as the target PCs of calls/returns, which designates an HB and initiates an MLS-guided prefetch.

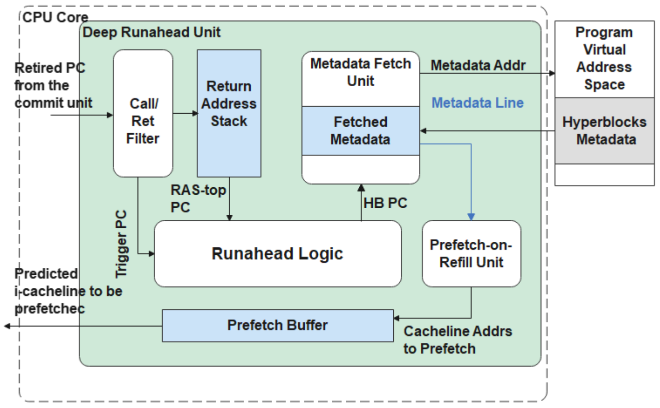

Figure. 2. DEER hardware overview

The system uses a Most Likely Successor (MLS) scheme as its building block to predict future execution paths. HBs are formed per function, and another MLS layer connects them cross-functionally.

While the HW can dynamically predict the upcoming HBs, SSRA (semi-static runahead) is the focus of the paper due to its simplicity and storage advantages. In SSRA, the runahead chain is computed statically before runtime. The metadata for each HB directly contains the entire list of I-cachelines in its MLS chain up to a certain point. This list is prefetched upon invoking the runahead mechanism. While calculating the runahead statically avoids the need for multiple memory lookups to traverse the HB chain, SSRA's static RAS limits how far it can trace paths involving returns from functions whose callers are not known statically.

The DEER hardware is non-speculative, triggering based on committed instructions. A Call/Ret filter identifies these instructions. The Trigger PC (target of the call/return) is used to generate a memory request for its SSRA metadata line. An additional request is issued for the PC at the top of the RAS to help recover depth lost in SSRA formation. We also prefetch-upon-refill, meaning the cachelines are added to the Prefetch Buffer when the request is returned from memory.

Figure. 3 Illustrates how the Metadata encoding is structured into 64-bit subentries, each covering three 512-byte regions, encoding base addresses and cacheline bitmaps. Deltas are used for addresses beyond the first region in each group. Each HB metadata entry is 16 Bytes and can encode up to 48 instruction cache lines. The offset in the metadata table is derived by hashing the HB-start PC, and the base address is loaded into the `HBT_PTR` register at program load or context switch. The metadata storage overhead in memory is marginal (around 9% of exercised code path, 2% of code+data footprint).

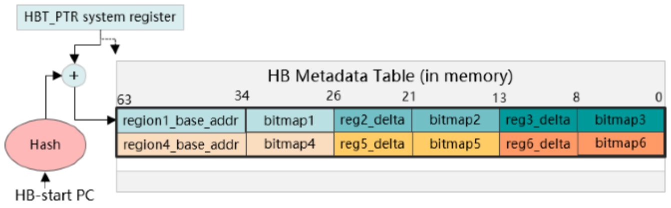

Figure. 3. HB metadata encoding and access scheme using the

The choice of call/return instructions as Trigger PCs is based on their frequency in mobile workloads (a call every ~50 instructions). The metadata granule is the target PC of every call or return. An optimization removes metadata for HB-chains fully contained within another chain.

III. Experimenal Results

Experiments were conducted using gem5 in SE mode with an O3 ARM core and a two-level cache hierarchy (256KB L1 I/D, 2MB unified L2). Evaluation used 15 simpoints captured from real mobile apps across various categories (news, games, video players, social networks, etc.). The simulated setup included a Stride Prefetcher in the L2 cache. DEER was compared against four rivals: two Record-and-Replay variants: 50-HB RnR, 50-Unique-HB RnR/Hierarchical-Prefetching*(HP*) [16]. And I-Spy* [14] and EFetch* [15], adapted to prefetch into L2 and without instruction overhead for fairness. DEER used SSRA with a max runahead depth of 50 HBs and enabled RAS-top prefetch by default.

