

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

Hot Superpage Buffers to Reduce Translation Overhead for Long Term Reused Pages

Permalink

<https://escholarship.org/uc/item/4g91c9wm>

ISBN

9798186381983

Author

He, Yuchen

Publication Date

2026-06-06

Peer reviewed|Thesis/dissertation

University of California
Santa Barbara

Hot Superpage Buffers to Reduce Translation Overhead for Long Term Reused Pages

A Thesis submitted in partial satisfaction
of the requirements for the degree

Master of Science
in
Electrical and Computer Engineering

by

Yuchen He

Committee in charge:

Professor Jonathan Balkind , Co-Chair
Professor Juan Luis Aragón, University of Murcia , Co-Chair
Professor Timothy Sherwood
Professor Dmitri Strukov

June 2026

The Thesis of Yuchen He is approved.

Professor Timothy Sherwood

Professor Dmitri Strukov

Professor Jonathan Balkind , Committee Co-Chair

Professor Juan Luis Aragón, University of Murcia , Committee Co-Chair

June 2026

Hot Superpage Buffers to Reduce Translation Overhead for Long Term Reused Pages

Copyright © 2026

by

Yuchen He

To my maternal grandfather, who left before I could say goodbye.

Acknowledgements

Upon finishing my master's degree, though two years are not long, I have so many people who have brought joy and support to me to thank.

I want to thank my supervisor, Prof. Jonathan Balkind for accepting me into ArchLab and giving me so much freedom to discover what I want to do. He also provided the opportunity for me to collaborate with Prof. Juan Luis Aragón, which finally led to this thesis work. In my journey at UCSB, Jon was not only an academic mentor to me; his kind and chill vibe always helped me out whenever I stressed myself out.

I want to thank Prof. Juan Luis Aragón, who provided the main idea of this thesis. He provided helpful insights and mentored me with great patience. As a naive student in the computer architecture area, I learned a lot from him. Due to the time zone differences, he had to meet with us outside of his working hours, but without a single complaint. Huge thanks for that!

I want to thank Prof. Behrooz Parhami, he provides a series of interesting courses which I enjoyed a lot. He helped me turn my research papers in his courses into publications, giving me a valuable starting point in academia. He is supportive of his students and he cares deeply about social issues not only through words but also through actions. He is a respectable person who stands up against injustice in his own way.

I want to thank Prof. Dmitri Strukov for giving me the chance to serve as teaching assistant in ECE154A, I appreciate this chance and am grateful for being trusted.

I also want to thank Prof. Timothy Sherwood for serving on my thesis committee even though he was under no obligation to do so. Thanks for your time!

Finally, I want to thank my friends and family who supported me emotionally and financially. Without them, I could not even have the chance to study at this beautiful campus.

I had a wonderful two years at UCSB, and I will never forget this chapter of my life.

Abstract

Hot Superpage Buffers to Reduce Translation Overhead for Long Term Reused Pages

by

Yuchen He

Graphs provide a natural abstraction for representing relationships and connectivity in real-world systems, including social networks, web pages, road networks, and recommendation systems. As a result, many modern applications rely on graph algorithms, making graph processing an active area of research across algorithms, software systems, and hardware architectures. However, graph algorithms are difficult to execute efficiently on modern processors, especially when processing large-scale input graphs. Much of their execution time is spent on traversing edges and accessing vertex data through pointer chasing and dependent memory accesses. These access patterns expose irregular memory-access behavior that stresses the cache and TLB hierarchies, leading to data-cache misses, TLB misses, and costly page table walks (PTWs). This thesis focuses on the address-translation behavior of graph workloads. We observe that PTWs are not uniformly distributed across the page footprint. Instead, a large fraction of PTWs is caused by a relatively small set of hot pages. These hot pages are not simply cold pages that are touched once and then disappear; rather, their translations exhibit long-term reuse, but their reuse distance is often too large for the conventional TLB hierarchy to capture consistently. Motivated by this observation, we explore how graph workloads can benefit from a specialized translation structure that exploits hot pages with long-term translation reuse. We propose the Hot Superpage (HSP) buffer, a secondary translation structure that retains translations evicted from the STLB and uses a lightweight hotness metric to distinguish repeatedly useful translations from cold entries.

Contents

Abstract	vii
List of Figures	x
List of Tables	xi
1 Introduction	1
1.1 Organization	3
2 Background	4
2.1 Virtual Memory and Address Translation	4
2.2 Huge Page and Translation Reach	5
2.3 Mitigating Address Translation Overhead	6
2.4 Graph Workloads	7
2.5 GAP Benchmark Suite	7
2.6 ChampSim Architectural Simulator	8
3 Hot Superpage Buffer Design	10
3.1 Design Motivation	10
3.2 HSP Buffer Organization	12
3.3 Hotness Tracking and Counter Decay	14
3.4 Replacement Policy	15
3.5 Promotion Policy	16
3.6 Hardware Organization and Implementation Considerations	17
4 Methodology	20
4.1 Workloads and Trace Generation	20
4.2 Evaluated Configurations	20
4.3 Evaluation Metrics	22

5	Experiment	23
5.1	Overall Page Table Walk Reduction	23
5.2	Overall Total Cycles Reduction	23
5.3	Anomaly of PageRank	24
5.4	Hotness Counter Overhead	26
6	Limitations and Future Work	27
6.1	Trace-Driven Simulation Limitations	27
6.2	Hardware Cost Modeling	27
6.3	Future Work	28
	Bibliography	30

List of Figures

2.1	x86 radix page table organization[1].	5
3.1	Coverage of translation by the top 5000 pages in GAPBS workloads. Bars show the percent of each workload captured by the top-k PTW pages. X-axis labels show selected pages as a percent of all pages.	11
3.2	Translation lookup flow with the HSP buffer placed after the STLB. An STLB miss probes the HSP buffer before initiating a PTW; an HSP hit supplies the translation without a PTW.	12
3.3	Conceptual layout of an HSP buffer entry. In addition to the translation information and access permissions, each entry maintains a hotness counter used for replacement and promotion decisions.	14
3.4	Effect of the promotion policy on translation behavior and total cycles for sssp . Results compare a 4K-entry HSP buffer without promotion against the same HSP buffer with promotion. Both configurations use a decay interval of 5M cycles and an initial hotness value of 0.	16
5.1	Reduction in PTWs relative to the baseline STLB configuration across GAPBS workloads. The 4K-entry L3 TLB and the 4K-entry HSP buffer provide equal additional translation capacity, whereas the huge L3 TLB is included as an optimistic high-capacity reference. For these results, the HSP buffer does not use promotion and applies counter halving every 10 million cycles. The same HSP policy parameters are used for all workloads.	24
5.2	Reduction in total simulated cycles relative to the baseline STLB configuration across GAPBS workloads. The 4K-entry L3 TLB and the 4K-entry HSP buffer provide equal additional translation capacity, whereas the huge L3 TLB is included as an optimistic high-capacity reference. For these results, the HSP buffer does not use promotion and applies counter halving every 10 million cycles. The same HSP policy parameters are used for all workloads.	25
5.3	Average and median HSP hotness-counter values for sssp (sampled every halving cycle).	26

List of Tables

3.1	Summary of STLB–HSP buffer interactions.	13
4.1	Graph applications used in the evaluation	21
4.2	Evaluated translation-hierarchy configurations.	21
5.1	PTWs and total cycles for pr across different configurations.	25

Chapter 1

Introduction

Graphs provide a natural abstraction for representing relationships and connectivity in real-world systems, including social networks, web pages, road networks, and recommendation systems. As a result, many modern applications rely on graph algorithms tailored to their specific needs, such as PageRank [2] (**pr**) for ranking web pages in search engines and Single-Source Shortest Path (**sssp**) for route planning in navigation systems. The widespread use and practical importance of graph applications have made graph processing an active area of research. Researchers have explored optimizations across multiple layers, including algorithms, software implementations, and hardware architectures, to improve the efficiency of graph processing.

