Micro-Architectural Techniques to Alleviate Memory-Related Stalls for Transactional and Emerging Workloads

by

Islam Atta

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2016 by Islam Atta
Abstract

Micro-Architectural Techniques to Alleviate Memory-Related Stalls for Transactional and Emerging Workloads

Islam Atta
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2016

Recent technology advances enabled computerized services which have proliferated leading to a tremendous increase in digital data at a pace of 2.5 quintillion bytes produced daily. From individuals using internet-enabled mobile devices to pay for a taxi ride, to businesses relying on sophisticated platforms to predict consumer behavior, many such services are hosted on data centers which incorporate hundreds or thousands of computer servers. The cost of installing and more so of operating these server “farms” is significant. The performance/cost efficiency of the server machines thus dictates the costs needed to provide a desired level of support.

Unfortunately, modern server architectures are not well tailored for some emerging data center applications. Accordingly, this dissertation targets memory bound applications where memory-related stalls are a major source of execution inefficiency. Specifically, the dissertation focuses on Online Transaction Processing (OLTP) systems and on a set of emerging algorithms commonly used in the so called “Big Data” applications.

OLTP workloads are at the core of many data center applications. They are known to have large instruction footprints that foil existing first-level instruction (L1-I) caches resulting in poor overall performance. Several proposed techniques remove some instruction stalls in exchange for error-prone instrumentation to the code base, or a sharp increase in the L1-I cache unit area and power.

This dissertation presents STREX and SLICC, two programmer transparent, low cost techniques which reduce instruction cache misses significantly thereby improving the performance of OLTP workloads. Both techniques exploit repetition in the instruction reference stream within and across transactions, where a transaction prefetches the instructions for similar subsequent transactions. STREX time-multiplexes the execution of similar transactions dynamically on a single core so that instructions fetched by one transaction are reused by all other transactions executing in the system as much as possible. SLICC moves transactions among multiple cores, spreading the instruction footprint over several L1-I caches, virtually increasing the cache capacity observed by transactions. Both techniques use heuristics to dynamically detect when is the best time to switch threads. SLICC works well with high
core counts where the aggregate L1-I cache capacity is sufficient to hold the actively accessed set of instructions, however it performs sub-optimally or may hurt performance when running on fewer cores. Since SLICC outperforms STREX when enough cores exist, and vice versa otherwise, this dissertation proposes a hybrid technique that combines STREX and SLICC, thereby guaranteeing maximum benefits regardless of the number of available cores and the workload’s footprint. For a 16-core system, evaluation shows that SLICC and STREX respectively reduce instruction misses by 64% and 37%, resulting in overall performance gains of 67% and 49%, and energy reductions of 26% and 20%, on average.

Big Data applications have emerged to make sense of and to extract value from the digital data deluge. As these applications operate on large volumes of semi-structured data, they exhibit intricate irregular, non-repetitive memory access patterns, exacerbating the effect of the much slower main memory. Worse, their irregular access streams tend to be hard to predict stressing existing data prefetching mechanisms.

This dissertation revisits precomputation prefetching targeting long access latency loads as a way to handle access patterns that are hard to predict. It presents Ekivolos, a precomputation prefetcher system that automatically builds prefetching slices that contain enough control flow and memory dependence instructions to faithfully and autonomously recreate the program’s access behavior without incurring monitoring and execution overheads at the main thread. Ekivolos departs from the traditional notion of creating optimized short precomputation slices, and in contrast focuses on accuracy showing that even longer slices can run ahead of the main thread as long as they are sufficiently accurate. Ekivolos operates on arbitrary application binaries and takes advantage of the observed execution paths in creating its slices. On a set of emerging workloads Ekivolos is shown to outperform three state-of-the-art hardware prefetchers and a model of past, dynamic precomputation-based prefetchers.
Acknowledgements

All praise and thanks are for Allah, the One who, by His blessing and favor, good works are accomplished. Abu Huraira reported: The Prophet, peace and blessings be upon Him, said, “He has not thanked Allah who has not thanked people.”

Many people have helped and supported me along this journey, whether directly or indirectly. First, and foremost, I would like to thank my supervisor Prof. Andreas Moshovos. Andreas influenced most of the technical and non-technical skills I gained throughout my studies. Specifically, he helped improve my analytical problem solving, writing, and presentation skills. Above all, he was always open minded, never forced a decision, and enriched my experience by giving the full lead and responsibility of my research. On a different note, he valued my personal life and was always patient whenever I had to take time off to support my small family.

Professors, colleagues and collaborators played an important role in my success. I thank my committee members for their early feedback on the research and thesis; University of Toronto Professors Tarek Abdelrahman, Natalie Enright-Jerger, Nick Koudas and Michael Stumm, and my external appraiser Prof. Sandhya Dwarkadas from University of Rochester. Special thanks goes to the late Prof. John Gregory Steffan; he was an incredible teacher and advisor. I would like to acknowledge the continuous support and feedback of my wonderful colleagues within the AENAO research group. I met and worked with the following individuals: Jorge Albericio, Kaveh Asaraai, Ioana Baldini, Michel ElNacouzi, Mostafa Hassan, Patrick Judd, Mahmoud Khallaf-Allah, Goran Narancic, Myrto Papadopoulou, Xin Tong, Di Wu, Vitaly Zakharenko and Jason Zebchuk. I thank my collaborators at École polytechnique fédérale de Lausanne (EPFL), Anastasia Ailamaki and Pınar Tözün, and at IBM Research, Ioana Baldini and Viji Srinivasan, for their expert technical advice. My internships at IBM Research and Amazon Web Services enriched my PhD experience, and helped me land on my first job after graduation; I would like to thank all those who were involved.

During my research, I used a number of open-source tools developed and maintained by other institutions. Specifically, I would like to give credit for the following tools: the Zesto processor simulator (Georgia Tech), the Shore-MT storage manager (EPFL), the ESESC processor simulator (University of California, Santa Cruz), the McPAT processor power and area model (University of Notre Dame, HP Labs, Seoul National University, and University of California, San Deigo), and the CACTI cache and memory model (HP Labs).

Administrators at the University of Toronto were always there to help. Special thanks goes to Kelly Chan, Jayne Leake, Judith Levene, and Darlene Gorzo.

My research has been funded from multiple sources including an NSERC Discovery Grant, an NSERC Discovery Accelerator Supplement, an NSERC CRD with IBM, the Electrical and Computer Engineering Fellowship and Rogers Scholarship from University of Toronto.

Finally, without the ongoing support and patience of my wife, Nour, I may not have made it thus far. She was there on day zero, and remained steady throughout the days and nights when I was absent and when I was busy. I ask her and my children, Mohamed and Sara, to forgive any negligence they may have encountered.

I dedicate this work to my father and the memory of my mother. They are the main reason I am here, they brought me up, they inspired me to go down this path, and they supported me by all means.
# Contents

1 Introduction  
1.1 Transactional Workloads – Online Transaction Processing  
1.2 “Big Data” Workloads – Emerging Data-Intensive Algorithms  
1.3 Dissertation Contributions  
1.4 Thesis Organization  

2 Background  
2.1 Cache Design  
2.1.1 Associativity  
2.1.2 Compulsory, Conflict and Capacity Misses  
2.1.3 Replacement and Insertion Policies  
2.2 Cache Prefetching  
2.2.1 Data Cache Prefetchers  
2.2.2 Instruction Cache Prefetchers  

3 OLTP Workload Characteristics  
3.1 Architectural Analysis  
3.1.1 Methodology  
3.1.2 OLTP Instructions and Data Misses  
3.1.3 Replacement Policies  
3.2 Understanding the Application Internals  
3.2.1 Redundancy Across Threads  
3.2.2 OLTP Code Structure  
3.2.3 Inter-Transaction Temporal Overlap  
3.3 Concluding Remarks  

4 SLICC: Exploiting the Aggregate Cache Capacity on CMPs  
4.1 Thread Migration: Exploiting Multiple Caches on a CMP  
4.1.1 Example Scenario  
4.1.2 SLICC Requirements  
4.1.3 Effect on Data Misses  
4.2 SLICC Design  
4.2.1 Transaction-Type-Oblivious SLICC  
4.2.2 Implementation Requirements
4.2.3 Exploiting Transaction Type Information ............................................. 27
4.2.4 Hardware to Manage Threads ............................................................. 29
4.2.5 Support for Thread Migration ............................................................. 29
4.2.6 Limitations ......................................................................................... 29
4.3 Evaluation .............................................................................................. 30
  4.3.1 Methodology ..................................................................................... 30
  4.3.2 Exploring SLICC’s Parameter Space .................................................. 31
  4.3.3 Cache Signature Accuracy ................................................................. 32
  4.3.4 Instruction Miss Change .................................................................... 32
  4.3.5 Data Miss Change ............................................................................. 34
  4.3.6 Performance ...................................................................................... 35
  4.3.7 Energy ............................................................................................... 36
  4.3.8 Hardware Cost ................................................................................. 36
  4.3.9 Remote Cache Segment Search Activity ........................................... 37
4.4 Analysis of Transaction Latency ............................................................... 37
  4.4.1 Performance of Individual Transactions .......................................... 38
  4.4.2 SLICC-Friendly Transactions ........................................................... 39
  4.4.3 SLICC-Fr Transaction Scheduler ....................................................... 40
4.5 Related Work ......................................................................................... 42
  4.5.1 Synchronized Transactions through Explicit Processor Scheduling .... 42
  4.5.2 Computation Spreading .................................................................... 42
  4.5.3 Advanced Instruction Chasing for Transactions ............................... 42
  4.5.4 Data-Oriented Transaction Execution ............................................ 43
  4.5.5 Other Thread Migration Techniques ............................................... 43
4.6 Concluding Remarks ............................................................................. 43
5 STREX: Stratifying Transaction Execution ................................................. 45
  5.1 Exploiting Transaction Instruction Overlap ......................................... 46
  5.2 STREX .................................................................................................. 48
    5.2.1 Optimal Synchronization for Identical Transactions ..................... 48
    5.2.2 STREX Synchronization Algorithm ................................................ 50
    5.2.3 Implementation ............................................................................... 50
    5.2.4 Discussion ....................................................................................... 51
  5.3 Evaluation .............................................................................................. 52
    5.3.1 Methodology .................................................................................... 53
    5.3.2 L1 Miss Rate ................................................................................... 53
    5.3.3 Throughput ..................................................................................... 54
    5.3.4 Energy ............................................................................................. 55
    5.3.5 Transaction Throughput vs. Latency .............................................. 55
    5.3.6 Hardware Cost ............................................................................... 56
    5.3.7 Replacement Policies ...................................................................... 57
  5.4 Combining STREX and SLICC ............................................................... 58
    5.4.1 Evaluation of the Hybrid Solution ................................................ 59
  5.5 Related Work ......................................................................................... 59
# EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

6.1 Motivating Example
6.1.1 Precomputation Prefetching to the Rescue
6.2 Ekivolos Prefetch System
6.2.1 P-slice Construction Process
6.2.2 Overall Architecture and Execution
6.2.3 Constructing P-slices Dynamically
6.2.4 When and How Can the P-slice Run Ahead
6.2.5 Limitations
6.3 Workloads
6.4 Methodology
6.4.1 Experimental Setup
6.4.2 Evaluated Prefetchers
6.5 Evaluation
6.5.1 Effect on Misses
6.5.2 Performance
6.5.3 Sensitivity to SpVM Sparseness
6.5.4 Energy
6.5.5 P-Slice Characteristics
6.5.6 Prefetch Core
6.6 Related Work
6.7 Concluding Remarks

# Summary

7.1 Contributions
7.2 Final Thoughts and Directions for Future Work

Bibliography
List of Tables

3.1 Workloads. ................................................................. 13
3.2 System parameters. ..................................................... 14

4.1 SLICC’s hardware component storage costs. ............... 37

5.1 STREX’s hardware component storage costs. ............... 57
5.2 FPTable: instruction footprint size per transaction in L1-I size units. .... 58
5.3 Extra hardware cost for the hybrid STREX/SLICC system. .... 59

6.1 Workloads and their access patterns. ......................... 73
6.2 System parameters. ..................................................... 74
6.3 Workload inputs and execution samples. ...................... 76
6.4 P-Slice and DLI statistics. .......................................... 81
6.5 Total instruction counts (all workloads). ...................... 82
# List of Figures

2.1 Intel Nehalem cache hierarchy: higher cache levels are smaller in size and faster to access. 7

3.1 L1 instruction and data misses, and relative performance as a function of cache size. . . . 15
3.2 L1-I MPKI with different cache replacement policies. ................................. 16
3.3 Breakdown of accesses according to instruction block reuse. OLTP transactions incur significant overlap, which is even more with same-type transactions. ................. 17
3.4 Transaction Flow Graphs for TPC-C transactions Payment and New Order, tagged with the instruction footprint size; R: Lookup, U: Update, I: Insert, IT: Index Scan. .... 18
3.5 Temporal Overlap Analysis for transactions Payment and New Order. ................. 20

