# Temperature-Guided Instruction Caching Using Page-Based Hardware Attributes

Henry Kao, Nikhil Sreekumar, Prabhdeep Singh Soni, Ali Sedaghati,
Fang Su*, Bryan Chan, Maziar Goudarzi and Reza Azimi
Huawei Technologies Canada, Toronto, Canada Huawei, Shenzhen, China*
Email: {henry.kao1, nikhil.sreekumar, prabhdeep.singh.soni3, ali.829657}@huawei.com
{fang.su, bryan.chan, maziar.goudarzi, reza.azimi1}@huawei.com

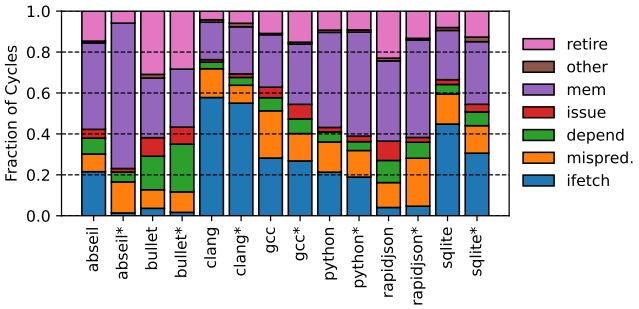


Fig. 1: Top-Down profiles of proxy mobile benchmarks. PGO'd benchmarks are marked with a "*".

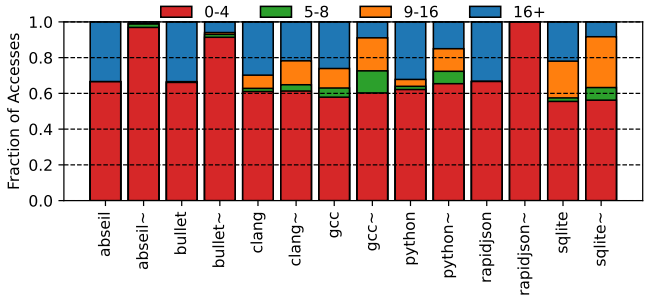


Fig. 2: Reuse distance distribution of hot cache lines measured in the L2 cache.

***Abstract*—Modern mobile CPU software pose challenges for conventional instruction cache replacement policies due to their complex runtime behavior causing high reuse distance between executions of the same instruction. Mobile code commonly suffers from large amounts of stalls in the CPU frontend and thus starvation of the rest of the CPU resources. We present a novel software-hardware co-design approach that enables the compiler to analyze, classify, and transform code based on "temperature" (hot/cold), and to provide the hardware with a summary of code temperature utilizing ARM's PBHA (Page-Based Hardware Attributes). The lightweight hardware extension employs code temperature attributes to optimize the instruction cache replacement policy resulting in the eviction rate reduction of hot code. TRRIP can reduce the L2 MPKI for instructions by 26.5% resulting in geomean speedup of 4.2%, on top of RRIP cache replacement running mobile code already optimized using PGO.**

## I. Introduction & Motivation

The system software on mobile platforms include shared libraries that implement user interface behavior, graphics rendering, hardware abstraction interfaces, inter-process communication, language runtimes, as well as interpreters and JIT (Just in Time)/AOT (Ahead of Time) compilers. All applications will call/use system software components during their execution, thus system software performance optimizations do not only pertain to specific applications; their effect will be observed system-wide spanning across all applications. We have observed that the system software components exhibit frontend bottlenecks due to instruction cache misses. Instrumentation PGO along with the associated code layout optimizations have shown considerable performance improvement, and is now integrated into the compilation flow of mobile system software (e.g., OpenHarmony [5]). Frontend bottlenecks however still appears to be the most prevalent bottleneck even with PGO applied in mobile software.