However, graph algorithms are difficult to execute efficiently on modern processors [3], especially when processing large-scale input graphs. This difficulty comes from the connection-centric nature of graph processing, where much of the execution time is spent on traversing edges and accessing vertex data, often through pointer chasing and dependent memory accesses. These access patterns lead to low utilization of compute resources and expose irregular memory-access behavior that stresses the data-cache and TLB hierarchies. Prior workload characterization [4] further showed that graph workloads can

be limited by poor memory-level parallelism, which can leave memory bandwidth underutilized even though graph workloads are memory intensive.

Most prior architectural optimizations for graph workloads focus on the data-memory side, including improving data supply[5], cache management[6], prefetching[7], or traversal scheduling[8]. In contrast, address translation [9] has received comparatively less attention, even though TLB misses can introduce substantial memory-access overhead through PTWs. As other memory-system bottlenecks are reduced, address translation can become a more visible bottleneck for graph workloads. This motivates a closer examination of translation behavior in graph workloads and whether workload-guided translation structures can reduce this overhead.

In this work, we explore how graph workloads can benefit from a specialized translation lookaside buffer (TLB) structure that exploits translations with long-term reuse. Our key observation is that translation with long-term reuse is not uniformly distributed across the page footprint. Instead, a relatively small set of hot pages accounts for a substantial fraction of costly PTWs. These hot pages are different from cold pages that are touched only once or rarely reused. They remain useful over long execution intervals, but their reuse distance is often too large for the conventional TLB hierarchy to capture consistently.

Motivated by this observation, we target hot pages with long-term reuse rather than uniformly enlarging the TLB hierarchy. To preserve such translations beyond the capacity of the STLB, we propose a Hot Superpage (HSP) buffer, a secondary translation structure that stores translations evicted from the STLB and tracks their usefulness using a lightweight hotness metric. The HSP buffer therefore acts as a hotness-aware translation-retention mechanism rather than a simple L3 TLB. We evaluate this design in ChampSim [10] against both the baseline hierarchy and an idealized huge L3 TLB configuration that serves as an optimistic high-capacity reference for translation perfor-

mance.

1.1 Organization

Chapter 2 provides background on virtual memory, address translation, graph workloads, the GAP Benchmark Suite, and the ChampSim architectural simulator. Chapter 3 presents the design of the Hot Superpage buffer, including its organization, hotness tracking mechanism, decay policy, and interaction with the existing TLB hierarchy. Chapter 4 describes the experimental methodology, including workload trace generation, evaluated translation configurations, and evaluation metrics. Chapter 5 presents the experimental results and analyzes the impact of the HSP buffer on PTWs and total cycles. Chapter 6 discusses limitations and future work.

Chapter 2

Background

2.1 Virtual Memory and Address Translation

Virtual memory provides a flexible and secure programming model by allowing each process to operate within its own virtual address space. This abstraction improves isolation between processes and can allow the memory footprint of applications to exceed the capacity of physical memory. However, it also introduces overhead because virtual addresses must be translated into physical addresses before memory can be accessed.

Modern processors reduce this translation overhead using translation lookaside buffer (TLB), which caches recently used virtual-to-physical address translations. When a requested translation is not found in the TLB hierarchy, the processor must perform a page-table walk (PTW) to locate the corresponding page-table entry (PTE) in memory.

In common x86-64 systems, page tables are organized as multi-level radix trees. Fig.2.1 depicts the structure of a four-level page table, which is used in the x86-64 architecture. The virtual page number is divided into several index fields, and each field selects an entry at one level of the page table. The entry read from one level provides the base address of the next level, until the final PTE provides the physical page number.

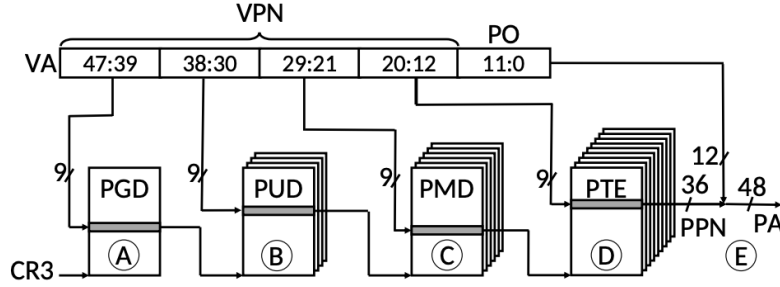


Figure 2.1: x86 radix page table organization[1].

Therefore, Each memory access in a multi-level PTW depends on the PTE obtained from the previous level. This dependency chain makes PTWs difficult to parallelize and can add significant latency to the memory request.

2.2 Huge Page and Translation Reach

The amount of virtual memory covered by a TLB is limited by both the number of entries it contains and the size of each page. This coverage is often referred to as the TLB reach. When an application's translation working set exceeds the reach of the TLB hierarchy, translations may be evicted before they are reused, causing additional TLB misses and PTWs. A straightforward way to increase translation coverage is to add more TLB entries, but larger TLBs can increase access latency, area, and energy.

Huge pages increase TLB reach by mapping a larger virtual memory region with a single translation. For example, a 2 MB page covers 512 times more memory than a 4 KB page with one TLB entry. Huge pages can also reduce the number of page-table levels traversed during a PTW and reduce the total number of page-table entries needed to describe a large memory footprint. As a result, huge pages can substantially reduce address-translation overhead for applications with large memory footprints.

However, huge pages also introduce practical challenges. Traditional huge pages require contiguous virtual memory and physical memory. In long-running systems, physical memory can become fragmented due to allocation and deallocation. Huge pages may also introduce internal fragmentation due to page retirement and require operating-system policies for page promotion and demotion. These limitations motivate complementary approaches that reduce translation overhead without relying entirely on larger page mappings.

2.3 Mitigating Address Translation Overhead

Prior work on reducing address translation overhead focus on two directions: increasing TLB reach and reducing PTW latency. The first direction increases the amount of virtual memory that can be covered by translation structures. Huge pages are a common approach because a single TLB entry can map a much larger virtual memory region than a regular 4KB page. Other work exploits translation contiguity by compressing or coalescing groups of contiguous translations into fewer TLB entries [11]. Du et al. proposed a superpage mechanism that can support mappings to non-contiguous physical memory [12]. This approach addresses one practical limitation of conventional huge pages, where fragmented physical memory in long-running systems can make it difficult to allocate large contiguous physical regions.