4.1 Thread migration common code segment reuse example. Left: T1-3 and T4-5 are threads running similar transactions. Right: 8-core system. The shaded area is the cache activity (thick border = warm-up phase). .................................................. 23
4.2 Thread Migration algorithm. ................................................................. 26
4.3 SLICC architecture. ............................................................... 27
4.4 MPKI and relative performance as a function of fill-up\_t and matched\_t thresholds. 31
4.5 MPKI and relative performance as a function of \( \text{dilation}\_t \) (\( \text{fill-up}\_t = 256, \text{matched}\_t = 4 \)). 32
4.6 Accuracy of the partial-address bloom filter as a function of size, used as an approximate signature for the L1 instruction cache contents. .......................... 33
4.7 Instructions and data MPKI for the Baseline and various SLICC variants. ............. 33
4.8 Performance of various SLICC variants and a model for PIF, normalized to the no-prefetching Baseline. ................................................................. 35
4.9 Power, Energy and Energy-Delay products of various SLICC variants normalized to the Baseline. ................................................................. 36
4.10 Distribution of transaction latencies for the Baseline and SLICC. X-axis displays latency in million CPU cycles. Y-axis displays the total transaction population. The average latencies are shown in parentheses. 38
4.11 Average latency per transaction type for the Baseline and SLICC. SLICC’s latency is broken down into active execution time and migration-related overheads. 39
4.12 Instruction (left) and data (right) misses per transaction type, for the Baseline and SLICC. 39
4.13 L1 data misses for Baseline, Baseline-grouping and SLICC-migration-only. ........... 40
4.14 Comparing SLICC-Fr Transaction Scheduling to the Baseline and SLICC. ............ 41
4.15 Distribution of Transaction Latency with SLICC-Fr. ................................. 41
5.1 Scheduling three identical transactions. Each transaction executes in order code segments A, B and C. (a) Conventional Execution: each transaction executes until completion. (b) STREX: transactions are multiplexed on the same core. (c) SLICC: code segments are distributed among multiple cores, and transactions migrate to the cores holding the code segment to be executed.

5.2 I-MPKI reduction with STREX and identical transactions.

5.3 Effect of STREX on L1 instruction and data misses, for varying core counts.

5.4 Throughput of STREX compared to state-of-the-art instruction miss reduction techniques.

5.5 Power, Energy and Energy-Delay products for SLICC and STREX.

5.6 Transaction latency distribution (left) and average latency (right) as a function of team_size.

5.7 Overall throughput for a range of team_size values.

5.8 State-of-the-art replacement policies.

6.1 SpVM simplified code snippet, p-traces and p-slices. The target DLI is in bold.

6.2 Sparse Vector-Matrix multiplication in CSR format. Vector $V$ is represented by two arrays: indexes ($V_i$) and values ($V_v$). Matrix $M$ is represented by three arrays: indexes ($M_i$), values ($M_v$), and row beginnings ($M_b$).

6.3 A sketch displaying the relation between the program’s main thread (dark), and concurrently running p-slices. Accurate slower p-slices (blue) can be more effective than faster inaccurate p-slices (red), if the latter is frequently restarted. $\theta$ represents acceleration.

6.4 EK’s p-slice construction steps.

6.5 Taxonomy of DLI dependency vs. and p-slice overlaps.

6.6 A sketch of the p-slice execution with respect to the main thread.

6.7 Absolute L3 MPKI.

6.8 Normalized L1, L2 and L3 MPKI: EK can reduce misses significantly for all different access patterns.

6.9 Overall speedup normalized to a no-prefetching Baseline.

6.10 Access latency (excluding queuing delays) w.r.t. matrix density. Speedup is normalized to the $10^{-6}$ baseline.

6.11 Overall Energy and Energy/Delay metrics normalized to a no-prefetching Baseline.

6.12 Delinquent load prefetch coverage.
Chapter 1

Introduction

We currently live in a “computer-driven” world; computers are directly impacting the daily lives of individuals, the business processes within corporations, and even climate change. Software applications are being used to make decisions, interact socially and professionally, teach and learn, capture special moments, trade, travel, invest, diagnose diseases, transfer money, secure buildings, and enjoy sports, to name a few. The efficiency and throughput at which these software services operate is critical. At the back-end of most such services, software applications run atop modern server systems. Unfortunately, conventional server architectures are not well suited to run some of these applications.

Given the unprecedented increase in digital data [62, 150] and sophistication of the offered software services [6, 47, 63, 106, 145], many of these applications are memory-bound. Such applications exacerbate the effect of the slower memory hierarchy, a critical performance and power consideration in modern architectures, thus impacting the system’s efficiency and throughput. Accordingly, this dissertation aims at alleviating memory-related stalls\(^1\) for some data center memory-bound workloads. Specifically, this dissertation targets two emerging application domains: transactional-based systems and some of the “Big Data” applications.

Online Transaction Processing (OLTP) is a multi-billion dollar industry that increases by 10% annually [45]. OLTP refers to a class of systems that facilitate transaction-oriented applications, e.g., banking, online retail, warehouse management, and brokerage markets. OLTP needs have been driving innovations by both database management system and hardware vendors, and OLTP performance has been a major metric of comparison across vendors [65, 148, 149]. Unfortunately, modern servers are not tailored well for the characteristics of OLTP applications [36]. Literature shows that OLTP workloads are memory bound; memory access stalls account for 80% of execution time, most of which are due to first-level instruction cache misses [36, 80, 144]. Accordingly, this dissertation proposes techniques which focus on reducing instruction fetch stalls.

“Big Data”\(^2\) is a class of applications built to operate on the ever increasing digital data [157]. The volume of digital data increases at a fast pace with 2.5 quintillion bytes produced daily [62]. Such data

\(^1\) Memory stall: the processor is stalled while waiting on a memory load or store. Typically this happens on a cache miss.

\(^2\) The term “Big Data” is a broad term that refers to data sets that are so large and complex that traditional data processing applications, e.g., database management systems, are inadequate. Challenges include extracting value from large amounts of unstructured data. However, the term often refers simply to the use of predictive analytics or other certain advanced methods to extract value from data, and seldom to a particular size of data set. In this thesis, we use the term in its broader context where the data set size is relatively “big”.

1
Chapter 1. Introduction

is increasingly semi-structured or unstructured exhibiting intricate patterns. “Big Data” applications capture, manage, and analyze extensively large amounts of data to derive useful insights. Data analysis is intended to identify correlations and thus spot business trends, determine quality of research, prevent diseases, link legal citations, combat crime, determine real-time roadway traffic conditions, and more. As these applications operate on large volumes of semi-structured data they exacerbate the effect of the much slower main memory.

This dissertation aims to improve data center efficiency by proposing micro-architectural techniques to increase throughput and/or reduce energy, for the application domains mentioned previously. Current data centers are built around general-purpose processors, which were originally optimized for conventional desktop applications [134, 137, 152]. However, OLTP and Big Data workloads have relatively large data working sets which they process with complex operations. As a result, they exhibit large instruction and/or data footprints\(^3\) which in turn stress two weak components of conventional architectures: (a) the small first level instruction caches (L1-I), and (b) the slow, and limited-bandwidth off-chip memory. For OLTP workloads, this thesis presents SLICC [11] and STREX [12], two hardware transaction scheduling techniques which improve L1 instruction cache locality. SLICC and STREX exploit opportunities in the application behavior, where transactions co-operate by reusing common instruction cache blocks. For Big Data applications, this thesis presents Ekivolos [10], a precomputation-based prefetcher which addresses the intricate data access patterns incurred by some algorithms typically found in such applications. Ekivolos manages to effectively prefetch access patterns which foil existing state-of-the-art hardware and precomputation-based prefetching techniques.

The rest of this chapter highlights, for the applications studied, the memory behavior relevant to this work and the techniques proposed to alleviate memory-related stalls, followed by the thesis organization.

1.1 Transactional Workloads – Online Transaction Processing

OLTP applications have instruction footprints that are several times larger than typical L1 instruction caches\(^4\) (L1-I) [36, 80]. This disparity causes cache thrashing and results in a high L1 instruction miss rate (e.g., up to 140 misses per kilo instructions [36]). This is also true for other data center applications, however OLTP workloads exhibit the most instruction misses [36]. Fortunately, the instruction working set of such workloads typically fits on-chip, thus instruction misses are mostly served from lower cache levels. However, deep cache hierarchies, long on-chip interconnects, and the relatively high access latencies of lower cache levels create a wide gap between the L1 hit and miss latencies. Hence, L1-I misses hurt performance significantly.

In more detail, OLTP workloads are primarily composed of transactions which are typically assigned randomly to worker threads, each of which usually runs on one core of a modern multi-core system. The instruction footprint of a typical transaction does not fit into a single L1-I cache, thus thrashing the cache and incurring a high instruction miss rate. Fortunately, as corroborated by our experimental results, OLTP workloads exhibit a high-degree of instruction reuse both within a transaction and across concurrently running transactions [25, 55]. The techniques presented in this thesis exploit this behavior.

\(^3\)A program’s memory footprint refers to the amount of main memory that a program uses or references while running. The instruction footprint portion refers to the program’s code segment. The data footprint portion refers to all the data being referenced or manipulated such as arrays, graphs, hash tables, etc.

\(^4\)L1 instruction caches are not expected to grow soon; today’s technology and CPU clock cycle constraints prevent deploying L1-I caches larger than 32KB on high-end processors [2, 52, 130].
This dissertation proposes two techniques, STREX [12] and SLICC [11], to improve the instruction cache locality of OLTP by co-scheduling transactions that access similar code segments. The techniques dynamically divide the instruction footprint into smaller code segments, such that each segment fits in a single L1-I cache. Accesses to these segments can be either time-multiplexed on a single cache, or spread across multiple caches of a chip multiprocessor (CMP) and pipelined. STREX groups transactions that are expected to follow similar code paths together, then, it time-multiplexes these transactions on a single core, such that one transaction fetches a code segment that is reused by the rest of the group. After all transactions within a group take turn executing a code segment, a new code segment is fetched and the process repeats until all transactions in the group complete execution. Alternatively, SLICC spreads these segments over multiple cores, so that each L1-I cache holds part of the instruction footprint. A transaction migrates among cores such that it is dynamically assigned to the core which holds the code segment it needs. Both techniques avoid the cache thrashing that would be observed by an individual transaction that tries to fetch the whole instruction footprint on a single L1-I cache. When the instruction footprint fits in the aggregate cache capacity available for the application at runtime, SLICC works better than STREX.

STREX and SLICC are hardware solutions, which avoid undesirable program instrumentation, utilize available core and cache capacity resources, cover user as well as system-level code, and require no changes to the existing user code or software system\textsuperscript{5}. Previous OLTP instruction miss reduction techniques differ in at least one of these characteristics [25, 37, 55, 70, 115] (reviewed in more detail in Sections 4.5 and 5.5). STREX and SLICC incur overheads due to thread migrations and context switching, which must be amortized to improve performance. Thus they rely on hardware thread migration and context switching mechanisms to provide a programmer transparent solution that has low overheads [32, 124, 125].

For a 16-core system, evaluation shows that SLICC and STREX respectively reduce instruction misses by 64% and 37%, resulting in overall performance gains of 67% and 49%, and energy reductions of 26% and 20%, on average. SLICC outperforms STREX when the runtime available core count, and hence the aggregate L1-I cache capacity, is sufficient. Vice versa, STREX outperforms SLICC by an average of 49%, for two, four and eight cores, when the instruction footprint size exceeds the available aggregate cache capacity. A hybrid solution which dynamically selects between STREX and SLICC is also presented. Evaluation shows that this hybrid solution manages to select the best-performing technique given the available runtime resources.

1.2 “Big Data” Workloads – Emerging Data-Intensive Algorithms

“Big Data” applications use algorithms from several domains such as data mining, machine learning, scale-free graphs, linear algebra, mathematical programming, and financial analysis. As these applications operate on large volumes of semi-structured data they exacerbate the effect of the much slower main memory. Most of these algorithms exhibit irregular, non-repetitive data access patterns generated by traversals of massive data structures, such as graphs and sparse matrices. Worse, their irregular access streams tend to be hard to predict stressing existing hardware data prefetching mechanisms.

A precomputation-based prefetcher (PbP) can in principle predict any arbitrary access pattern [8,\textsuperscript{6}]

\textsuperscript{5}On systems where transactions do not initiate a new thread per transaction, STREX requires minimal modifications to the system interface to be able to identify the beginning of a transaction. More on this in Chapter 5.
9, 26, 42, 82, 93, 117, 162, 164]. A PbP executes as a separate thread a *precomputation slice* (p-slice), a skeleton of the program that ideally is sufficient to generate the addresses for the most *delinquent loads*, and do so early enough. For the most part, past PbPs favor p-slices with much fewer instructions than the corresponding main program slice. Using shorter p-slices was viewed as the key to allowing p-slices to run sufficiently ahead of the main program. Unfortunately, such p-slices sacrifice control flow and memory dependencies, and thus accuracy, triggering frequent aborts and restarts. As this dissertation demonstrates, some of the algorithms used by “Big Data” applications, which are increasing in importance, elude PbPs that are based on prior proposals.