Performance Gains: Figure. 4 shows the speedup (IPC) gains of 50-HB DEER (gained from an average L2 I-miss-rate reduction of 19.9%), 50-HB RnR (average L2 I-miss-rate reduction of 5.02%), 50-Unique-HB RnR/HP* (average L2 I-miss-rate reduction of 5.08%), I-Spy* (average L2 I-miss-rate reduction of 4.8%), and EFetch* (average L2 I-miss-rate reduction of 0.7%). This translates to average IPC gains of 4.7% for DEER, compared to around 1-2% for the rivals. DEER's superior gains are attributed to:

- Deeper runahead and call-stack prefetching.

- Skipping loops/recursions and prefetching return paths.
- Using most-likely path prediction based on profiles, which is more effective for workloads with less immediate repetition, unlike RnR methods that record the last observed path. Figure. 5 shows lower prediction accuracy for 50-Unique-HB RnR (HP*).

DEER effectively covers cold, capacity, and conflict misses. On smaller applications gains primarily come from covering cold misses, which is important given frequent context switches in mobile environments.

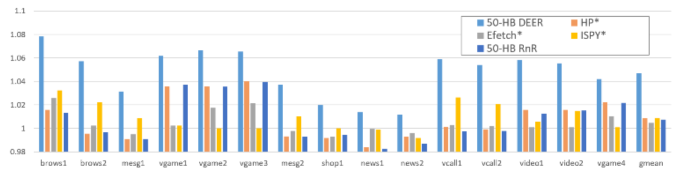

Figure. 4. DEER speedup compared to rivals.

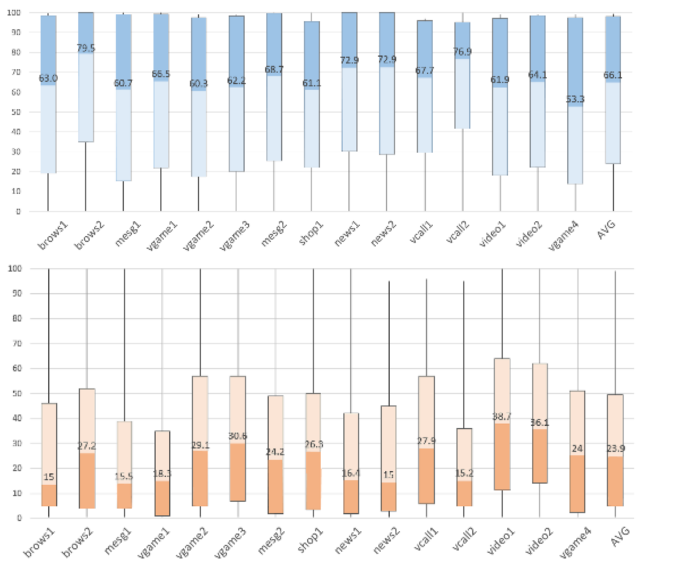

Figure. 5. DERR (above) average path prediction accuracy vs 50-Unique-HB RnR (HP*).

Prefetch Usefulness: Figure. 6 breaks down prefetches into "hit" (already in cache), "useful" (accessed after filling, covering cold or non-cold misses), and "evicted without use". The breakdown shows a positive relationship between usefulness and performance gains. Both cold and capacity/conflict misses contribute significantly to useful prefetches on average.

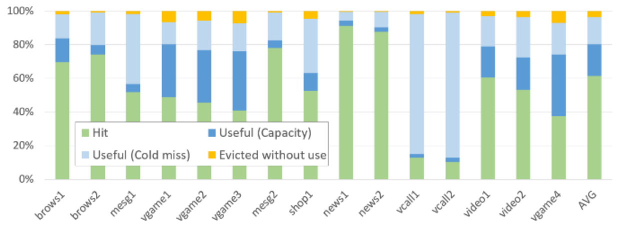

Figure 6 DEER prefetch usefulness.