The second direction reduces the latency of PTWs after a TLB miss occurs. Because PTEs are stored in memory and may be cached in the conventional cache hierarchy, optimizations can target the placement, caching, and prefetching of PTEs. For example, prior work has proposed pinning hot PTEs in the last-level cache (LLC) [13], prefetching PTEs to hide future PTW latency [14, 1], and using dedicated MMU caches to store intermediate page-table entries and reduce page-walk latency [15].

2.4 Graph Workloads

Graph workloads are known for their memory-intensive behavior, irregular memory access patterns, and large memory footprints. Unlike computation-intensive workloads, which often access memory in regular and predictable patterns, graph algorithms frequently traverse sparse and pointer-rich data structures such as adjacency lists and vertex/edge arrays. These accesses often involve indirect indexing and data-dependent control flow, which reduce spatial and temporal locality.

As a result, graph workloads place significant pressure on both the data-cache hierarchy and the TLB hierarchy. Poor cache locality can increase the number of data-cache misses, while large and irregular address footprints can exceed the coverage of the TLB hierarchy and lead to frequent PTWs. This makes graph workloads a useful target for studying address translation overhead and translation-aware microarchitectural optimizations.

2.5 GAP Benchmark Suite

The GAP Benchmark Suite (GAPBS) is a benchmark suite designed to standardize the evaluation of graph-processing systems and graph algorithm implementations [16]. It provides optimized baseline implementations of representative graph kernels. Because graph workloads are sensitive to implementation details, graph structure, and input size, using a standardized benchmark suite helps make evaluation results more reproducible and comparable across studies.

This thesis uses GAPBS as the source of graph workloads. GAPBS includes a set of representative graph kernels, such as Breadth-First Search (**bfs**), Single-Source Shortest Paths (**sssp**), Connected Components (**cc**), Betweenness Centrality (**bc**), Triangle

Counting (**tc**), and PageRank (**pr**). These workloads cover a range of common graph-analysis tasks and are therefore useful for evaluating memory-system behavior in graph applications.

In this work, GAPBS applications are used to generate instruction traces that are later replayed in ChampSim. The experimental evaluation focuses on **bfs**, **sssp**, **cc**, **bc**, **tc** and **pr**. For synthetic input graphs, this work uses the GAPBS Kronecker graph generation option, which follows the Graph500-style[17] graph generation methodology used by GAPBS.

2.6 ChampSim Architectural Simulator

ChampSim is a trace-driven architectural simulator designed for computer architecture research, education, and competition environments [10]. Its design emphasizes configurability, modularity, and relatively low startup cost, making it well suited for evaluating microarchitectural mechanisms such as cache replacement policies, prefetchers, branch predictors, and TLB-related structures. Rather than executing the application directly during simulation, ChampSim replays instruction traces generated from prior program execution. This trace-driven methodology enables repeatable experiments over the same dynamic instruction stream while allowing the simulator configuration to be changed across runs.

ChampSim models an out-of-order core, cache hierarchy, memory system, and virtual-memory hardware support. In its virtual-memory model, instruction and data memory references are translated through TLB structures before accessing the cache hierarchy. On a miss in the STLB, requests are forwarded to the page table walker, which performs memory accesses to obtain the required translation. PTEs are accessed through the memory hierarchy similarly to ordinary data, so PTWs can incur additional latency

when the required entries are not cached.

The configurability of ChampSim makes it useful for this thesis because the TLB hierarchy can be modified to evaluate alternative translation structures under the same workload traces.

Chapter 3

Hot Superpage Buffer Design

3.1 Design Motivation

To motivate HSP buffer, we first study whether irregular graph workloads exhibit concentrated long-term translation reuse. We classify translation reuse into three categories.

- Short-term reuse: The translation is reused soon enough to remain in the L1 TLB. The subsequent access to the same page is therefore satisfied by an L1 TLB hit.
- Medium-term reuse: The translation has been evicted from the L1 TLB but is still retained in the STLb. The subsequent access to the same page is therefore satisfied by an STLb hit.
- Long-term reuse: The translation has been evicted from the conventional TLB hierarchy before it is reused. The subsequent access to the same page therefore triggers a TLB miss and a PTW.

Fig.3.1 shows the translation coverage of the top 5000 pages in six GAPBS workloads. These pages represent only a small fraction of the total page footprint in most workloads,