This dissertation presents Ekivolos [10], EK for short, a PbP which advocates a “slow and steady wins the race” approach showing that p-slices that are not much shorter than the main thread are able to run ahead in large datasets as long as the p-slice accurately predicts the relevant control flow. EK constructs highly accurate p-slices using just the application’s binary via a lightweight, trace-based approach. EK uses a two-phase, runtime system-supported approach. First, the application is profiled offline capturing instruction traces leading to *delinquent loads*; that is, problematic load instructions which incur a significant portion of the memory stall cycles. Such traces are merged using a simple novel algorithm to construct control-flow and memory-dependence inclusive p-slices. Second, during online execution, p-slices run concurrent to the main thread generating accurate and timely prefetches improving performance and energy.

EK proves effective on various data access patterns representing emerging data-intensive workloads, and its prefetches were found to be 100% accurate for the evaluated workloads (a PbP that constructs single-path p-slices similar to past work [26, 162] is only 34% accurate, on average). EK outperforms state-of-the-art hardware prefetching systems for irregular, non-repetitive data access patterns.

### 1.3 Dissertation Contributions

This dissertation makes the following main contributions:

- It characterizes the memory behavior of OLTP workloads demonstrating significant code overlap among concurrently executing transactions, highlighting opportunities to improve instruction cache access behavior. It then proposes SLICC [11], a hardware dynamic thread migration algorithm which partitions the relatively large instruction footprint into smaller code segments, and spreads them on multiple cores within a CMP, while dynamically migrating threads to the cores which hold the relevant code segment. SLICC reduces instruction misses by 64%, improving performance by 67% and reducing energy by 26%, on average, over the baseline; a system with no effort to improve cache performance. SLICC outperforms a model of a state-of-the-art instruction prefetcher by up to 27%. However, SLICC’s benefits are limited to scenarios when the available aggregate L1-I cache capacity fits the instruction footprint.

- This dissertation proposes STREX [12], a hardware thread scheduling technique which groups and synchronizes the execution of similar transactions on a single core. On average, STREX reduces instruction and data misses by 37% and 15%, respectively, resulting in 41–55% performance gains for systems with 2–16 cores, over the baseline. STREX outperforms SLICC when the available aggregate cache capacity does not fit the transactions’ instruction footprints.
Chapter 1. Introduction

• This dissertation studies a set of algorithms commonly used in “Big Data” applications demonstrating hard-to-predict access patterns which foil state-of-the-art prefetching techniques. The dissertation revisits precomputation-based prefetching as a solution to such access patterns advocating that precomputation slices do not have to be much shorter than the main program to be effective, and demonstrating that including just enough control flow in the p-slices results in accurate and timely prefetching requests for workloads that past single-path p-slices are ineffective. It then proposes Ekivolos [10], a precomputation-based prefetcher which proves effective on hard-to-predict access patterns by constructing control-flow and data dependence inclusive precomputation slices. Ekivolos reduces L2 and L3 cache misses by 67% and 70%, on average, over a no-prefetching baseline, outperforming state-of-the-art prefetching techniques and a model of past single-path precomputation-based prefetchers.

1.4 Thesis Organization

The remaining of this thesis is organized as follows.

• Chapter 2 provides an overview of basic cache design concepts essential to understand the material in this thesis. In addition, it reviews prior art on data and instruction cache prefetching.

• Chapter 3 analyzes in detail the memory access characteristics of OLTP workloads. First, it illustrates experimentally the instruction and data cache access behavior. Then, it presents an overview of how OLTP transactions operate internally at the code level, highlighting potential opportunities to improve cache access behavior.

• Chapter 4 presents SLICC, a hardware transaction scheduling technique which improves performance by improving instruction cache locality. SLICC exploits the aggregate instruction cache capacity on modern CMPs to virtually increase the cache capacity observed by individual transactions.

• Chapter 5 presents STREX, another hardware transaction scheduling technique to improve instruction cache locality. However, as opposed to SLICC, STREX operates only on a single core. STREX improves locality by dynamically pipelining and coordinating the execution of similar transactions so as to increase the instruction cache block reuse.

• Chapter 6 first identifies and studies a set of emerging data-intensive algorithms commonly used in “Big Data” applications. The data access patterns for some of these algorithms are shown to be hard-to-predict. Then, the chapter presents Ekivolos, a precomputation-based prefetcher which effectively addresses such access patterns.

• Finally, Chapter 7 summarizes the contributions in this thesis, and discusses directions for future work.
Chapter 2

Background

This dissertation attempts to study a number of crucial applications which foil cache hierarchies on modern processors as well as state-of-the-art prefetching techniques. Hence, it is essential to review relevant cache design considerations, and prefetching techniques.

In an attempt to mitigate the so called “memory-wall” [160], modern high-end processors employ multi-level cache hierarchies, and aggressive prefetching systems [72, 85]. However, some existing and emerging classes of applications foil such architectures. These applications may be classified as instruction- and/or data-intensive. Instruction-intensive applications exhibit an instruction footprint size (100s of KBs) that is an order of magnitude larger than the relatively small level-one instruction caches (32KB is common on systems existing today), causing ongoing cache thrashing, hurting performance. OLTP workloads are instruction-intensive. On the other hand, some emerging data-intensive applications do not only have huge data footprints (GBs to TBs) that do not fit on chip, but also exhibit intricate access patterns that elude state-of-the-art prefetching systems. Some big data applications are considered as data-intensive. Section 2.1 reviews relevant cache design concepts to maximize cache utility. Section 2.2 reviews instruction- and data-prefetching systems which try to hide the memory access latency.

2.1 Cache Design

Since this dissertation focuses on improving cache access behavior for certain classes of applications, it is essential to describe some of the relevant cache design considerations on modern architectures. Patterson and Hennessy [112] and Heuring et al. [58] present a comprehensive discussion on cache design principles. This section provides a brief overview and reviews recent work on only the topics closely related to the material presented in this dissertation.

Caches are relatively small on-chip memory storage arrays that are faster than the main memory and are placed closer to the CPU\(^1\), thus incurring much lower access latency. Caches are intended to bridge the access latency gap between the fast CPU, and the much slower main memory subsystem. On a CPU memory request, data is fetched from the main memory and a copy is stored in the cache. Typically a cache holds fixed size chunks of contiguous data called cache blocks (e.g., 64B). Ideally, a cache block that

---

\(^1\)The Central Processing Unit (CPU) is the circuitry which fetches, decodes and executes the instructions of a program by performing the arithmetic, logical, control and input/output operations specified by the instructions.
Figure 2.1: Intel Nehalem cache hierarchy: higher cache levels are smaller in size and faster to access.

is inserted in the cache will get reused multiple times, saving memory access latency, thus improving the overall performance. In the common case, applications exhibit spatial and/or temporal locality. Spatial locality entails two or more references to data items that happen to reside near each other in terms of address space, hence potentially residing in the same cache block. Temporal locality entails multiple references that are close in time to the same cache block. Both types of locality generally render the caches useful.

Modern architectures employ multiple levels of cache hierarchy [2, 72]. Higher levels of cache (e.g., first-level or L1) are smaller in size, yet faster to access and closer to the CPU. Vice versa, lower cache levels are slower, placed farther away from the CPU, and have larger capacity. On multi-core systems, caches can be either private (its contents are fully owned by a single core), or shared (its contents are shared among multiple cores). Typically, higher cache levels are private (e.g., L1 or L2) and lower levels are shared (e.g., L3). Furthermore, the private L1 cache is usually divided into two separate instruction (L1-I) and data caches (L1-D), for several reasons including: allow simultaneous accesses to instructions and data, simplify the circuitry for instruction caches which are read-only, and prevent the typically much larger data working set from overwhelming the instruction working set. For example, Figure 2.1 depicts the Intel Nehalem cache hierarchy which incorporates three levels of cache, private separate L1 instruction and data caches, private unified\(^2\) L2 caches, and a shared L3 cache [2, 129, 139].

A number of design considerations govern cache management with the ultimate goal of reducing cache misses. Caches are associative structures divided up into one or more sets. Given the address of a cache block, indexing determines which specific set(s) can hold that cache block. When inserting a new cache block, insertion policies determine the insertion location within the corresponding set, and replacement policies determine which cache block gets evicted from the set, if necessary. The following sections briefly describe cache associativity, illustrate different types of cache misses, and review relevant prior art on replacement and insertion policies.

### 2.1.1 Associativity

In general, caches fall into one of three categories: (1) direct-mapped, (2) N-way set associative, or (3) fully associative [51]. In direct-mapped caches, a cache block can only be inserted in one spot in the cache (1-to-1 mapping). Searching for a cache block is easy, but it is not very flexible about where to

\(^2\)A unified cache holds both instruction and data cache blocks.
put the blocks. For example, two blocks that map to the same location can never coexist, even if both 
blocks are accessed frequently. \textbf{N-way set associative} caches are divided into a number of \textit{sets} such 
that each set can hold up to \(N\) cache blocks. Thus on a cache lookup, up to \(N\) cache blocks must be 
searched to determine whether the requested block resides in the cache or not. However, set associative 
caches are more flexible than direct-mapped caches when it comes to where to insert new blocks (1-to-\(N\) 
mapping). Finally, \textbf{fully associative} caches are \textit{N-way} set associative caches with a single set, i.e., \(N\) 
is equal to the total number of cache blocks (at the other extreme, direct-mapped caches have 1-way 
sets). Fully-associative caches provide maximum flexibility for insertion, however the cost incurred by a 
cache lookup (searching the entire cache with 100s to 1000s of blocks) makes these designs impractical 
to implement. Modern systems typically implement 2-, 4- or 8-way L1 caches, and up to 32-way L3 
caches [2, 52, 130].

\section{Compulsory, Conflict and Capacity Misses}

Hill \textit{et al.} classify cache misses as one of three: \textit{compulsory}, \textit{capacity}, or \textit{conflict}, depending on the main 
cause of the cache miss [59]. If the miss is a cold miss, i.e., happens on the first access to a cache block, it 
is considered as a \textit{compulsory} miss, since it cannot be avoided (assuming no prefetching). Hence, even 
a cache that is unrealistically infinite in size would still incur compulsory misses. \textit{Capacity} misses are 
the misses to cache blocks which have been evicted prior to the cache miss due limitations on cache size. 
Thus a fully-associative cache that is larger than the working set of a program would not see capacity 
misses. Finally, \textit{conflict} misses are caused by set-associativity. Thus a fully-associative cache would not 
incur conflict misses. Knowing the type of misses which dominate a given application’s cache accesses 
may reveal insight into the application’s access pattern, and potential solutions to mitigate such misses.

\section{Replacement and Insertion Policies}

To make room for a new cache block to be inserted, often another cache block has to be evicted, the 
\textit{victim}. Replacement policies determine the criteria to select the victim block, at the granularity of a 
set. The policies try to maximize the cache reuse by retaining the cache blocks that are expected to be 
reused again in the future, and evicting blocks that are less likely to be reused.

The most popular replacement policy is the \textit{Least Recently Used}, or \textit{LRU}. LRU is based on an 
observation that subsequent accesses to the same cache block, if any, would be close in time, while cache 
blocks that have not been accessed for a while are less likely to be re-accessed again. With an LRU policy, 
the cache blocks within a set are sorted by their most recent access: when a cache block is accessed, it 
is moved to the top of the sorted stack, denoted as the \textit{Most Recently Used}, or \textit{MRU} position. LRU 
policy inserts new blocks at the MRU position, and evicts blocks from the bottom of the stack, the 
LRU position. Hence, LRU performs well with applications when blocks have short re-reference periods. 
Applications with streaming accesses exhibit the exact opposite pattern that LRU tries to exploit and 
as a result tend to thrash an LRU-managed cache.

Qureshi \textit{et al.} show that some workloads, including those that have long-term reuse, are not LRU-
friendly [115]. For such workloads, LRU cycles through a large footprint while it would be best to keep 
at least some part of the footprint cache resident. They modify LRU by introducing new static (LIP, 
BIP) and dynamic (DIP) insertion policies where the newly accessed blocks are not necessarily inserted 
at the MRU position. The Least Insertion Policy (LIP) inserts a cache block in the LRU position. LIP
Chapter 2. Background

prevents thrashing for workloads whose working set is greater than the cache size and obtains high hit rates for workloads that have a cyclic access pattern. The Bimodal Insertion Policy (BIP) inserts most blocks in the LRU position, similar to LIP, however, it infrequently inserts blocks in the MRU position, similar to the basic LRU policy. BIP tries to adapt to changes in the working set while retaining the thrashing protection of LIP. Finally, the Dynamic Insertion Policy (DIP) toggles dynamically between LRU and BIP based on which policy incurs fewer misses.