Figure 1 is a Top-Down analysis of the proxy benchmarks that are optimized without PGO, and with PGO (marked with an "*") on a simulation platform detailed later in Section III. Benchmarks *clang* [3], *gcc* [8], *python* [6] represent the interpreters and JIT/AOT class of system components. Google's *abseil* [1] represent calls to C++ libraries, *bullet* [4] for rendering, Tencent's *rapidjson* [7] for JSON parsing, and *sqlite* [9] for mobile database engine. Different input sets are used for profile generation and evaluation. All the benchmarks are built from source code using *-O3* with full LTO (Link-Time Optimization) enabled and use LLVM's IR instrumentation PGO. Fraction of cycles spent doing useful work/computation is shown as (`retire`) or stalled in the different stages of the CPU. Stalled cycles are collected from instruction fetch (`ifetch`), branch mispredictions (`mispred.`), data dependencies (`depend`), full issue queues (`issue`), and CPU backend data access due to latencies in accessing caches and DRAM (`mem`). Stalls not accounted for in the aforementioned stages are classified as `other`. Comparing non-PGO'd to PGO'd profiles, PGO reduces frontend stalls in the proxy benchmarks, but does not completely solve the problem. Stalls in the frontend are problematic as the subsequent stages of the CPU pipeline are starved of work.

PGO itself is able to classify code *temperature* (i.e., *hot*, *warm* and *cold*) by calculating the contribution of a region of code to the total amount of execution time. Code is classified as *hot* if it contributes a large portion to the total. Code is *cold* if it only contributes a negligible portion to the total. PGO in the compiler itself already makes a best effort to improve spatial locality of *hot* code, so we inspect the temporal locality instead by measuring reuse distance of the instruction cache lines corresponding to the *hot*. We measure reuse on a cache set granularity as the number of unique cache lines (both instruction and data) seen between two subsequent access of the same line. We target the L2 for our reuse distance measurements as misses in L2 incur considerably higher costs due to the increasing access latencies in downstream caches and main memory which are often off-chip. Figure 2 shows a breakdown of reuse distances of *hot* instruction cache lines at the unified L2 cache level.

The reuse distance is measured for each benchmark in two ways; (1) a base measurement where all unique lines are counted between subsequent accesses of a *hot* line and (2) and an optimistic measurement where only unique *hot* lines are counted between subsequent accesses of a *hot* line (post-fixed with "∼"). The former measures true temporal locality of *hot* lines in the benchmark and the latter measures temporal locality of *hot* lines in the absence of non-*hot* lines (i.e., *warm*, *cold* and data lines). Most of the *hot* instruction lines show high temporal locality in the base measurements since short reuse distances, from 0 to 4, makes up ∼60% of the accesses. Assuming a cache with a conventional replacement policy that inserts new lines as MRU (Most Recently Used) (i.e., LRU replacement), or promotes existing lines to MRU (e.g., RRIP replacement [12]), a cache with 4-way set associativity should be able to keep most of this portion of *hot* lines in the cache without misses. Up to 8-way set associativity should be able to keep most of the *hot* lines that make up 0-4 and 5-8 reuse distance in the distribution without incurring substantial misses. Hot cache lines that have reuse distance of 9 or greater will be evicted out of an 8-way set associative in conventional replacement policies. A large portion of the code that takes up a large portion of the benchmarks's instruction count, *hot* code, are evicted out of the cache before its next use due to poor temporal locality (i.e., high reuse distances). A considerable number of *hot* code evictions are caused by allocating non-*hot* lines to the cache set seen by comparing the base and optimistic reuse distance measurements. A method to prioritize *hot* lines and de-prioritize non-*hot* lines in cache sets would mitigate evictions on code lines most executed by the program.

Conventional and state-of-the-art cache replacement policies aim to keep frequently used memory in the caches for longer, and to keep infrequently used memory out of the caches. They are limited to seeing only short phases of the memory characteristics of the code due to hardware budget and the high costs of tracking detailed and deep history runtime behavior. Application behaviors spanning over long duration can only be realistically captured using software profiling techniques as in the case of PGO. Software profiling is able to classify code *temperature* for substantially longer execution windows. Cache replacement policies would benefit from knowing the *temperature* of instruction memory when deciding how to prioritize keeping it in the cache; the hotter the code, the longer the corresponding cache lines should stay in the cache. However this information can only be sustainably obtained using software mechanisms.

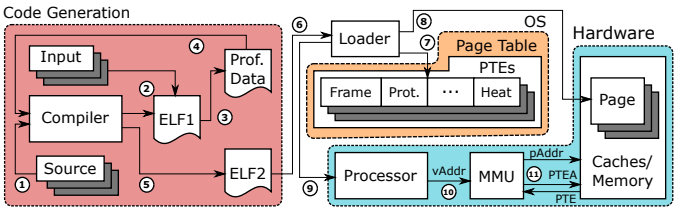


Fig. 3: Co-designed components and interfaces for temperature-guided cache replacement using ARM's PBHA.