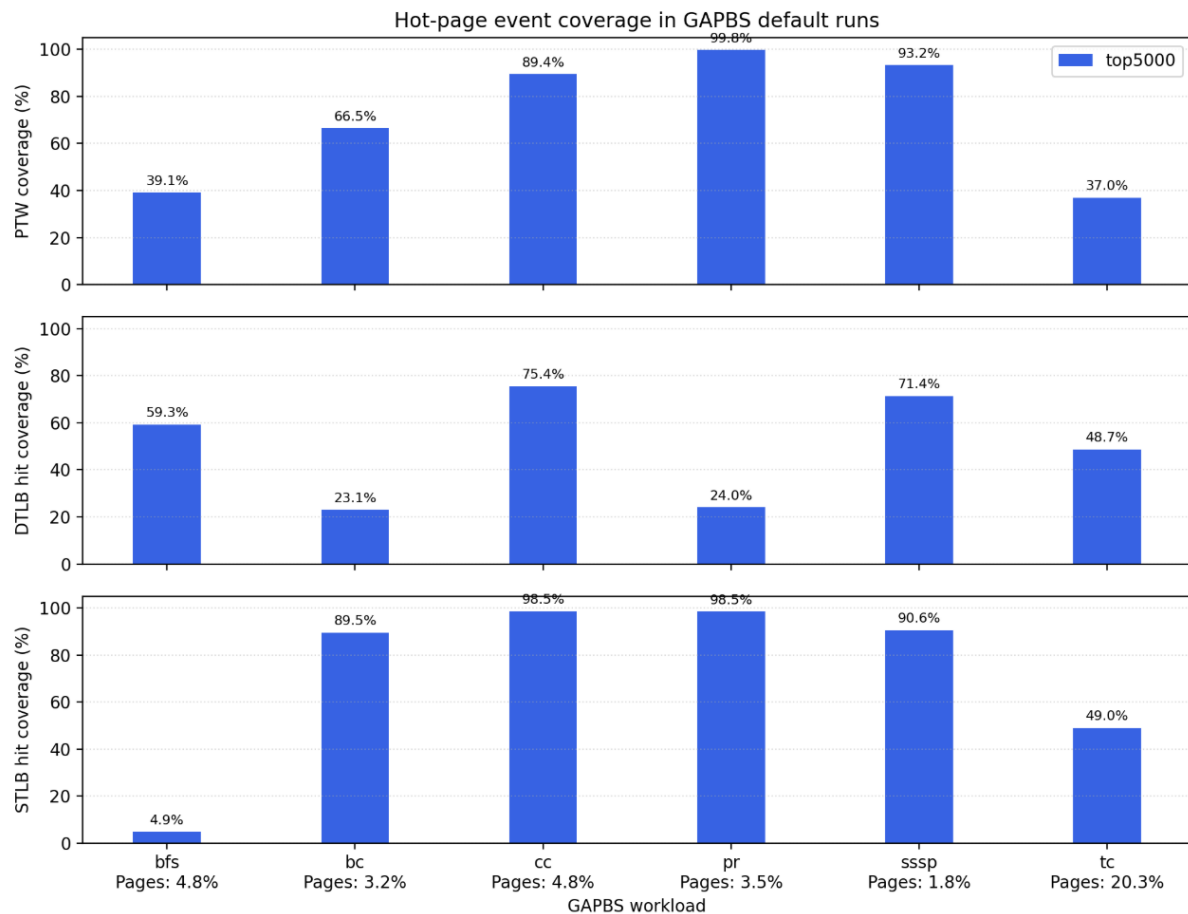


Figure 3.1: Coverage of translation by the top 5000 pages in GAPBS workloads. Bars show the percent of each workload captured by the top-k PTW pages. X-axis labels show selected pages as a percent of all pages.

but they account for a large fraction of PTWs. For example, the top 5000 pages cover 89.4% of PTWs in CC, 93.2% in SSSP, and 99.8% in PR, while representing only 4.8%, 1.8%, and 3.5% of all pages, respectively. BC also shows a clear concentration pattern, where 3.2% of pages are responsible for 66.5% of PTWs. TC is included for completeness, but because its dynamic instruction count is much larger, our current simulation covers only a partial trace. Therefore, we do not draw conclusions from the TC result.

This result indicates that long-term reuse is not uniformly distributed across the page footprint; instead, it is concentrated in a relatively small set of hot pages. Importantly,

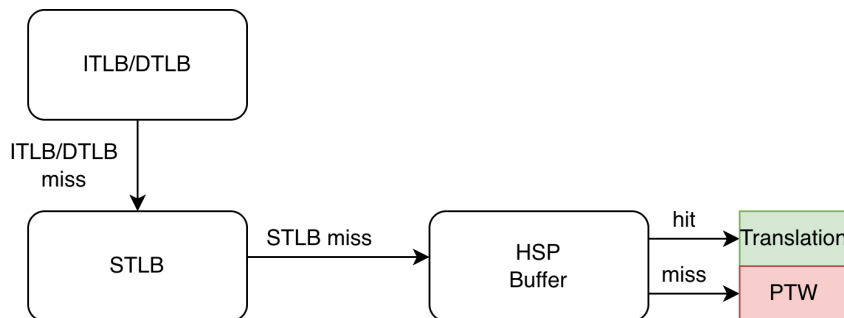


Figure 3.2: Translation lookup flow with the HSP buffer placed after the STL B. An STL B miss probes the HSP buffer before initiating a PTW; an HSP hit supplies the translation without a PTW.

these PTW-heavy pages are not merely cold pages that are touched once and then disappear. The same top-PTW pages also account for substantial DTLB and STL B hit coverage in several workloads. This means that these pages remain active throughout execution and exhibit short-term and medium-term reuse at multiple time scales. Therefore, graph workloads contain pages that are hot over the execution but whose reuse distance is often too long to be consistently captured by the existing L1 TLB and STL B.

These observations motivate a dedicated mechanism for hot-page tracking. Since long-term reuse is both significant and concentrated, treating all pages uniformly wastes capacity on pages that provide little PTW reduction opportunity. A smart structure such as HSP can instead focus on identifying and retaining translations for the small subset of pages that repeatedly cause PTWs, thereby targeting the long-term reuse that the conventional TLB hierarchy fails to capture.

3.2 HSP Buffer Organization

The HSP buffer is designed to exploit hot pages that are repeatedly reused over long time intervals but are often evicted from the conventional TLB hierarchy before their next reuse. Its operation has three main components. First, the HSP buffer retains valid

Table 3.1: Summary of STLB–HSP buffer interactions.

STLB	HSP Buffer	Condition	Action
Hit	–	–	Return the translation from the STLB; do not access the HSP buffer.
Miss	Hit	Promotion disabled, or enabled but below threshold	Return the translation from the HSP buffer, increment hotness, and retain the entry in the HSP buffer.
Miss	Hit	Promotion enabled and threshold reached	Return the translation from the HSP buffer, increment hotness, and promote the entry into the STLB. If a valid STLB entry is evicted, insert that victim into the HSP buffer.
Miss	Miss	–	Perform a PTW and fill the returned translation into the STLB. If a valid STLB entry is evicted, insert that victim into the HSP buffer.

translations evicted from the STLB instead of discarding them immediately. Second, each HSP entry maintains a hotness counter so that translations that repeatedly avoid PTWs can be distinguished from cold entries. Third, the design supports an optional promotion policy that moves a useful HSP entry back into the STLB after an HSP hit. In the overall evaluation, promotion is disabled unless explicitly stated otherwise. The promotion policy is evaluated separately in Section 3.5.

The HSP buffer is organized as a secondary translation structure placed behind the STLB. Fig3.2 shows the translation lookup flow. When a translation request misses in the first-level TLBs and the STLB, the HSP buffer is checked before invoking a PTW. On an HSP hit, the requested translation is supplied from the HSP buffer, avoiding a PTW. On an HSP miss, the request proceeds to the page table walker in the same way as in the baseline design. Table 3.1 summarizes the possible interactions between the STLB and the HSP buffer in detail.

Fig. 3.3 shows the internal layout of an HSP buffer entry. The entry is similar to

HSP Buffer Entry

Valid	VPN	PPN	Permission	Hotness Counter
1 bit	52 bits	52 bits	R/W/X	N bits

Figure 3.3: Conceptual layout of an HSP buffer entry. In addition to the translation information and access permissions, each entry maintains a hotness counter used for replacement and promotion decisions.

a conventional STLB entry, except that the replacement-state bits are replaced with a hotness counter. Therefore, replacement in the HSP buffer is based on the hotness value rather than the conventional replacement state. Entries with low hotness values are treated as better replacement candidates, while entries that repeatedly hit in the HSP buffer are preserved for longer.

3.3 Hotness Tracking and Counter Decay

Hotness is the primary mechanism used to distinguish cold translations from translations that are likely to remain useful. In the proposed design, each entry in the HSP buffer maintains a hotness counter. On an HSP buffer hit, the corresponding counter is incremented by one.

When an HSP entry is promoted back into the STLB, its hotness value is preserved but is not further modified while the entry resides in the STLB. This design choice is based on the assumption that a page with long-term reuse may experience multiple short-term reuse periods throughout its lifetime. Once a translation is promoted into the STLB, subsequent accesses are expected to be captured by the conventional TLB hierarchy. Therefore, the HSP hotness counter is intended to measure the possibility of an entry to be a page with long-term reuse, rather than counting every short-term reuse

within the STLB.