Jaleel et al. propose the SRRIP, BRRIP, and DRRIP re-reference interval based insertion policies [70]. Their Re-reference Insertion Prediction (RRIP) chain represents the order in which blocks are predicted to be re-referenced. The block at the head of the RRIP chain is predicted to have a near-immediate re-reference interval; i.e., the block is predicted to be re-referenced almost immediately. The block at the tail of the RRIP chain is predicted to have a distant re-reference interval; i.e., the block is predicted to be re-referenced far in the future. On a cache eviction, a distant block is replaced. Re-references to a block promote its position towards the head of the chain. Given the RRIP chain, the LRU policy predicts newly inserted blocks to have a short re-reference interval and thus inserts such blocks in the near-immediate position. In contrast, Static RRIP (SRRIP) predicts newly inserted blocks to have a long re-reference interval, which is an intermediate state between near-immediate and distant re-reference intervals. The goal is to prevent cache blocks with distant re-references from polluting the cache, while still allowing the policy some time to learn the application behavior. The exact position of insertion varies based on cache hits and misses. When the re-reference interval of all blocks is larger than the available cache, i.e., thrashing behavior, SRRIP causes cache thrashing and results in no cache hits. Bimodal RRIP (BRRIP), tries to avoid such scenarios by predicting a distant re-reference interval for most cache blocks, while in-frequently predicting long re-reference intervals. Analogous to DIP [115], Dynamic RRIP (DRRIP) dynamically selects between the scan-resistant SRRIP and the thrash-resistant BRRIP.

Since replacement and insertion policies focus on maximizing the reuse of what has been already brought into the cache, they do not target compulsory misses. Hence different replacement/insertion policies have almost the same effect when the misses are predominantly compulsory. Section 3.1.3 evaluates OLTP applications with the policies discussed in this section, focusing only on instruction misses (which are mostly non-compulsory). The analysis shows that OLTP transactions cycle through a relatively large instruction footprint that cannot be captured by simply changing the replacement/insertion policy.

2.2 Cache Prefetching

A memory request can be very costly in terms of performance if it is being serviced by lower cache levels or off-chip memory. Prefetching is an approach to hide the much higher access latency by anticipating what data the program will need and fetching it in advance. However, prefetching is not always effective, and can even hurt performance. The following sections review prior art on data and instruction cache prefetching. Further prior art on cache miss reduction techniques are presented in Chapters 4, 5 and 6.

2.2.1 Data Cache Prefetchers

Data prefetching has been a hot topic for decades [84, 131]. Conceptually, a prefetcher can be decoupled into two stages: (a) speculatively compute the address of the block to be prefetched, the target address, then (b) fetch that block. Ideally, the fetch would be issued ahead of the program’s original reference,
Chapter 2. Background

thus fully or partially\(^3\) hiding the memory access latency. If the speculative target address computation
was inaccurate, or if the prefetched cache block gets evicted before its original reference, the prefetched
data may be useless and pollute the caches, potentially hurting performance. Thus the accuracy and
timeliness of the target address computation is key to effective prefetching.

Numerous hardware and software prefetching techniques exist targeting various memory access pat-
terns such as linear progressions [41, 61, 108], or repeating patterns in the access streams or in the
relative distance of subsequent accesses [68, 74, 104, 132]. For example, on a cache block miss, a simple
linear next-line prefetcher will prefetch the next sequential block, if it is not resident in the cache. By
monitoring the program’s access behavior, predictor-based prefetchers try to detect the program’s ac-
cess pattern, and use this information to predict future accesses. For example, a stride prefetcher [41]
attempts to detect stride patterns \((a, a + n, a + 2n, \ldots)\) and issue prefetch requests accordingly. More
sophisticated prefetchers try to detect irregular repeating patterns [74], and using this information to
calculate addresses that may be referenced in the future. As this work corroborates, there are access
patterns that elude such prediction-based prefetchers. In particular, non-repetitive irregular access pat-
terns cannot be predicted since they neither exhibit a regular access pattern that can be detected, nor
do past accesses entail future accesses. This dissertation tackles such hard-to-predict access patterns.

This dissertation revisits precomputation-based prefetchers (PbP’s), which can in principle prefetch
any arbitrary access pattern [8, 9, 26, 42, 82, 93, 162, 164]. A PbP executes as a separate thread a
precomputation slice (p-slice), a skeleton of the program that ideally is sufficient to generate the addresses
for the most delinquent loads (DLIs) [27], and do so early enough. A delinquent load is a problematic load
instruction that causes frequent CPU stalls due to cache misses, impacting performance significantly.
Prior PbP work varied in the way the p-slice is constructed and how it is executed. Specifically, p-
slices were constructed either manually [164] or automatically using compiler techniques [82, 93, 117], or
extracted from application binaries [8, 9, 26, 42, 162]. Past work embeds the speculative precomputation
alongside the program’s instructions [117] or executes it on an auxiliary pipeline [94], a prefetch engine [8],
an SMT thread [26, 88, 162, 164], a conventional core in a CMP [9, 42], or in-memory [56]. Section 6.6
reviews past PbP techniques in more detail.

For the most part, past PbP’s favor p-slices with much fewer instructions than the corresponding
main program slice. Using shorter p-slices was viewed as the key to allowing p-slices to run sufficiently
ahead of the main program, and hence be able to hide the memory access latency of a DLI. This
dissertation revisits this notion by demonstrating that, for a set of emerging data-intensive workloads,
shorter p-slices may lack sufficient accuracy to be effective. Chapter 6 presents a detailed analysis of
our observations, and presents Ekivoulos, a PbP system that effectively tackles hard-to-predict access
patterns.

2.2.2 Instruction Cache Prefetchers

Similar to data prefetching, instruction prefetching has been studied for decades [79, 119]. Instruction
prefetchers have evolved from simple stream buffers [75, 131], to branch-direction based predictors [121,
133], to highly accurate, sophisticated address-correlation prefetchers [37, 39]. Simple linear prefetchers
exploit sequences in the instruction access stream, and prefetch instruction cache blocks with various

\(^3\)Prefetching a cache block ahead of its original reference can fully hide the memory access latency if the distance in
time between the prefetch request and the original reference is equal to or greater than the runtime access latency of that
block. Otherwise, a prefetch can at most partially hide the access latency.
lookahead depths\textsuperscript{4} [75, 131]. However, complex programs incur control-flow transfers that are beyond the reach of naïve sequential lookahead. Thus researchers proposed relying on the branch predictor to predict the program’s control flow and prefetch instructions accordingly [121, 133]. However, this approach may fail when control flow is dominated by data-dependent branches (low prediction branch accuracy). Other more sophisticated prefetchers can predict complex execution paths by recording previously seen instruction streams, and then prefetching cache blocks when a part of a recorded stream is touched again. Ferdman et al. propose prefetchers that record temporal streams of accessed instruction blocks, TIFS [39], or streams of committed instructions, PIF [37]. These prefetchers prove to be highly accurate even for workloads with extensive data-dependent control-flow. However, their bookkeeping storage overhead scales with the instruction footprint size, reaching the point of doubling the area of the L1 instruction caches. Kaynak et al. observe that for homogeneous server workloads, where concurrent threads execute similar code regions, the recorded instruction stream history will be similar, and hence can be combined [79], reducing the storage overhead significantly.

Hardavellas et al. [53] observe that more than 60% of the accesses to a distributed shared L2 cache are for instructions. They adapt a NUCA\textsuperscript{5} block placement policy according to workload categorization, and allow replication of (read-only) instructions, which shortens the distance between L1-I caches and L2s. This reduces the L1-I miss penalty, but does not reduce the miss rate.

Instead of prefetching instructions, this dissertation proposes techniques to improve instruction locality by maximizing the instruction cache block reuse. Chapters 4 and 5 demonstrate, for OLTP workloads, that the proposed techniques achieve similar or better performance when compared to a model of PIF [37], without its relatively large storage overhead.

\textsuperscript{4}The lookahead depth is the number of sequential cache blocks prefetched ahead of the most recently accessed block. E.g., on accesses to cache block with address tag $a$, and lookahead depth $n$, cache blocks $a + 1$, $a + 2$, up to $a + n$ will be prefetched.

\textsuperscript{5}Non-Uniform Cache Access: the memory access time depends on the memory location relative to the processor [81].
Chapter 3

OLTP Workload Characteristics

By studying how Online Transaction Processing (OLTP) applications operate at the application level and how they perform on modern architectures, this chapter serves as motivation to SLICC (Chapter 4) and STREX (Chapter 5), two novel hardware scheduling techniques that improve instruction locality for OLTP. First, Section 3.1 studies the instruction and data cache behavior for OLTP. Then, to understand why OLTP applications suffer from instruction misses, Section 3.2 presents an overview of how OLTP transactions operate, and explains briefly their code structure as individual transaction entities, and as a group of concurrently executing threads. This analysis reveals the following:

- OLTP transactions suffer from instruction misses and their instruction streams exhibit long-range recurring patterns leading to eviction of useful blocks that are re-referenced later.

- The instruction footprint of a typical OLTP transaction fits comfortably in the aggregate L1-I cache capacity of modern many-core chips.

- Recently proposed cache replacement policies [70, 115] reduce instruction misses by 8% on average for the best policy, but leave ample room for improvement.

- OLTP workloads exhibit a high-degree of code overlap both within a transaction and across concurrently running transactions [25, 55].

3.1 Architectural Analysis

This section examines the OLTP memory access behavior. The analysis targets TPC-C [146] and TPC-E [147], which resemble state-of-the-art commercial OLTP applications. Their behavior is contrasted with MapReduce [36], a data center workload which has a smaller instruction footprint. Results show that for OLTP transactions:

1. Most instruction misses are due to limited cache capacity, whereas most data misses are compulsory.

2. The instruction footprint of most transactions would fit in the aggregate L1 instruction cache capacity of even small-scale chip multiprocessors (eight cores). The same is not true for data footprints.

3. Existing non-LRU cache replacement policies reduce the instruction miss rate, but only by a fraction of what would be possible with larger caches.
### 3.1.1 Methodology

Table 3.1 lists the workloads used. Both TPC-C variants [146] and TPC-E [147] run on top of the scalable open-source storage manager Shore-MT [73], which was shown to have similar components and micro-architectural behavior as commercial Database Management Systems (DBMS’s) [5]. The client-driver and the database are kept on the same machine and the buffer-pool is configured to keep the whole database in memory. The experiments simulate 1K transactions for these workloads, or approximately 1.2B and 2.6B instructions for TPC-C and TPC-E, respectively. TPC-C-1 and TPC-C-10 use two different scaling factors for the TPC-C database and serve to demonstrate, in later chapters, that SLICC and STREX remain effective even when the database size, and thus the data footprint grows larger. We focus most of our evaluation on TPC-C-1 (referred as TPC-C). TPC-C and TPC-E have larger instruction and data footprints compared to other scale-out workloads [36]. The MapReduce CloudSuite workload [111], which has a relatively small instruction footprint [36] serves to demonstrate that the techniques presented in Chapters 4 and 5 (SLICC and STREX) are robust in that they do not reduce performance for workloads that do not have similar behavior to OLTP. The MapReduce workload divides the input dataset across 300 threads, each performing a single map/reduce task. For clarity, the discussion focuses on TPC-C and TPC-E with MapReduce being included only where absolutely necessary.

The experiments replay x86 execution traces, modeling the timing of all events, and maintaining the original transaction sequence. The traces for TPC-C and TPC-E include both user and kernel activity, and were collected using QTrace [141], an instrumentation extension to the QEMU full-system emulator [16]. For MapReduce, PIN [95] was used to extract execution traces. All simulations use a modified version of the Zesto x86 multi-core architecture simulator [92].

One limitation arises with this trace-based methodology: the sequence of synchronization-related events that occurred while extracting the execution traces might differ on the simulator. However, we are inclined to believe that the results reported are still valid for several reasons. First, the execution traces were extracted in single threaded mode eliminating the execution overhead due to locking contention. Second, OLTP transactions are independent, by design, and the communication between transactions is mostly in the form of locking and latching for shared database meta-data. Furthermore, Shore-MT [73] (the used data-storage engine) employs techniques to partition the accesses to database records and pages, which minimize this form of communication and the lifetime of the locks [110, 143].

Table 3.2 details the baseline architecture. With $N$ cores, the baseline architecture has $N$ hardware contexts with the OS making thread scheduling decisions. CACTI 6.5 [102] is used to model the access
Table 3.2: System parameters.