Effective Runahead Depth: We observe and average of 4963 dynamic instructions and 469 static instructions as the effective prefetch runahead depth across these simpoints. The cycle-skipping feature allows going significantly deeper beyond loops, with a median of 2.2 cycles skipped, but up to 15x more in extreme cases. The SSRA chain covers 15.5 HBs on average, translating to an effective static runahead depth that helps timely prefetching, while 50-HBs-ahead was chosen as the maximum depth an SSRA chain aimed to achieve as a balance between timeliness and cache pollution.

Overheads: DEER's overheads are minimal:

- On-chip storage: Only about 304 bytes for the prefetch buffer, RAS, and fetched-metadata buffer. This is two orders of magnitude smaller than full-hardware RnR prefetchers.
- Metadata size on binary: Marginal compared to program binaries.
- Metadata storage in memory: Marginal (around 2.17% of memory-resident code+data footprint), even when using hashing (like cuckoo or Murmur3) which might double the footprint.

IV. Summary and Conclusion

DEER demonstrates that a simple co-designed profile-based path predictor can effectively forecast upcoming cache lines, enabling our co-designed prefetcher to timely prefetch them. By continuously and timely refilling the cache, DEER allows a smaller cache to achieve performance comparable to a much larger one lacking this feature. This is particularly valuable for mobile workloads due to frequent preemptions, complex big-little cache hierarchies, and deep cross-library calls.

DEER provides a low-cost, Kilo-instructions-deep instruction prefetch mechanism that effectively reduces frontend stalls in mobile workloads by covering cold, capacity, and conflict misses. It employs a static most-likely-path predictor that dynamically corrects itself based on retired control flow. The SW/HW interface is lightweight, requiring only setting a single system register pointing to the metadata table in memory, which is saved/restored on context switch. DEER works on binaries, and consequently can be applied even if the source codes of the libraries are not available. By offloading path prediction to software, DEER eliminates the need for expensive on-chip metadata storage, achieving over 4x higher gains than full-hardware rivals at two orders of magnitude lower cost.

References

[1] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 643–656. IEEE, 2018

[2] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front- International Symposium on Computer Architecture, ISCA ’19, page 462–473, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Spec home page. In https:https://www.spec.org/.

[4] Geekbenc home page. In https://www.primatelabs.com/.

[5] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: unified instruction supply for scale-out servers. In Proceedings of the 48th International Symposium on Microarchitecture, pages 166–177, 2015.

[6] Chi-Keung Luk and T.C. Mowry. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture, pages 182–193, 1998

[7] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 146–159, 2020.

[8] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. Re-establishing fetch-directed instruction prefetching: An industry perspective. In 2021 IEEE In-ternational Symposium on Performance Analysis of Systems and Software (ISPASS), pages 172–182. IEEE, 2021

[9] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pages 16–27. IEEE, 1999.

[10] Arm ltd. arm cortex-a78ae core technical reference manual revision r0p1, cpuectlr el1, cpu extended control register, el1.

[11] Arm ltd. arm cortex-x2 core technical reference manual r2p0, imp cpuectlr el1, cpu extended control register.

[12] Sam Ainsworth and Lev Mukhanov. Triangel: A high-performance, accurate, timely on-chip temporal prefetcher. arXiv preprint arXiv:2406.10627, 2024

[13] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 996–1008, 2019.

[14] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 146–159. https://doi.org/10.1109/MICRO50266.2020.00024

[15] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2014. EFetch: optimizing instruction fetch for event-driven webapplications. In Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (Edmonton, AB, Canada) (PACT '14). Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/2628071.2628103

[16] Tingji Zhang, Boris Grot, Wenjian He, Yashuai Lv, Peng Qu, Fang Su, Wenxin Wang, Guowei Zhang, Xuefeng Zhang, and Youhui Zhang. 2025. Hierarchical Prefetching: A Software-Hardware Instruction Prefetcher for Server Applications. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 529–544.