Hotness information can become stale over time. A translation that was useful during one phase of execution may stop being reused in a later phase. Without an aging mechanism, such entries could retain high hotness values indefinitely and prevent newly useful translations from entering the HSP buffer. To avoid this problem, the HSP buffer periodically decays the hotness counters of entries currently resident in the HSP buffer. Entries that have been promoted back into the STLB are not included in this periodic halving operation while they reside in the conventional TLB hierarchy. This keeps the decay logic local to the HSP buffer and avoids adding counter-update or counter-decay logic to the L1 TLB or STLB.

In this thesis, decay is implemented by halving each hotness counter at a fixed cycle interval. Halving provides gradual aging that reduces the influence of old HSP hits while preserving the relative differences among entries. Compared with resetting counters to zero, halving is less aggressive because entries that have accumulated strong evidence of long-term reuse are not immediately discarded. At the same time, entries that stop receiving HSP hits eventually decay toward zero and become replacement candidates.

3.4 Replacement Policy

The HSP buffer uses a minimum-hotness replacement policy. When a valid STLB victim must be inserted into a full HSP buffer, the resident entry with the lowest hotness counter value is selected for replacement.

This policy differs from conventional recency-based TLB replacement. The HSP buffer targets translations with long-term reuse, so recent use alone is not sufficient to determine whether an entry should be preserved. Instead, the hotness counter records whether an entry has actually avoided PTWs after being inserted into the HSP buffer.

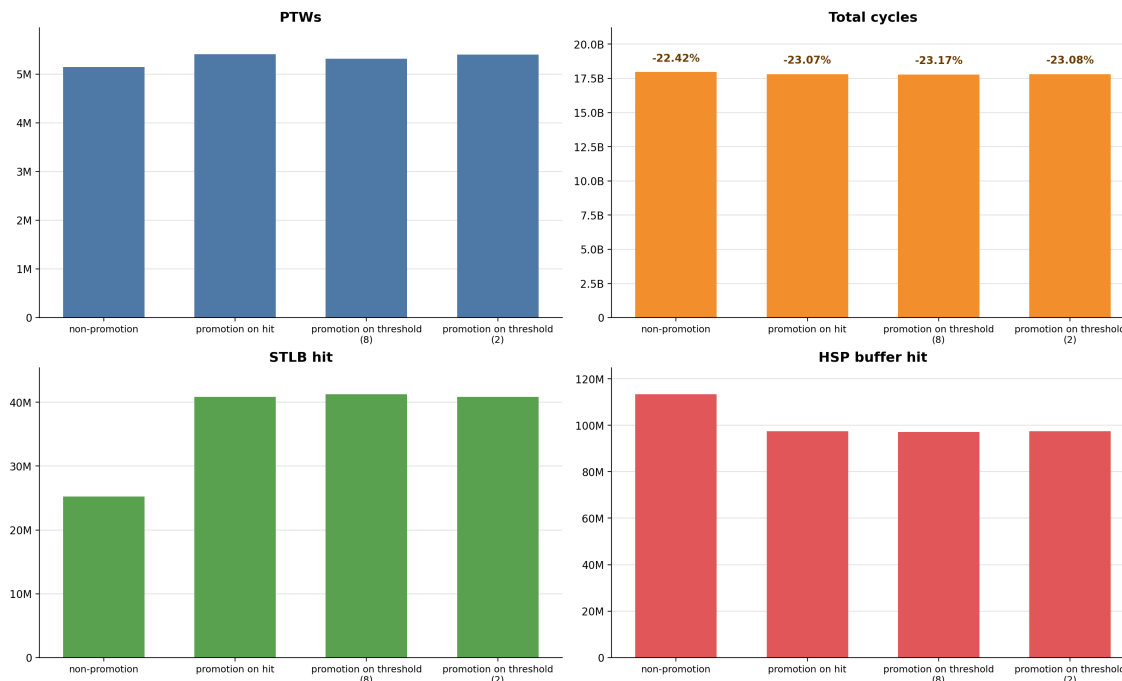


Figure 3.4: Effect of the promotion policy on translation behavior and total cycles for `sssp`. Results compare a 4K-entry HSP buffer without promotion against the same HSP buffer with promotion. Both configurations use a decay interval of 5M cycles and an initial hotness value of 0.

Entries that repeatedly hit in the HSP buffer accumulate higher hotness and are less likely to be replaced. Entries that do not produce hits remain cold and are preferentially evicted.

3.5 Promotion Policy

When promotion is enabled, an HSP hit first increments the matching entry’s hotness counter. If the updated counter reaches the promotion threshold, the entry is promoted into the STLb. Otherwise, it remains in the HSP buffer.

The motivation for promotion is that an HSP hit may indicate that the page is entering a short-term or medium-term reuse period. Promoting the translation back into the STLb allows subsequent accesses to benefit from the lower latency of the conven-

tional TLB hierarchy. Promotion can also free HSP capacity for other long-term reuse candidates. Fig. 3.4 compares STLb hits, HSP buffer hits, PTWs, and total cycles for HSP buffers with and without the promotion for `sssp`. Without promotion, the HSP buffer has significantly fewer STLb hits than the design with promotion. This is because some HSP buffer hits are converted into STLb hits when promotion is enabled. Overall, promotion provides an additional 0.6% reduction in total cycles. Although this improvement is modest, we expect the benefit to become larger when the HSP buffer capacity is more constrained.

3.6 Hardware Organization and Implementation Considerations

3.6.1 Evaluated Model and Practical Organization

In this thesis, the HSP buffer is initially modeled as a 4k-entry fully associative structure with a fixed lookup latency of 12 cycles. This configuration serves as an experimental starting point for evaluating the potential benefit of retaining evicted translations with long reuse intervals. It is not intended to represent a final hardware implementation, nor does it imply that a monolithic 4096-way associative lookup is the preferred design.

A more realistic organization would use multiple smaller HSP buffers rather than one large monolithic structure. For example, each HSP buffer could contain 512 entries, corresponding to the number of 4KB pages covered by a conventional 2MB superpage. This organization motivates the name Hot Superpage: each HSP buffer provides superpage-sized translation capacity while still storing individual 4KB translations. Unlike a real 2MB huge page, however, an HSP buffer does not require the operating system to create a large-page mapping or allocate contiguous physical memory. Instead, it retains selected

translations that have been evicted from the STLB and are likely to be reused later.

This organization reduces the hardware difficulty compared with a single 4K-entry associative structure. A 512-entry associative HSP buffer is still non-trivial, but it is easier to justify than a monolithic 4096-way search. One possible optimization is to use a Bloom filter as a low-cost membership test before performing the associative lookup. When a translation is inserted into an HSP buffer, its virtual page number is also recorded in the Bloom filter. On a later STLB miss, the requested virtual page number first queries the Bloom filter. If the Bloom filter returns negative, the translation is guaranteed not to be present in that HSP buffer, so the associative lookup can be skipped. If the Bloom filter returns positive, the translation may be present, and the HSP buffer performs an associative tag lookup to confirm the hit. False positives only introduce unnecessary associative lookups, while correct hits are not lost.