<table>
<thead>
<tr>
<th>Processing Cores</th>
<th>16 OoO cores, 2.5GHz 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private L1 Caches</td>
<td>32KB, 64B blocks, 8-way 3-cycle load-to-use, 16 MSHRs MESI-coherence for L1-D</td>
</tr>
<tr>
<td>Private L2 Caches</td>
<td>256KB, 64B blocks, 16-way 7-cycles, 32 MSHRs MESI-coherence for L1-D</td>
</tr>
<tr>
<td>L3 NUCA Cache</td>
<td>Shared, 1MB per core, 32-way 64B blocks, 16 slices 16-cycles, 64 MSHRs</td>
</tr>
<tr>
<td>Interconnect</td>
<td>2D Torus, 1-cycle hop latency</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks 8B Bus Width, Open Page Policy ( t_{CAS}^{\text{}-10} ), ( t_{RCD}^{\text{}-10} ), ( t_{RP}^{\text{}-10} ), ( t_{RAS}^{\text{}-35} ) ( t_{RC}^{\text{}-47.5} ), ( t_{WR}^{\text{}-15} ), ( t_{WTR}^{\text{}-7.5} ) ( t_{RTRS}^{\text{}-1} ), ( t_{CCD}^{\text{}-4} ), ( t_{CWD}^{\text{}-9.5} )</td>
</tr>
</tbody>
</table>

latencies of different cache sizes\(^1\). Throughput is measured as the inverse of the number of cycles required to execute all transactions. The experiments report the L1 misses per kilo instructions for instructions (I-MPKI) and data (D-MPKI).

### 3.1.2 OLTP Instructions and Data Misses

In typical multi-threaded OLTP systems, each transaction is assigned to a worker thread. Individual threads, whose memory footprints do not fit in the L1 cache, suffer from high miss rates. Ferdman et al. show that in OLTP, memory stalls account for up to 80% of the execution time [36], while Tözün et al. show that instruction stalls account for 70–85% of the overall stall cycles [144]. The relatively large instruction footprint is to be blamed, as Lo et al. report a 556KB instruction footprint for OLTP using an Oracle DBMS [91].

We measure instruction and data L1 misses. Figure 3.1 shows the number of misses per kilo-instructions (MPKI) for a range of L1 instruction (L1-I) and data (L1-D) cache sizes. To isolate the effect of varying the instruction or data cache sizes on overall performance, we first vary the L1-I cache size (16KB–512KB) while keeping the L1-D cache size at 32KB (our baseline), and then we vary the L1-D cache size while keeping the L1-I at 32KB. We note that instruction misses are not affected by varying the L1-D cache size, and vice versa. Figure 3.1 shows a breakdown of instruction and data L1 misses into three categories: capacity, conflict and compulsory [59]. By identifying where most misses come from, we highlight the reasons behind the memory stalls; is it cache size, associativity, or cold misses?

---

\(^1\)CACTI is configured to use UCA model (Uniform Cache Access), aggressive interconnect, ITRS-HP transistor type (High Performance transistors based on ITRS projections [1]), and global wires.
Chapter 3. OLTP Workload Characteristics

Figure 3.1 shows that for OLTP workloads, capacity misses dominate instruction misses. This implies that the instruction footprint does not fit in the cache and has lots of reuse; cache blocks are evicted from the cache before they are re-referenced. Hence larger L1-I caches, which can hold cache blocks for longer periods, can reduce instruction misses. To keep up with the CPU clock speeds, technology constraints have limited the sizes of L1 instruction caches to about 32KB today. With 32KB caches, instruction capacity misses are an order of magnitude more than data capacity misses. Compulsory misses, which occur for the first reference to each unique cache block, dominate data misses. Thus, larger data caches can do little to reduce data misses. For 32KB caches, compulsory data misses are an order of magnitude more than compulsory instruction misses. Unless data is prefetched, or predicted by some means [24, 89, 99], data misses cannot be reduced.

For MapReduce, 71% of the total L1 misses are compulsory for 32KB caches, hence larger L1 instructions or data caches are not as beneficial.

Figure 3.1 also shows overall performance improvement normalized to the baseline. Performance improvements with larger L1-D caches are negligible at 1%, but can be as high as 21% with larger L1-I caches (MapReduce shows less than 3% improvement). If increasing cache size did not also increase latency, performance improvements would be higher; for example, a 512KB L1-I with the latency of a 32KB L1-I would result in a 72% performance improvement for TPC-C.

We conclude that (a) OLTP transactions have instruction footprints that, while larger than typical L1-I caches, could fit in 256–512KB caches\(^2\). (b) OLTP data footprints are much larger and do not fit even on an order of magnitude larger L1-D cache. Additionally, (c) OLTP instruction streams exhibit a significant reuse over regions that exceed typical L1-I cache sizes leading to the eviction of useful blocks that are re-accessed.

3.1.3 Replacement Policies

Figure 3.2 reports the MPKI for the replacement policies described in Section 2.1.3, for the baseline 32KB L1-I cache. BRRIP and DRRIP perform best reducing misses by an average of 8% over LRU.

\(^2\)The size of the instruction footprint depends on the implementation of the underlying data storage manager. Our experiment use Shore-MT [73].
This reduction is only a fraction of what is possible with larger caches as Figure 3.1 showed.

Thrashing applications favor Bimodal RRIP (BRRIP), which predicts a distant re-reference interval for most blocks [70]. Dynamic RRIP (DRRIP) selects the best policy from Static RRIP (SRRIP) and BRRIP at runtime, using sampling. Figure 3.2 shows that DRRIP chose BRRIP most of the time. Contrary to the Least Insertion Policy (LIP), which promotes a referenced block to the MRU position, BRRIP uses access frequency to promote cache blocks gradually towards the head of the chain.

While Section 3.1.2 showed that OLTP instruction streams exhibit recurring patterns with long re-reference periods, this section shows that existing insertion/replacement policies cannot fully capture such re-reference intervals. These insights suggest that a more invasive solution is required, which may either predict future accesses (as in prefetching), or alter the sequence by which cache blocks are being accessed. The next section studies the instruction streams at the application level, demonstrating opportunities to reschedule the cache block accesses to maximize their reuse.

3.2 Understanding the Application Internals

OLTP systems are composed of transactions. A transaction is an atomic sequence of operations which, when combined, serve the same purpose. For example, a purchase transaction may involve subtracting from the available stock of the purchased item, creating a new shipping request and updating the customer purchase log. While the standard OLTP benchmarks, TPC-C [146] and TPC-E [147], include only five and twelve different transaction types, respectively, the overall system is expected execute hundreds of transactions every second. Improving the performance of OLTP is measured by throughput; how many transactions are completed per second [148, 149].

Conventional OLTP systems make no explicit effort to improve instruction reuse in processor caches. OLTP systems are multi-threaded applications and typically assign transactions to cores in an ad-hoc manner, aiming to balance the work across nodes. A transaction is assigned to a core where it executes to completion. As a result, as the length of transactions in the system increases, OLTP workloads suffer from instruction stalls due to high L1-I cache miss rates. This section demonstrates that OLTP transactions exhibit substantial code overlap, which can create significant locality across different instances of transactions. Therefore, there is opportunity to improve instruction reuse by coordinating transaction execution.
Figure 3.3: Breakdown of accesses according to instruction block reuse. OLTP transactions incur significant overlap, which is even more with same-type transactions.

Section 3.2.1 demonstrates a significant instruction overlap across concurrent OLTP threads. Section 3.2.2 takes a closer look at the code structure of transactions revealing that transactions ought to naturally exhibit significant temporal locality, with transactions of the same type exhibiting the most locality. Section 3.2.3 shows that same-type transactions do follow similar execution paths touching mostly overlapping code segments.

### 3.2.1 Redundancy Across Threads

Chakraborty et al. profile OLTP instruction accesses and report an 80% redundancy across multiple cores for user and OS code [25]. Figure 3.3 corroborates these results and further shows that 98% of the instruction cache blocks are common among threads executing the same transaction type; although similar transactions do not follow the exact same control flow path, they have common code segments at a coarser granularity. Figure 3.3 shows a breakdown of all instruction cache accesses classified according to the reuse experienced by the accessed block over the duration of the application. The figure presents three coarse reuse categories: single, few and most that correspond to blocks accessed by only one thread, at most, or more than 60% of all the threads, respectively. The cut-off value of 60% is a simple heuristic to represent the majority of transactions. This behavior highlights an opportunity to reduce instruction misses by exploiting locality across multiple threads, particularly threads of the same transaction type. For example, instead of having multiple transactions fetch a common cache block on multiple cores, it might be better to fetch this cache block once, and have multiple transactions access it on the same core.

### 3.2.2 OLTP Code Structure

This section discusses the code structure of some representative OLTP transactions. The results of this section demonstrate that at a high-level, transactions of the same type ought to follow similar instruction paths during execution touching similar code segments. The rest of this section focuses on the TPC-C [146] New Order and Payment transactions, which comprise 88% of the TPC-C workload mix, according to the benchmark’s specification. The discussion and results for the other transactions are similar, so they are omitted for brevity.

OLTP transactions are composed of actions that in turn may execute several basic functions. Basic function examples include looking up a record through an index, scanning and updating an index,
inserting a tuple to a table, updating a tuple, etc. No matter how different the output or high-level function of one transaction is from another, all database transactions are composed of a subset of the aforementioned basic functions, repeated several times for different inputs, and appearing in various permutations.

Figure 3.4 shows the action flow graphs for the Payment and New Order transactions. Boxes represent actions, which are tagged with their corresponding instruction cache footprint; the arrows show the action execution order; and rhombuses represent control flow decisions. The flow diagrams mark each action with the basic function it performs, e.g., R() is an index lookup. Each function manipulates input data in a different way; further details are immaterial to this discussion.

The diagrams suggest that there is significant overlap across transactions of the same type. All Payment transactions execute the same sequence of actions, with the exception of conditionally executing the IT(CUST) action. The instruction stream may not be identical across all Payment transactions, as the actual code path may vary depending on the input data. Nevertheless, there is overlap for data-independent code segments (i.e., code segments which are executed independent of the input data), and for data-dependent code segments that follow the same path according to similar control-flow decisions. New Order transactions exhibit significant overlap as well, despite that their code paths may diverge more over time due to the inner loop conditioned on the OL_CNT value.

The diagrams also suggest that there is overlap across transactions of different types. Initially, Payment and New Order transactions perform index lookups on the same tables: Warehouse, District, Customer. Therefore, their code paths are similar at first, and then they diverge. New Order has a
3.2.3 Inter-Transaction Temporal Overlap

In this section we aim to study the temporal behavior of concurrently executing transactions. Section 3.2.1 showed that throughout the lifetime of transactions they tend to mostly touch common code. Section 3.2.2 tried to justify this code overlap by showing that transactions ought to execute similar actions, albeit on different data, and that same-type transactions tend to execute similar action sequences, suggesting that at the granularity of actions they touch common code segments in a similar order. However, given that actions typically operate on different data per transaction, their instruction streams may diverge by time. This section quantifies this divergence, showing that same-type transactions maintain significant temporal code overlaps throughout their lifetime.

To quantify the temporal code divergence, we execute a number of transactions and try to match the instruction cache blocks they tend to touch across time. We try to design an experiment to measure the amount of temporal overlap across multiple instruction cache block traces. This is analogous to finding the longest common subsequence among multiple traces of cache blocks; an NP-Hard problem [18, 96]. Hunt and McIlroy implemented the efficient diff utility, a data comparison tool that calculates and displays the differences between two files [60]. However, this problem is more complex because (a) comparing two sequences only is statistically insignificant, (b) these sequences are expected to include lots of control flow (e.g., conditional branches and loops with different counts), and (c) their lengths are on the order of hundreds of thousands of cache blocks. Thus we cannot simply use diff, nor perform the comparison manually.

We devise an experiment to compare instruction cache block sequences, relying on a number of heuristics. Figure 3.5 depicts the degree of instruction footprint overlap over time for Payment and New Order transactions executing concurrently. This experiment uses 16 randomly chosen same-type transactions that concurrently execute on 16 cores, each with a 32KB L1-I at a rate of one instruction per cycle. Initially, all caches are empty. Every 100 instructions per core, the instruction block overlap is measured for the unique instruction blocks that were touched per core during this interval. The instruction overlap for a block is the number of L1-I caches that contain this instruction block. The graphs show the overlap ranges: 1, < 5, < 10, and ≥ 10. The measurements stop when at least half of the threads complete execution, to avoid comparing the blocks of active threads against the blocks on idle cores, which on a real system might have been displaced due to new threads.

The results show that more than 70% of the instruction blocks touched during an interval appear in at least five other cores (see point A on Figure 3.5). The overlap is even higher most of the time and more than 40% of the instructions touched during an interval appear in at least ten cores (see point B on Figure 3.5). Consequently, most of the instructions read by a given transaction are also read by at least five other transactions, and about 40% or more of these instructions are also read by at least ten transactions. In addition, all overlapping instruction fetches happen close enough in time as the instructions survive long enough to be detected in other caches. The measurements do show that the

---

3To verify that the 16 randomly chosen transactions are statistically significant, the experiments were repeated with multiple sets, each comprising 16 randomly chosen transactions. The overlap behavior was seen to be similar across multiple experiments, hence we display only one set of results.
transactions diverge, but very few instruction blocks (less than 10%) appear to be read by a single transaction.