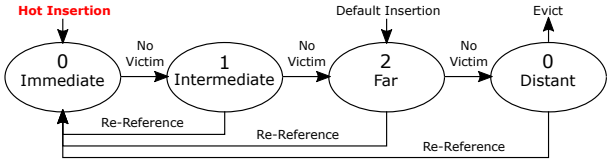


Fig. 4: Temperature-guided RRIP (TRRIP) policy using four states.

## II. Lightweight Temperature-Guided Instruction Caching Overview

We propose a software-hardware co-designed cache replacement policy that utilizes PGO to determine the insertion priority of instruction cache lines in the cache. Figure 3 illustrates the various interfaces between co-designed components labeled with numbers to follow the description of the technique. ① The flow starts with compiling program source code using a PGO-enabled compiler. ② The compiler will create the first executable, `ELF1`[1]. The executable is run with input sets representative of how it will be used in real-world scenarios, after which ③ a profile of the application will be generated. ④ The profile along with the original source of the program ① is fed back into the PGO-enabled compiler to re-optimized the code. ⑤ A final executable `ELF2` is generated. This re-optimized executable uses profiling results ④ to place code into different code sections of the ELF based on the classified temperature. Executing a program requires it to be loaded into memory first, which is the job of the loader. ⑥ The loader reads headers and sections in `ELF2` along with linking and runtime information to map the program to pages in memory. The loader calls functionality of the OS to allocate pages ⑧ for the executable, and also generate corresponding PTEs (Page Table Entry) ⑦, populating each entry with the runtime information obtained from `ELF2`. The typical information may include access permission flags (e.g.,

[1]The ELF (Executable and Linkable Format) executable format is predominantly used on Linux and Unix based systems, including mobile or embedded platforms, such as Android.

TABLE I: Simulator configuration.

| Component | Configuration |
|---|---|
| Core | 6-wide dispatch, 128 entry ROB, 2GHz, pseudo-FDIP prefetching |
| Branch | 1K-entry BTB, 512-entry indirect-BTB, 256-entry loop predictor, 1K-entry global predictor |
| L1-D | 64KB, 4-way, LRU replacement, 1/3 (tag/data)-cycle latency, stride prefetcher |
| L1-I | 64KB, 4-way, LRU replacement, 1/3 (tag/data)-cycle latency, stride prefetcher |
| Unified Shared L2 | 512kB (128kB per core), 8-way, TRRIP replacement, 8/12 (tag/data)-cycle latency, inclusive, stride prefetcher |
| Unified Shared SLC | 1MB, 16-way, LRU replacement, 10/30 (tag/data)-cycle latency, exclusive |
| DRAM | 8 chips/DIMM, 4 DIMMs, 7.6 GB/s controller bandwidth, 400-cycle latency |

read-only, read-write). ARM's PBHA (Page-Based Hardware Attributes) [2] offers additional implementation-defined PTE bits to be transferred with memory requests. We modify the loader to read and populate existing PBHA bits allocated in the PTE to store temperature-based information of the code sections in `ELF2` allowing a transparent lightweight interface to transfer information between software and hardware with minimal to no additional implementation cost, in contrast to prior art which requires new instructions [13], [14] or modification to the existing ISA [16], or additional logic/storage need in the CPU microarchitecture or caches [15], [17]. ⑨ The program executes conventionally after being loaded into memory. ⑩ Instructions of the loaded program are fetched from caches or memory. This may require the translation of the instruction addresses from virtual (`vAddr`) to physical (`pAddr`) by the MMU ⑪. PTE temperature bits in the MMU are read and transferred along with the memory requests to the caches. The replacement policies are augmented to react to the temperature bits in the memory requests to prioritize keeping hot instruction memory in the cache for longer, while keeping cold code out of the cache to reduce CPU frontend stalls due to instruction cache misses.