3.6.2 Minimum-Hotness Victim Selection

A possible hardware implementation for selecting the minimum-hotness entry is a hierarchical comparison tree. The leaves of the tree receive the hotness values of candidate HSP entries, and each internal node forwards the entry with the smaller hotness value. The root of the tree therefore identifies a minimum-hotness replacement candidate. If multiple entries have the same minimum hotness value, the implementation can break ties using a simple round-robin or pseudo-random policy. Since replacement selection is required only when a new entry is inserted, this comparison logic does not need to be on the critical path for returning an HSP hit. It can be pipelined or maintained as background state, especially in a future implementation that uses multiple smaller HSP buffers rather than one monolithic structure.

3.6.3 Preserving Promotion Information

Maintaining full hotness counters for all entries in the conventional TLB hierarchy would introduce additional hardware overhead in real hardware. To reduce this cost, a potential design can instead use a single promotion-status bit per TLB entry. This bit records whether the entry was previously promoted from the HSP buffer. When the entry is later evicted from the STLB and inserted back into the HSP buffer, the hotness counter is initialized according to this bit. If the promotion-status bit is set, the HSP hotness counter is initialized to a fixed positive value. Otherwise, it is initialized to zero. This approximation preserves information about previous HSP usefulness without requiring full hotness counters inside the conventional TLB structures.

Chapter 4

Methodology

4.1 Workloads and Trace Generation

We use Intel Pin [18] to generate traces from GAPBS [16]. Table 4.1 summarizes the graph-generation configuration for each application. The approximate dynamic instruction counts include only instructions executed by the application, excluding graph generation. We use the `-g 22` flag to generate Kronecker graphs with 2^{22} vertices. In GAPBS, the default degree is 16, and the generated graph is automatically symmetrized. As a result, each graph application uses a graph with 2^{27} edges. Because `tc` executes approximately 220 billion dynamic instructions, which is substantially larger than the other evaluated workloads, we simulate only a partial trace of 3 billion instructions for `tc`.

4.2 Evaluated Configurations

We evaluate four translation-hierarchy configurations, summarized in Table 4.2. The baseline configuration uses ChampSim’s default TLB hierarchy, with private first-level

Table 4.1: Graph applications used in the evaluation

Graph application	Approximate dynamic instruction counts	Vertices	Edges
sssp	3B	2^{22}	2^{27}
bfs	200M	2^{22}	2^{27}
tc	220B	2^{22}	2^{27}
cc	400M	2^{22}	2^{27}
bc	4B	2^{22}	2^{27}
pr	4.5B	2^{22}	2^{27}

Table 4.2: Evaluated translation-hierarchy configurations.

Configuration	Organization	Entries	Latency
Baseline STLB	128 sets, 12 ways	1536	4 cycles
4K-entry L3 TLB	256 sets, 16 ways	4096	12 cycles
Huge L3 TLB	2048 sets, 16 ways	32768	40 cycles
HSP buffer	Fully associative	4096	12 cycles

instruction and data TLBs and a shared STLB. In ChampSim, TLB structures are identical to caches, and STLB misses are handled by a hardware page-table walker whose memory accesses pass through the data-cache hierarchy [10]. The baseline STLB has 128 sets, 12 ways, and a 4-cycle lookup latency.

The second configuration adds a 4K-entry L3 TLB after the STLB. This configuration has the same number of entries as the proposed HSP buffer and is used as a capacity-matched comparison point. This comparison helps separate the benefit of simply adding more translation capacity from the benefit of the HSP buffer’s hotness tracking.

The third configuration adds an idealized large L3 TLB after the STLB. This structure has 2048 sets, 16 ways, and a 40-cycle lookup latency. We use this configuration as an optimistic reference point to estimate the benefit of capturing long-term translation reuse with substantially larger translation capacity. Because of its large size, this configuration is not intended as a low-cost practical design.

The fourth configuration evaluates the proposed Hot Superpage (HSP) buffer. The HSP buffer is placed behind the STLB and stores selected translations evicted from the

STLB. In this work, the HSP buffer contains 4K fully associative entries and has a 12-cycle lookup latency. Each entry maintains a hotness counter, which is incremented on reuse and periodically decayed. This design targets translations with long-term reuse while avoiding the cost of a large conventional L3 TLB.

4.3 Evaluation Metrics

The primary metrics collected from ChampSim are the number of PTWs and total simulated cycles. PTWs directly measure the effectiveness of a translation mechanism in avoiding costly address translation misses, while total cycles capture the overall performance impact after accounting for other effects such as data-cache misses, memory-system behavior, and the latency of additional translation structures.

Chapter 5

Experiment

5.1 Overall Page Table Walk Reduction

Fig. 5.1 shows the overall PTW reduction for each workload relative to the baseline. In general, a huge L3 TLB provides an estimated lower-bound on the number of PTWs across most workloads. The main exception is **pr**, where the PTW count is nearly the same as that of the HSP buffer; we discuss this anomaly later. Across the workloads, the HSP buffer demonstrates the benefit of hotness-based tracking, which preserves long-term reuse information. Compared with an L3 TLB of the same capacity, the HSP buffer achieves additional PTW reductions of 2.9%, 8.8%, 13.3%, 14.1%, and 32.0% for **tc**, **sssp**, **cc**, **bfs**, and **bc**, respectively, excluding **pr**.

5.2 Overall Total Cycles Reduction

Fig. 5.2 shows the reduction in total cycles. The performance improvement is modest because we charge the HSP-buffer lookup latency on every HSP-buffer hit, and Champ-Sim models an out-of-order (OoO) core in which not every HSP-buffer hit lies on the

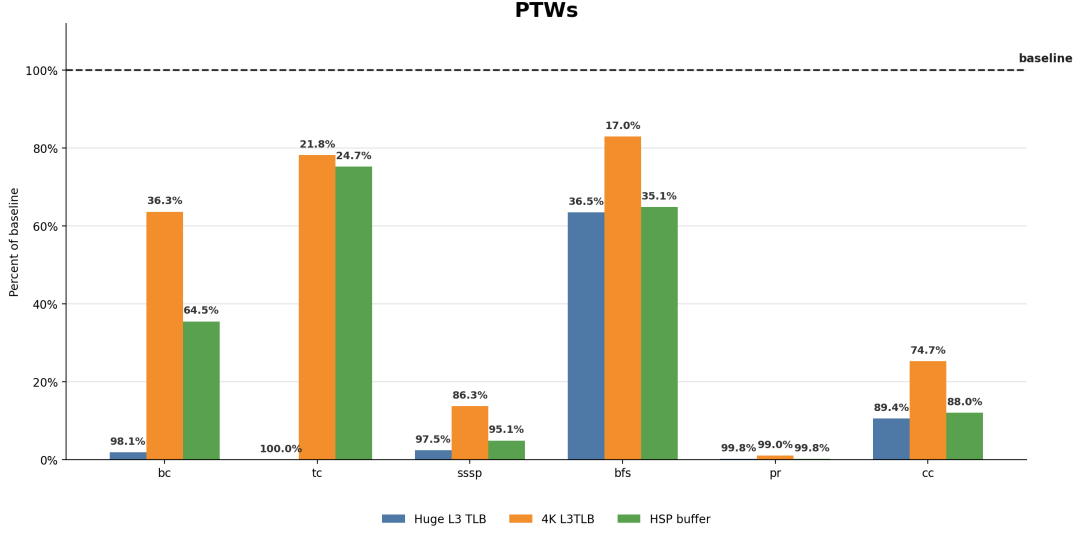


Figure 5.1: Reduction in PTWs relative to the baseline STL configuration across GAPBS workloads. The 4K-entry L3 TLB and the 4K-entry HSP buffer provide equal additional translation capacity, whereas the huge L3 TLB is included as an optimistic high-capacity reference. For these results, the HSP buffer does not use promotion and applies counter halving every 10 million cycles. The same HSP policy parameters are used for all workloads.