The findings of this section serve as motivation for SLICC (Chapter 4) and STREX (Chapter 5) as they show that there is significant temporal locality across transactions of the same type. SLICC and STREX exploit this locality to improve instruction reuse in the L1-I cache.

3.3 Concluding Remarks

OLTP workloads suffer from high instruction miss stalls since their transaction instruction footprints are by far larger than current L1-I caches, on high-end server processors, thus leading to ongoing cache thrashing. Literature showed that memory stalls for OLTP workloads account for 80% of their execution time [36], and L1 instruction misses account for 70-85% of overall stall cycles [144]. We corroborate these results and show that 94% of L1 capacity misses are for instructions. Additionally, we show that recently proposed replacement policies, which reduce miss rates for some workloads, leave a lot of room for improvement compared to using larger L1-I caches.

By studying the inherent behavior of OLTP, this chapter also analyzed and quantified the instruction overlap among multiple transactions of the same type. The analysis observed significant temporal instruction overlap among similar transactions, suggesting that techniques which can exploit such overlap might be effective in reducing the problematic high instruction miss rate.
Chapter 4

SLICC: Exploiting the Aggregate Cache Capacity on CMPs

OLTP systems typically assign transactions to cores aiming to balance the work across nodes of a multicore system, irrespective of cache behavior. This chapter proposes a dynamic transaction scheduling policy: transactions can dynamically migrate among multiple cores such that they cooperatively improve instruction cache reuse. Two opportunities motivate such an approach, as Chapter 3 shows, (a) while the instruction footprint of a typical OLTP transaction does not fit into a single L1-I cache, it fits comfortably in the aggregate L1-I cache capacity of modern many-core chips [91], and (b) concurrently running transactions exhibit significant code overlap.

This chapter presents SLICC (Self-Assembly of Instruction Cache Collectives) [11], a hardware technique that utilizes thread migration to reduce instruction misses for OLTP workloads. SLICC divides the instruction footprint of a transaction into smaller code segments and spreads them over multiple cores, so that each L1-I cache holds part of the instruction footprint. As part of this process the L1-I caches self-assemble to form a collective that reduces the instruction misses for this transaction and other similar ones. SLICC exploits intra- and inter-thread instruction locality in two orthogonal ways: (1) A thread looping over multiple code segments spread over multiple caches observes a lower miss rate (as opposed to a conventional system in which each segment would evict the others from the cache), thereby avoiding thrashing. (2) A preamble thread effectively prefetches and distributes common code segments for subsequent threads, thereby reducing the total miss rate. As execution progresses, old cache collectives are naturally disassembled and new ones are formed to hold the footprints of new transactions.

SLICC is a hardware solution that avoids undesirable instrumentation, utilizes available core and cache capacity resources, covers user as well as system-level code, and requires no changes to the existing user code or software system. SLICC incurs overheads due to thread migration; context-switching overheads and potential increases in data misses. To amortize such overheads, a hardware programmer-transparent thread migration mechanism is used which provides a low context switching overhead, and the positive impact of the reduction in instruction misses must outpace the extra data misses.

As we will explain later in Section 4.4, and similar to transaction batching schemes [55, 126, 154], SLICC is expected to increase transaction latency: the time elapsed between when a transaction starts execution until completion. This chapter studies SLICC’s impact on transaction latency, and introduces
SLICC-Fr, a variant of SLICC which applies transaction thread migration to only a subset of the transactions which are expected to benefit from SLICC. SLICC-Fr achieves a better balance between instruction and data misses, resulting in improved transaction latency and overall performance.

Experimental evaluation shows that, on average, thread migration eliminates 65% of the L1 instruction misses resulting in a 70% overall performance improvement over the baseline (methodology described in Section 4.3.1). Compared to PIF [37], a state-of-the-art instruction prefetcher, SLICC improves performance by 27% for TPC-E and comes within 2% for TPC-C, with only 2.4% of relative storage area overhead. SLICC is also robust as it does not affect the performance of MapReduce [36], a cloud workload, which has a relatively small instruction footprint.

The remaining of this chapter is organized as follows. Section 4.1 discusses the potential for thread migration and sets the requirements for an ideal solution. Section 4.2 describes the automated thread migration algorithm, SLICC, highlighting various design choices, and discussing the limitations. Section 4.3 demonstrates experimentally the performance benefits of SLICC. Section 4.4 analyzes transaction latency and presents SLICC-Fr, an improved version of SLICC which provides a better balance for the reduction in instruction misses and the increase in data misses. Section 4.5 reviews related work, while Section 4.6 concludes.

4.1 Thread Migration: Exploiting Multiple Caches on a CMP

SLICC exploits the aggregate L1-I cache capacity of many cores and the availability of multiple concurrent threads with inherent code overlap. It is a hardware transaction scheduling algorithm that spreads the instruction code footprint across multiple L1-I caches. SLICC dynamically pipelines and migrates threads to cores that are predicted to hold the code blocks to be accessed next. By migrating threads, SLICC virtually increases the cache capacity observed by a thread, and thus avoid what would otherwise be capacity or conflict misses, avoiding thrashing.

4.1.1 Example Scenario

Figure 4.1 exemplifies how thread migration can reduce instruction misses. Threads T1-T5 are scheduled to run on an 8-core system, where T1-T3 and T4-T5 execute respectively transactions of the same type. The transactions’ footprints are divided into code segments, where each segment fits in the L1-I cache of a single core, but two segments would not fit together. For the time being assume that the segments are known in advance (we will later introduce runtime heuristics for segment discovery). T1 executes the following code segments in order: A-B-C-A. Thus, its instruction footprint is 3× larger than the L1-I cache size. On a conventional system, where a transaction remains on the same core until completion, T1 will fetch segment A, then evict A to fetch B, then evict B to fetch C, and so on, thus incurring significant cache thrashing. Since T2 and T3 are of the same type as T1, they share common segments with T1, but their execution paths are not identical. A conventional system would schedule T1, T2, and T3 on separate cores, and since their footprints are larger than the L1-I cache size, each thread would suffer all instruction misses.

Figure 4.1 (right) demonstrates an ideal scenario for thread migration. Initially (at time $t_0$), T1 runs on core-0. When it is done with code segment A, i.e., all cache blocks for A have been brought in to the cache, T1 migrates to core-1 (at $t_1$), where it continues execution fetching cache blocks of segment B. At the same time, T2 can be scheduled to start execution on core-0 (at $t_1$), ideally reusing all blocks in
A (miss rate close to zero). This is inter-thread reuse. The process continues and T1 warms-up caches 1 and 2 with B and C, respectively. At t3, when T1 goes back to A, it migrates to core-0 benefiting from intra-thread reuse.

Migration is beneficial even if T1-T3 do not follow identical paths, as segment D illustrates. Since T1 did not touch segment D, T2 will suffer due to the corresponding misses while executing on core-3. If T3 follows T2 on core-3, it will not suffer any instruction misses for segment D.

T4-T5, that access different code segments, benefit as well if they get assigned to a different set of cores, avoiding conflicts with T1-T3. This process applies to all subsequent threads: if they touch a code segment that exists in some L1-I cache, they migrate to the corresponding core and avoid missing for these segments.

Without thread migration multiple requests would be repeatedly sent out for the same cache block from multiple cores. With thread migration, an instruction cache block is, under the ideal scenario, requested only once, and reused multiple times.

### 4.1.2 SLICC Requirements

Based on the preceding discussion, this section presents a preliminary set of features desired in SLICC. Section 4.2 later presents SLICC’s design in detail.

We identify three requirements for SLICC. SLICC should dynamically detect: (a) When a thread should migrate, i.e., when is the current cache full; and (b) where the thread should migrate to, i.e., which remote cache, if any, holds the code segment the thread will touch next. Since transactions vary in their control flow, SLICC should (c) not impose any specific thread-to-core assignment sequence, i.e., it should not restrict similar threads to follow the exact same path.

In order to meet these requirements, SLICC needs to maintain runtime information about caches and individual threads to be able to make judicious migration decisions. First, SLICC needs a mechanism to determine whether the cache is filled-up with useful cache blocks or not. In addition, with respect to a given thread that has decided to migrate, SLICC should be able to predict which remote core holds the cache blocks that will be touched next.
4.1.3 Effect on Data Misses

When a thread migrates, it leaves data that might be reused behind. This may increase the data miss rate. Section 4.3 shows that while SLICC does increase the data miss rate, the benefit from reducing instruction misses outpaces the performance loss due to data miss increase.

Intuitively, depending on the workload, instruction misses can impact performance more than data misses. For example, modern architectures use instruction level parallelism (ILP) to hide data miss latencies. Instruction misses restrict instruction supply, rendering such ILP techniques less effective. Existing core architectures make no effort to balance the relative cost of the two types of misses. SLICC provides a way of balancing the relative costs of data vs. instruction misses. In Section 4.3 we show that SLICC reduces the aggregate L1-I and L1-D miss rate.

4.2 SLICC Design

This work presents three different SLICC designs. The first design (SLICC) is transaction-type-oblivious, while the other two exploit transaction type information. Given that threads of the same transaction type tend to have similar footprints, knowing each thread’s transaction type can lead to better migration decisions. The transaction type information is either provided by the software (SLICC-SW), or detected at runtime by the hardware based on the initial instruction sequence each thread executes (SLICC-Pp). These implementations represent the two extremes and an in-between solution in terms of hardware/software co-operation.

4.2.1 Transaction-Type-Oblivious SLICC

SLICC is a dynamic hardware thread scheduling and migration algorithm that is programmer transparent. SLICC attempts to partition on-the-fly the instruction footprint of transactions into several segments where each segment fits in the L1-I cache, but two segments do not fit together. Ideally: (1) a thread will migrate to another core when it starts touching a different segment, and (2) the destination core will already have the segment cached.

Figure 4.2 shows the sequence of events that lead to thread migration. In the steady state, each core has a running thread and a hardware queue of waiting threads. Using a naïve load-balancing strategy, newly arrived threads are scheduled to the least congested core (i.e., the core with the least number of waiting threads). A SLICC agent at each core continuously monitors execution locally in order to determine whether (Q.1) the local cache is filled-up with useful instruction blocks, if so, (Q.2) whether these blocks are useful to the current thread and for how long, and (Q.3) where to migrate to if needed.

(Q.1) Is the cache full with useful blocks? As a thread starts executing on a core it may experience many misses. If the cache contains a segment that may be useful for other threads, it is best to migrate the current thread to another core. Otherwise, it is best to allow the current thread to load a new segment in the cache. SLICC uses a “cache full” detection heuristic\(^1\) to make this decision. Initially, all caches are “empty”. To detect whether a cache has been filled up with a segment, SLICC counts the number of misses using a resettable, saturating miss counter (MC) local to each core. When the number of misses exceeds the threshold, fill-up, the cache is considered full. In the long run, all MCs

\(^{1}\)Since this work aims at building hardware-based solutions, we generally rely on simple heuristics whenever an optimal solution requires complex processes that are difficult or expensive to build in hardware. We ensure that the final solutions are correct where inaccurate heuristics can only affect performance.
will saturate, preventing new segments from being cached effectively due to premature thread migration. To create opportunities for loading new segments, SLICC resets the MC when the core’s thread queue becomes empty (i.e., the cache is considered empty). The currently cached blocks are not flushed, so if a subsequent thread requires the same segment it will still find it there. However, a thread touching a new segment will be given the opportunity to cache it.

(Q.2) Are the current cache contents useful to this thread and for how long? When running a thread on a full cache, SLICC tries to determine whether the thread is going over the cached segment, or whether it is about to move to a new segment. For this purpose SLICC measures miss dilution, that is, the recent frequency of misses (detailed in Section 4.2.2). If miss dilution is low, then SLICC predicts that the thread is only temporarily diverting away from the cached segment. Since the thread will converge again soon, it is best to not migrate to benefit from the forthcoming instruction reuse. If miss dilution is high, then SLICC predicts that the thread is moving to a different segment. If it continues execution on this core it will evict useful cache blocks, which could be reused by other threads. SLICC predicts that it might be better to migrate the thread elsewhere. The question at this point becomes where to go?

(Q.3) Where to migrate to? Ideally, SLICC would migrate a thread to a cache that has the thread’s next segment. SLICC attempts the following in order: (1) If the thread is going to touch a code segment that is available on another core, the thread migrates there. (2) Otherwise, the thread migrates to an idle core, if any. (3) The thread stays put. In the last case, migrating the thread would incur overhead and would evict remotely cached segments that may be useful for other threads. SLICC opts for incurring the instruction misses locally avoiding the migration overhead.

To detect which, if any, remote cache has the next segment, SLICC uses a short sequence of matched_t number of tags of recent misses, predicting that they form the preamble of the next segment. Conceptually, once SLICC decides to try to migrate a thread, it searches all remote L1-I caches for these recently missed tags. Section 4.2.2 explains how this search can be implemented including an incremental method that uses the existing coherence protocol responses.