Figure 4 illustrates the temperature-guided cache replacement on top of RRIP [12], which we call TRRIP. Each state represents a re-reference prediction for a given cache line in a set. `Immediate` being the most likely to be re-referenced in the near future and thus is priotized in the cache set, and `Distant` being least likely to be re-refereneced in the near future, and first to be evicted. Baseline RRIP policies pessimistically insert cache lines at `Far/Distant` re-reference states and only get promoted to `Immediate` upon a subsequent cache hit. TRRIP directly inserts *hot* cache lines at `Immediate` since offline profiling has provided information that it is likely to be referenced more than other cache lines due to frequent execution. TRRIP priortizes *hot* cache lines and de-prioritize non-*hot* lines in cache to reduce evictions, and hence misses, on the most executed code lines in the program.

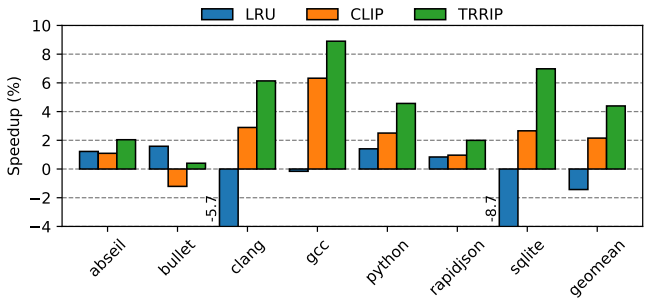


Fig. 5: Speedup of evaluated mechanisms normalized to SRRIP running the PGO'd benchmarks.

## III. Performance Impact & Conclusion

We simulate the co-designed temperature-based RRIP policy using Sniper [10] configured to represent energy-efficient cores of a heterogeneous mobile SoC (System on Chip) detailed in Table I. Different benchmarks have different duration of start-up code which we are not interested in. We fast forward to the hot part of the benchmark before starting performance measurements and then simulate 400M instructions afterward. Figure 5 shows speedup results for LRU, CLIP [11] and TRRIP cache replacement normalized to SRRIP [12] cache replacement. All evaluated techniques are applied to the L2 cache running the PGO'd benchmarks. SRRIP generally performs better than LRU and acts as a good foundation to build future techniques. Prior art CLIP blindly prioritizes all instruction cache lines on top of the RRIP policy. CLIP performs better on frontend bound applications compared to SRRIP with geomean speedup of 2%. TRRIP differs from CLIP by selectively prioritizing *hot* instruction cache lines using information provided by PGO. The selective priotization of *hot* instruction lines by TRRIP obtains a geomean speedup of 4.2%, greater than CLIP. TRRIP reduces L2 instruction MPKI (Misses Per Kilo-Instruction) by a geomean of 26.5%, compared to CLIP with a geomean L2 instruction MPKI reduction of 13.6%.

Our proposed temperature-guided co-design cache replacement mechanism requires no additional instructions nor modification to the ISA. It also incurs negligible power and area costs as we do not implement any additional hardware structures to track runtime information. Temperature information is provided using PGO and transferred with memory requests using ARM's PBHA implementation on the processor.

## References


[1] Abseil. https://github.com/abseil/abseil-cpp. Accessed: 2025-03-29.
[2] Arm® cortex®-a510 core technical reference manual – revision: latest. https://developer.arm.com/documentation/101604/latest/. Accessed: 2024-06-18.
[3] Clang source. https://github.com/llvm/llvm-project. Accessed: 2024-06-18.

[4] Llvm test suite. https://github.com/llvm/llvm-test-suite. Accessed: 2025-03-29.
[5] Openharmony. https://gitee.com/explore/harmony. Accessed: 2025-05-13.
[6] Python source. https://github.com/python/cpython. Accessed: 2024-06-18.
[7] Rapidjson. https://github.com/Tencent/rapidjson. Accessed: 2025-03-29.
[8] Spec cpu® 2017. https://www.spec.org/cpu2017/. Accessed: 2024-06-18.
[9] Sqlite. https://www.sqlite.org/. Accessed: 2024-06-18.
[10] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.
[11] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C. Steely, and Joel Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 343–353, 2015.
[12] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, New York, NY, USA, 2010. Association for Computing Machinery.
[13] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 816–829, New York, NY, USA, 2021. Association for Computing Machinery.
[14] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, ISCA 2021, June 2021.
[15] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A. Pokam, Simone Campanoni, and David I. August. Emissary: Enhanced miss awareness replacement policy for l2 instruction caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.
[16] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 742–756, New York, NY, USA, 2022. Association for Computing Machinery.
[17] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.