critical path. Therefore, these results should be viewed as conservative estimates of the HSP buffer’s performance potential.

Overall, the huge L3 TLB still provides an estimated lower-bound on total cycles. For **bc**, **sssp**, **bfs**, and **cc**, the HSP buffer reduces total cycles by 12.4%, 7.0%, 0.6%, and 2.6%, respectively. For **tc**, as discussed above, the total-cycle result for the HSP buffer is more conservative. Since the difference between the 4K L3 TLB and the HSP buffer is only 0.3%, we expect the HSP buffer to achieve comparable, or potentially better, performance under a less conservative latency calculation.

5.3 Anomaly of PageRank

A possible explanation for the anomaly in **pr** is that its page accesses follow a relatively regular aging pattern. Although **pr** has a large memory footprint, only a subset of pages

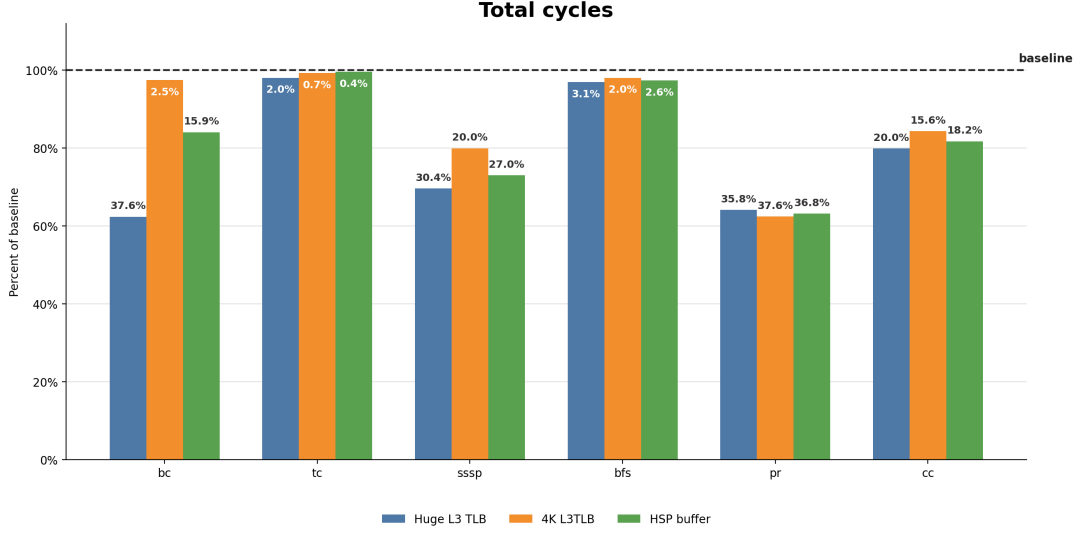


Figure 5.2: Reduction in total simulated cycles relative to the baseline STL configuration across GAPBS workloads. The 4K-entry L3 TLB and the 4K-entry HSP buffer provide equal additional translation capacity, whereas the huge L3 TLB is included as an optimistic high-capacity reference. For these results, the HSP buffer does not use promotion and applies counter halving every 10 million cycles. The same HSP policy parameters are used for all workloads.

Table 5.1: PTWs and total cycles for **pr** across different configurations.

Configuration	Total cycles	PTWs
Huge L3 TLB	128445727834	829673
4K L3 TLB	125023900597	4316786
HSP buffer	126539775092	826138

may be actively accessed within a given execution interval before being aged out. This behavior allows even a relatively small L3 TLB to capture much of the useful translation reuse. Therefore, the strong PTW reduction achieved by the 4K L3 TLB is reasonable. At the same time, as shown in Table 5.1, the HSP buffer further reduces PTWs compared with the 4K L3TLB. This result helps explain why the HSP buffer achieves a PTW count comparable to that of the huge L3 TLB.

For total cycles, the difference between the 4K L3TLB and the HSP buffer is only 0.8%. Thus, under a less conservative latency model for HSP buffer hits, we expect the HSP buffer to achieve comparable, or potentially better, performance.

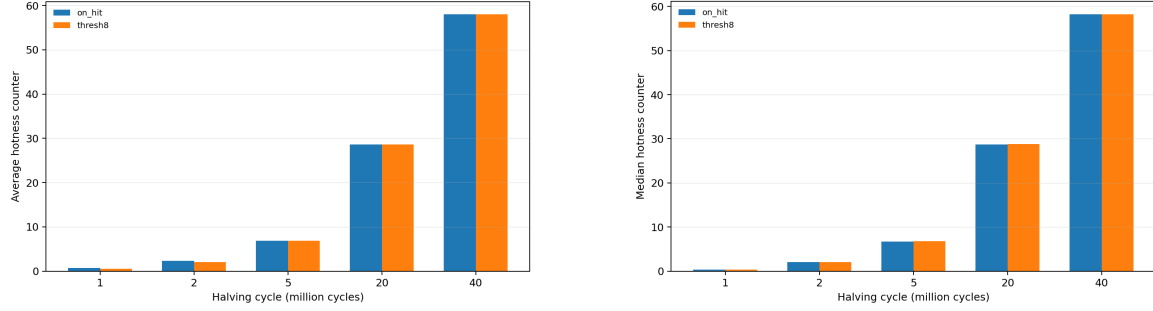


Figure 5.3: Average and median HSP hotness-counter values for **sssp**(sampled every halving cycle).

5.4 Hotness Counter Overhead

Fig. 5.3 shows the distribution of HSP-buffer hotness counters during the execution of **sssp** under two promotion policies: promotion on hit and promotion on hit with a threshold of 8. For each sampled distribution, we compute the average and median counter values, and then average these statistics across all samples. The results show that the counter values remain small (around 60) even when the halving interval is increased to 40M cycles. Therefore, the HSP buffer does not require wide counters to track translation reuse. Hence, a small counter width is sufficient in practice, which helps keep the hardware overhead of the HSP buffer low.