Figure 4.2 summarizes the execution stages of a thread on a core until it migrates, or completes execution.

4.2.2 Implementation Requirements

Figure 4.3 shows that SLICC’s implementation comprises: (a) a cache-full detector, (b) a miss dilution tracker, and (c) a remote cache segment search unit. SLICC uses hardware thread migration, and thus, interacts with the OS as Section 4.2.5 explains in more detail. The three aforementioned units, described subsequently, track all cache accesses, including speculative ones. All the decisions related to thread migration are performed locally; the information about the local and remote caches is gathered at the originating core.

Cache Full Detection

A \( \log_2(\text{L1-I cache blocks}) \) wide saturating miss counter (MC) continuously counts the number of misses. When MC saturates at a value of fill-up_t SLICC assumes that the cache has now captured a full segment and may trigger migrations accordingly. We experimentally found that using a value in the order of \( \frac{\text{cache size}}{2} \) for the fill-up_t threshold works reasonably well, with little sensitivity to the exact
value of this parameter. Other fill-up detection mechanisms may be possible but are beyond the scope of this work and can be the target of future work.

**Miss Dilution Tracking**

It is not always beneficial to migrate threads immediately after a cache becomes full or when a thread incurs a few misses. SLICC must predict whether the thread is only temporarily diverging due to conditional control flow or whether it is moving to a completely different segment. Furthermore, since threads have to miss for a few blocks before migrating (matched tags must be located on a remote cache), a few useful cache blocks may be evicted, creating gaps in the exiting segment and causing a corresponding number of misses for subsequent threads. Finally, a thread may immediately loop back to the same code segment or may temporarily follow a somewhat different path after being selected for migration.

SLICC handles these cases by considering the frequency of instruction misses; it restricts migration to the cases when a thread starts to miss more frequently. If the thread is moving to a new segment, it will incur more misses than hits. SLICC counts the number of misses in a window of recent accesses. When this count is above the dilution threshold, \( \text{dilution}_t \), migration is enabled. The miss shift-vector (MSV) is a 100-bit FIFO shift vector recording the hit/miss history for the last 100 cache accesses (enabled when cache is filled-up). A logic-0 and logic-1 represent a cache hit and miss, respectively. When the number of logic-1 bits reaches a threshold (\( \text{dilution}_t \)), SLICC enables migration. SLICC resets the MSV with every migration.

**Remote Cache Segment Search**

When SLICC decides to migrate a thread it has to determine which cache, if any, contains the segment the thread is executing. To do so, SLICC records recently missed tags in the Missed Tag Queue (MTQ), which is a matched_t entry FIFO of \( n \)-bit entries, where \( n \) is the number of cores. A logic-1 on bit index
Figure 4.3: SLICC architecture.

$C$ for MTQ entry $i$ indicates that the $i$th recently missed cache block was cached at core $C$. Thus, by ANDing all bits at index $C$ we know whether core $C$ holds all the recently missed cache blocks. This information does not have to be exact or accurate, since it is used by a prediction mechanism. SLICC gathers this information incrementally as misses occur and stores it in the MTQ. The remote cache segment search is distributed and the decision is made locally by the core we migrate from. A directory coherence protocol could report the complete or partial sharing vector for misses that are tracked by the MTQ.

Alternatively, or if the coherence protocol is snoop-based, SLICC could broadcast the missed tags as they occur and explicitly request that remote cores identify themselves. On snoop coherence systems, these requests can piggyback on the existing snoop requests. Searching remote L1-I caches requires extra bandwidth on the remote caches that is proportional to the number of missed tags and cores.

To avoid this bandwidth overhead, we use an approximate cache signature in the form of a partial-address bloom filter that supports evictions [113]. When the index size of the bloom filter is larger than the cache set index, collisions occur only within sets. Hence on evictions, only the set of the evicted block is checked for collisions. Every core maintains such a filter, representing a superset of the currently cached blocks. In this design, once migration is triggered, remote-cache search requests are answered by the approximate signature, avoiding contention with the original cache references of the remote core. In Section 4.3.3, we evaluate the tradeoff of the bloom filter’s accuracy versus its size. We find that for a 32KB cache, a 256B bloom filter is sufficient.

If no matching remote cache is found, SLICC will attempt to find an idle core. SLICC either broadcasts a request for idle cores to report, or piggybacks this information on the responses received during the miss tag search phase. Thread migrations are relatively infrequent (every 3.2K instructions on average), reducing the relative overhead of remote cache segment and idle core searching.

### 4.2.3 Exploiting Transaction Type Information

Section 3.2 showed that the instruction footprint overlap is higher among threads of the same transaction type. The basic SLICC does not directly exploit this phenomenon. It tries to detect, on-the-fly, whether
a thread matches the segment on the core it is currently executing. Thus, a thread of type X may partially kick-out cache blocks used by threads of type Y. If the transaction type for each thread were known, SLICC could schedule similar threads on the same set of cores to reduce conflicts. We propose two SLICC variants that exploit such thread transaction type information.

Assigning Transaction Types

SLICC-SW relies on the OLTP software layer to annotate each thread upon launch with a transaction type. This guarantees correctness, but requires some modifications to the software/hardware interface, such as introducing a new instruction type, or writing into a reserved memory address.

Alternatively, SLICC-Pp uses a hardware preprocessing phase to assign types to threads as they launch. SLICC-Pp exploits the observation that in OLTP, transactions of the same type have a common entry point at the software level. Hence, the first few instructions executed are the same for same-type threads, while they differ across different-type threads. SLICC-Pp only needs to know when a new thread is launched. A middle-ware layer assigns threads in groups to a core devoted for this purpose (scout core). There, each thread executes a few tens of instructions, while the instruction addresses are hashed. The resulting values are used as thread type identifiers. Experiments show that SLICC-Pp is 100% accurate when executing a small number of instructions. SLICC-Pp dedicates one core for pre-processing.

Type-Aware Migration

Using thread type information, SLICC groups similar threads into teams to improve opportunities for co-scheduling and overlap. For each thread, SLICC records a unique numerical ID, a type ID, and an arrival time-stamp. The time-stamp of a team is that of its oldest thread. The oldest team is scheduled, without preemption if possible.

Forming teams of same-type transactions may be a complicated task. First, the overall transaction population is not evenly distributed among different transaction types; some types naturally appear more often than others. Second, forming teams that are too small may result in low code reuse (fewer threads sharing the resources), while forming teams that are too large may result in high transaction latency (threads wait too long for their turn). Section 4.4 discusses transaction latency in detail.

We design an intuitive scheduling algorithm that maximizes the core utilization and reduces the queuing delay of threads, relying on some heuristics. Team sizes differ and for an N-core architecture we categorize them into large (1.5× to 2× N threads), medium (0.5× to 1.5× N threads), and small (less than 0.5 × N threads) teams. Cores are time-multiplexed among teams. To increase core utilization, when large teams are scheduled, they are allowed to execute on all cores, however medium size teams are limited to half the resources (0.5 × N cores). Since small teams do not have much reuse to exploit, they are treated as stray threads, and are not grouped. Rather, stray threads are scheduled, individually, to idle cores, or in parallel with a medium size team. Stray threads are not allowed to migrate; they execute on the same core until completion. For SLICC-SW and SLICC-Pp, when a team of threads completes execution, SLICC resets all MCs, MTQs and MSVs.
Potential Impact on Data Misses

In addition to the potential increase in data misses due to thread migration (Section 4.1.3), grouping same-type transactions and executing them concurrently might introduce even more data misses. For example, two concurrent Payment transactions executing an Update operation on the Customer tables might contend for writes to shared meta-data (see Figure 3.4). Section 4.3 shows that the reduction in instruction misses outweighs the increase in data misses from a performance perspective. Further, Section 4.4 measures the impact of transaction grouping on data misses and suggests improvements to reduce them.

4.2.4 Hardware to Manage Threads

This section details the extra hardware structures required to manage threads. SLICC requires a thread queue that holds threads waiting for cores. The thread queues can be local to each core, or centralized. Accesses to the thread queue are off the critical path. On each core, a SLICC agent is responsible for managing the thread queue. The thread queue is a circular FIFO buffer and the first entry is executed until it migrates, completes, or gets blocked for I/O. On the latter case, the thread is moved to the end of the queue.

The team management table is responsible for forming teams of similar threads (used only by SLICC-SW and SLICC-Pp). The team management table is best thought of as centralized, since every core needs to know which cores are assigned to which teams.

4.2.5 Support for Thread Migration

The thread migration performed in SLICC transfers architectural register contents as in Thread Motion [118]. To reduce the set-up time for the thread, its context is saved in the L3 cache bank closest to the target core and is then retrieved at the target core. Since modern commercial processor technologies (e.g., Intel Virtualization (VT) [151] and AMD Secure Virtual Machine (SVM) [7]) provide hardware support for thread migration, minimal modifications are required to make the migration process transparent to higher software layers.

Typical OS kernels are responsible for assigning threads to cores. Hardware support for thread migration that is transparent to higher layers avoids any software overhead. Otherwise, the OS scheduler must be informed about these migrations. An alternative is a hybrid system in which hardware mechanisms provide counters and migration acceleration, while leaving the policy choice to software. This enables easier integration between existing schedulers and platforms with Virtualization support.

4.2.6 Limitations

This section briefly mentions some of SLICC’s limitations, which are discussed in more detail in later sections.

The key ingredient to SLICC’s success is its ability to segment the relatively large instruction footprint into smaller chunks where each fits in a single cache, and to distribute these chunks across multiple cores of a CMP. When the number of cores available, and hence the aggregate instruction cache capacity, is not sufficient to hold the entire instruction footprint, SLICC will thrash the caches, similar to the
conventional system, and not perform as anticipated. Chapter 5 studies the limitations of SLICC in these scenarios, and introduces an alternative solution [12].

Similar to transaction batching schemes [55, 126, 154], SLICC tries to improve overall system throughput (i.e., reduce the execution time of all the transactions) at the expense of potentially higher individual transaction latency. The latency of a transaction is the time elapsed between when it starts execution until completion. Section 4.4 confirms the increase in transaction latency, and presents a solution which tries to reduce the increase in transaction latency.

To identify transaction starting points, the type-aware SLICC-SW and SLICC-PP variants require modifications to the hardware/software interface. Such modifications are doable given the trends toward hardware/software co-design in the high-performance computing community [3, 159].

4.3 Evaluation

The evaluation in this section: (1) Studies the configuration thresholds for SLICC (Section 4.3.2). (2) Determines the trade-off between bloom filter size and remote cache segment search accuracy demonstrating that relatively small size filters provide high accuracy (Section 4.3.3). (3) Demonstrates SLICC’s effect on instruction (Section 4.3.4) and data misses (Section 4.3.5), compared to the Baseline. (4) Reports the performance improvement with the different flavors of SLICC compared to the Baseline and to a state-of-the-art instruction prefetcher, PIF [37] (Section 4.3.6). (5) Estimates SLICC’s energy cost (Section 4.3.7). (6) Estimates the HW cost for SLICC’s components (Section 4.3.8). (7) Reports statistics about remote cache segment search activity showing that it imposes negligible overhead on the system (Section 4.3.9).

4.3.1 Methodology

Current operating systems do not support thread migration at the hardware level. The OS kernel assumes full control over thread assignment in multi-core environments. To work around this limitation, we extract x86 execution traces using QTrace [141], an instrumentation extension to the QEMU full-system emulator [16], which are annotated to identify transactions. We then replay traces, modeling the timing of all events and maintaining the original thread sequence. We model thread migrations by injecting writes and reads for all architectural state and thread context information (six cache blocks). Section 3.1.1 details the experimental infrastructure and workloads evaluated. Energy was estimated using McPAT 1.2 [87].

As explained in Section 3.1.1, a limitation arises with this trace-based methodology: the sequence of synchronization-related events that occurred while extracting the execution traces might be altered by SLICC’s dynamic scheduling decisions. Section 3.1.1 justifies this methodology by illustrating that OLTP transactions on Shore-MT [73] incur minimal synchronization. Additionally, we observe a strong correlation between instruction code segments and shared data implying that SLICC, which distributes code segments across multiple cores and pipelines execution on these cores, may pipeline the execution on shared locks hence further reducing the impact of synchronization (see Sections 4.4 and 5.3.2).

We use misses per kilo instructions (MPKI) as our metric for instruction (I-MPKI) and data (D-MPKI) misses. We measure performance by counting the number of cycles it takes to execute all transactions. With \( N \) cores, our Baseline architecture can run up to \( N \) concurrent threads with the OS making thread scheduling decisions. To enable SLICC to form teams as described in Section 4.2.3, it
manages a thread pool of up to $2N$ threads. Unless otherwise indicated, all SLICC results are for the SLICC-SW configuration – i.e., the SW layer transfers knowledge about thread types to the HW layer.