Chapter 6

Limitations and Future Work

6.1 Trace-Driven Simulation Limitations

Unlike full-system simulation tools such as `gem5` [19], trace-driven simulation replays a pre-collected instruction stream rather than executing the full software stack dynamically.

As a result, trace-driven simulation cannot fully capture dynamic operating-system behavior, such as page allocation, page faults, context switches, or huge-page promotion and demotion. Moreover, `ChampSim` does not support multi-threaded trace simulation. This limitation further affects how closely our results reflect real execution environments, since `GAPBS` [16] is written using `OpenMP` [20]. Here, we compiled program in `GAPBS` with single thread to adapt `ChampSim`’s single thread simulation.

6.2 Hardware Cost Modeling

A limitation of the current HSP design is that the simulator models the HSP buffer functionally rather than as a detailed circuit-level implementation. The evaluated 4K-entry fully associative structure is useful for isolating the benefit of retaining evicted

translations, but a direct monolithic implementation may be expensive in terms of area, energy, and timing. In particular, associative tag lookup, hotness-counter storage, and minimum-hotness victim selection all require additional hardware support. A practical implementation could reduce these costs by using multiple smaller HSP buffers, approximate lookup structures, or a set-associative organization. Bloom filters could be used as a low-cost membership test before performing an associative search, reducing unnecessary tag comparisons on HSP misses. The minimum-hotness replacement logic could also be pipelined or maintained in the background, since victim selection is only needed when a new translation is inserted. This thesis does not model these implementation details in detail, so the reported results should be interpreted as an architectural exploration of HSP’s potential rather than a complete hardware cost evaluation.

6.3 Future Work

There are several directions for extending this work. First, future work should develop a more realistic hardware organization for the HSP buffer. In this thesis, the HSP buffer is modeled as a 4K-entry fully associative structure to evaluate the potential benefit of retaining translations with long reuse intervals. This model is useful as an experimental starting point, but it does not fully capture the cost of implementing a large associative translation structure. A more practical design could use multiple smaller HSP buffers, each with 512 entries, corresponding to the number of 4KB pages covered by a 2MB superpage. Such an organization would better match the Hot Superpage intuition while avoiding a monolithic 4096-way associative lookup. Additional mechanisms, such as Bloom-filter-based membership tests or pipelining, should be explored to reduce lookup cost and energy overhead.

Second, future work should evaluate more HSP management policies. This thesis

primarily studies hotness-based replacement, periodic counter decay, and a promotion policy that moves useful HSP entries back into the STLB. However, there is a larger policy design space. For example, selective insertion could filter which STLB evictions are inserted into the HSP buffer, reducing pollution from translations that are unlikely to be reused. These policies may become more important as the HSP buffer capacity is reduced toward a more practical hardware implementation.

Third, future work should evaluate the HSP buffer on a broader set of workloads and input graphs. This thesis focuses on GAPBS workloads using Graph500-style Kronecker graphs. Although this setup captures important irregular graph-processing behavior, graph topology can significantly affect memory-access and translation behavior[4]. Additional inputs, such as road networks, web graphs, social-network graphs, and uniform-random graphs, would help determine which graph structures benefit most from HSP-style translation retention. Beyond graph analytics, other irregular workloads may also benefit from HSP. For example, hardware simulation tools such as Verilator [21] may be a useful candidate because hardware designs contain rich signal-dependency structures during elaboration and scheduling. Therefore, hardware simulation with Verilator may exhibit irregular memory-access behavior similar to graph workloads, making it a useful target to study.

Bibliography

- [1] S. A. V. Ghahani, S. Khadirsharbiyani, J. B. Kotra, and M. T. Kandemir, *Athena: An early-fetch architecture to reduce on-chip page walk latencies*, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, (New York, NY, USA), p. 359–371, Association for Computing Machinery, 2023.
- [2] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, *Computer Networks* **30** (1998) 107–117.
- [3] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY, *Challenges in parallel graph processing*, *Parallel Processing Letters* **17** (2007), no. 01 5–20, [<https://doi.org/10.1142/S0129626407002843>].
- [4] S. Beamer, *Understanding and Improving Graph Algorithm Performance*. PhD thesis, EECS Department, University of California, Berkeley, Sep, 2016.
- [5] A. Manocha, T. Sorensen, E. Tureci, O. Matthews, J. L. Aragón, and M. Martonosi, *GraphAttack: Optimizing Data Supply for Graph Applications on In-Order Multicore Architectures*, *ACM Transactions on Architecture and Code Optimization* **18** (Dec., 2021) 1–26.
- [6] P. Faldu, J. Diamond, and B. Grot, *Domain-Specialized Cache Management for Graph Analytics*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (San Diego, CA, USA), pp. 234–248, IEEE, Feb., 2020.
- [7] A. Manocha, J. L. Aragón, and M. Martonosi, *Graphfire: Synergizing Fetch, Insertion, and Replacement Policies for Graph Analytics*, *IEEE Transactions on Computers* **72** (Jan., 2023) 291–304.
- [8] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, *Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, Oct., 2018.

BIBLIOGRAPHY

- [9] A. Manocha, Z. Yan, E. Tureci, J. L. Aragón, D. Nellans, and M. Martonosi, *The Implications of Page Size Management on Graph Analytics*, in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 199–214, Nov., 2022.
- [10] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, *The Championship Simulator: Architectural Simulation for Education and Competition*, Oct., 2022. arXiv:2210.14324 [cs].
- [11] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, *Translation ranger: operating system support for contiguity-aware tlbs*, in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA '19*, (New York, NY, USA), p. 698–710, Association for Computing Machinery, 2019.
- [12] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, *Supporting superpages in non-contiguous physical memory*, in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 223–234, 2015.
- [13] O. Kwon, Y. Lee, and S. Hong, *Pinning Page Structure Entries to Last-Level Cache for Fast Address Translation*, *IEEE Access* **10** (2022) 114552–114565.
- [14] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, *Prefetched address translation*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-52*, (New York, NY, USA), p. 1023–1036, Association for Computing Machinery, 2019.
- [15] T. W. Barr, A. L. Cox, and S. Rixner, *Translation caching: skip, don't walk (the page table)*, *SIGARCH Comput. Archit. News* **38** (June, 2010) 48–59.
- [16] S. Beamer, K. Asanović, and D. Patterson, *The GAP Benchmark Suite*, May, 2017. arXiv:1508.03619 [cs].
- [17] Graph500 Steering Committee, “Benchmark specification.” https://graph500.org/?page_id=12, 2017. Accessed: 2026-04-21.
- [18] Intel Corporation, “Pin - A Dynamic Binary Instrumentation Tool.” <https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html>. Accessed: 2026-05-06.
- [19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, *The gem5 simulator*, *SIGARCH Comput. Archit. News* **39** (Aug., 2011) 1–7.
- [20] “OpenMP: The openmp api specification for parallel programming.” <https://www.openmp.org/>. Accessed: 2026-05-07.

BIBLIOGRAPHY

- [21] W. Snyder, “Verilator: The fastest Verilog/SystemVerilog simulator.”
<https://www.veripool.org/verilator/>, 2026. Accessed: 2026-05-04.