### 4.3.2 Exploring SLICC’s Parameter Space

SLICC utilizes three thresholds to make thread migration decisions: `fill-up_t`, `matched_t`, and `dilution_t`. This section explores their effect on L1 cache misses and overall performance. As defined in Section 4.2, `fill-up_t` sets the threshold for the initial fill-up period for an L1-I cache during which instructions are brought in until the cache is almost full. `Matched_t` sets the minimum number of tags that should be found on a remote cache before a thread migrates to it. `Dilution_t` is the minimum number of misses in the last 100 accesses to allow migration. The parameter choices could be thought of as a 3D space. Empirically, we observe that three parameters are independent. Accordingly, to simplify, we first keep `dilution_t` value at zero, and explore the parameter space of `fill-up_t` and `matched_t`. In addition, we assume zero-overhead to search for remote tags. We later model an actual search mechanism.

#### Varying Fill-up_t and Matched_t

Figure 4.4 reports I-MPKI, D-MPKI and performance relative to the Baseline as a function of `fill-up_t` (X-axis, top row) and `matched_t` (X-axis, bottom row). The `fill-up_t` values shown correspond to fractions of the L1-I cache capacity (512 cache blocks): $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$, and one. The `matched_t` range shown is 2 – 10; larger `matched_t` values further degrade performance. SLICC reduces instruction misses and increases data misses. Since instruction stalls, for OLTP workloads, account for 70-85% of overall cycle stalls [144], reducing instruction misses has a major effect on performance.

The results show that SLICC is not sensitive to different values of `fill-up_t`. `Fill-up_t` is actually a proxy for warming-up the caches; when the MC value saturates at `fill-up_t`, and the caches are deemed warm, migrations are allowed. Hence only the first migration from a core waits until MC saturates, and afterwards with more migrations the effect of `fill-up_t` diminishes. TPC-C and TPC-E transactions have large instruction counts and migrations. Figure 4.4 demonstrates that for `matched_t` values larger than four, performance benefits drop. On the other hand, although the overall MPKI at two is lower, performance at four is higher due to fewer migrations, and thus, lower overhead.
Chapter 4. SLICC: Exploiting the Aggregate Cache Capacity on CMPs

Figure 4.5: MPKI and relative performance as a function of $dilution_t$ ($fill-up_t = 256$, $matched_t = 4$).

Varying $dilution_t$

Using a small value for $dilution_t$ triggers more frequent migrations. Using too large a value for $dilution_t$ reduces migration overhead, but with a possible I-MPKI increase since it results in partial cache thrashing. Figure 4.5 shows L1 MPKI and relative to the Baseline performance for $dilution_t$ values 1 through 30 when $fill-up_t = 256$ and $matched_t = 4$ (best configuration from Figure 4.4). As $dilution_t$ increases, instruction misses are reduced improving performance up to a point. Afterwards, larger $dilution_t$ leads to fewer migrations, lesser overhead, but higher I-MPKI. There is a tradeoff between reducing instruction misses, and reducing migration overhead. Beyond $dilution_t$ values of 28 (TPC-C) and 24 (TPC-E), although the overall MKPI is reduced, the performance degrades due to more limited migration. At even higher $dilution_t$ values, migrations become rare and performance drops below the Baseline (for SLICC-SW, teams of transactions are injected to start on the same initial core, thus when migration stops some cores are underutilized).

In the remaining parts of this evaluation, we use $dilution_t = 10$, $fill-up_t = 256$ and $matched_t = 4$. The last parameter means that the Missed Tag Queue (MTQ) needs to keep track of only the four most recent misses, and that remote cache segment searching only requires finding where those four blocks are cached.

4.3.3 Cache Signature Accuracy

Section 4.2.2 explained that using a partial-address [113] bloom filter reduces the overhead of remote cache segment searching. Figure 4.6 shows the accuracy of bloom filters of different sizes. The smallest bloom filter requires 512 bits to support evictions for a 32KB cache, with 64B blocks, and 512-sets. Accuracy is measured for all cache accesses and an access is accurate if the bloom filter and the cache agree on whether this is a hit or a miss. The trend is similar for TPC-C and TPC-E. In the rest of this evaluation, we experiment with 2K-bits filters (99.3% accuracy) as their effect on performance is less than 0.5% compared to an ideal system which searches the cache with zero overhead.

4.3.4 Instruction Miss Change

Having determined a good SLICC configuration, this section shows that the three SLICC variants are able to reduce instruction misses much more than they increase data misses. Figure 4.7 shows the L1 I-MPKI for the Baseline, SLICC, SLICC-SW, and SLICC-Pp. For MapReduce, since the instruction
footprint fits in a 32KB cache, SLICC does not migrate threads and hence does not affect instruction or data misses.

Focusing on the other workloads, SLICC-SW reduces I-MPKI more than SLICC or SLICC-Pp. Compared to the Baseline, SLICC-SW reduces I-MPKI by 67% and 64% for TPC-C and TPC-E, respectively. I-MPKI reductions are slightly lower with SLICC-Pp, more so for TPC-E than TPC-C, for the following reason: SLICC-Pp uses one core for preprocessing. Given the runtime transaction mix and transaction footprint sizes, having that extra core can be more important. All three variants of SLICC are sometimes forced to overcommit the caches by concurrently running transactions whose aggregate footprint does not fit in the total available L1 cache capacity. When overcommitting the caches, it is best to use stray threads since little opportunity for instruction reuse is lost when overcommitting with stray threads (a stray thread by definition is one that has few, if any, other ready threads sharing the same footprint). Overcommitting with non-stray threads happens more often for TPC-E than TPC-C partly because only 3% of TPC-E threads are stray compared to 12% of TPC-C threads. Furthermore, the need for stray threads is higher for TPC-E than TPC-C; SLICC spreads the transactions of TPC-E across 8 – 10 cores, while TPC-C’s transactions are spread across up to 14 cores.

SLICC improves I-MPKI less than the thread-type-aware alternatives, as it has to predict which is the next segment a thread will execute and where that segment currently is, using only a small preamble of the segment. The difference in reduction is more pronounced for TPC-C than TPC-E. TPC-C’s overall instruction footprint is larger, resulting in higher variability in the instruction stream. Nevertheless,
SLICC reduces instruction misses by 51%, on average.

As a comparison of the results for TPC-C-10 to those for TPC-C-1 shows, I-MPKI reductions persist mostly unaffected with the larger database.

4.3.5 Data Miss Change

SLICC can potentially increase data misses for several reasons (enumerated 1 to 4). First, from the perspective of individual transactions, during thread migration from core-A to core-B, three possible scenarios lead to extra data misses that would not have occurred otherwise: (1) a thread may read data on core-B that it fetched on core-A (extra misses on core-B for the same data blocks), (2) data writes on core-B to blocks fetched on core-A lead to invalidations that would not have occurred without migration (extra misses on core-B and invalidations on core-A), and (3) when a thread returns to core-A, it may find that data it originally fetched has since been evicted by another thread, or invalidated by itself (extra miss on core-A). Second, from the perspective of a transaction group, (4) forcing similar transactions which modify shared data to execute concurrently would result in more cache invalidations. Section 4.3.6 shows that instruction misses are more expensive than data misses performance-wise. Most of the extra data misses are served on-chip, allowing out-of-order execution to mostly absorb their latency.

Figure 4.7 reports the L1 D-MPKI for all three SLICC variants and shows that SLICC-SW incurs an increase in D-MPKI of 16%, 6% and 9% over the Baseline for TPC-C-1, TPC-C-10 and TPC-E, respectively. The other two variants exhibit a similar trend in D-MPKI increase. There is less locality and sharing in the larger data set of TPC-C-10, reducing the D-MPKI overhead when migrating.

Most of the increase in D-MPKI is for stores, which form 45% of total memory accesses, while loads are nearly unaffected. Due to slightly fewer migrations, SLICC-Pp increases D-MPKI less than SLICC-SW. On the other hand, SLICC is worse with an average D-MPKI increase of 13%, due to more migrations.

Can Data Prefetching Mitigate the Increase in D-MPKI?

We considered data prefetching to mitigate the increase in data misses. For each thread, we recorded the tags of the last \( n \)-referenced data blocks and then prefetched those blocks to the core the thread migrated to. This prefetter did not improve performance, and past a value of \( n \), it hurts performance. There are several reasons why this prefetching proved ineffective. (1) The prefetched data increased the bandwidth on lower cache levels, which affects overall performance when \( n \) is high. (2) When \( n \) is low, there was not enough reuse. (3) Finally, not all prefetched blocks are re-referenced again, as Section 3.1.2 demonstrated.

Summary

This section showed that all SLICC variants improve I-MPKI significantly with a minor increase in D-MPKI, which is negligible with the larger database for TPC-C. These results suggest that SLICC can improve performance if, as expected, instruction cache misses degrade performance more than data cache misses; most extra data misses resulting from migration can be partially overlapped with out-of-order execution. If this is the case, SLICC has the potential to offer a better balance of instruction vs. data cache misses over a conventional architecture.
Similar to D-MPKI, D-TLB\textsuperscript{2} misses increase on average by 11% and 8% with SLICC and SLICC-SW, respectively. I-TLB misses are within +/- 0.5% of the Baseline.

### 4.3.6 Performance

This section reports the overall performance of SLICC relative to the Baseline, a Next-Line instruction prefetcher, and a state-of-the-art instruction prefetcher, PIF [37]. Figure 4.8 shows a $1.6 \times$ and $1.87 \times$ performance improvement over the Baseline for SLICC-SW, on TPC-C-1 and TPC-E, respectively. On average, SLICC-SW and SLICC improve performance by $1.74 \times$ and $1.67 \times$ over the Baseline, and $1.43 \times$ and $1.47 \times$ over a next-line instruction prefetcher.

Ferdman et al. report that PIF has nearly perfect coverage of L1-I misses, and its performance is close to that of a perfect-latency cache which always responds with the hit latency [37]. Thus, we model PIF’s performance using a perfect-latency cache, while the externally-observed behaviors of the cache (on-chip and off-chip traffic) matches a Next-Line prefetcher [37]. PIF’s storage requirements are $\sim 40$KB per core. For TPC-C, SLICC-SW is within 2% of PIF’s performance, with only 2.4% of PIF’s storage requirements per core (see next section). For TPC-E, SLICC-SW outperforms PIF by 27%. SLICC’s speedup compared to PIF (and also the Baseline) is a result of intelligent thread scheduling. Current schedulers assign threads to cores irrespective of their inherent-locality. Thus, even for larger caches, multiple similar threads run concurrently on different cores, each observing its own set of misses, for the same cache blocks. By pipelining similar threads, SLICC increases temporal inter-thread locality, hence it decreases overall miss rate observed by multiple threads.

PIF can prefetch for arbitrary applications and is not tied to transactional systems. However, PIF’s meta-data storage tables scale in size with the instruction footprint, consuming power and space on the order of the L1 instruction caches. Furthermore, prefetching the cache blocks consumes bandwidth and energy. In contrast, SLICC’s benefits are limited only to transactional-based systems. However, SLICC requires relatively tiny meta-data tables and improves cache performance by increasing the locality of cache blocks. Thus, SLICC saves the bandwidth and power cost of prefetching, and its meta-data power costs are negligible.

MapReduce, which has an instruction footprint that fits in the L1-I cache, remains practically unaf-

---

\textsuperscript{2}A translation lookaside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.
4.3.7 Energy

This section reports the impact of various SLICC designs on power and energy. Figure 4.9 plots the power, energy and energy-delay products (ED and ED$^2$) of SLICC, SLICC-Pp and SLICC-SW normalized to the no-prefetching Baseline. We report overall system energy. The power and energy are broken down into the static and dynamic components. The previous sections demonstrated that all SLICC variants reduce instruction miss stalls significantly, thereby increasing the utilization of the cores’ resources; i.e., the cores become more active at executing instructions. Consequently, the dynamic power increases by up to 57%, resulting in an overall power increase of up to 40%. On the other hand, taking into account that the overall system throughput improves, SLICC reduces overall energy consumption by up to 29%. SLICC also reduces ED and ED$^2$ metrics by up to 65% and 82%, respectively.

Naturally, since SLICC-SW is the best performing variant, Figure 4.9 shows that it consumes slightly more power than the type-oblivious SLICC (3% on average), however, with slightly less energy (0.5% on average). These findings demonstrate that a power–performance trade-off exists between various SLICC designs, however, overall energy is almost unaffected.

4.3.8 Hardware Cost

Table 4.1 details the cost of all SLICC’s hardware components. Sections 4.2.2 and 4.2.4 described these components.

The cache monitoring unit tracks the most recent missed blocks and whether other cores hold these blocks in the MTQ. It also tracks the frequency of misses in the MSV. A bloom filter acts a compressed signature of the cache contents.

The thread queue holds threads waiting for cores. Each entry contains a unique numerical ID, a pointer to the threads’ context, and a core ID. Since accesses to the thread queue are off the critical path, we opt to estimate the cost for a centralized queue. Fewer entries are required when the queues are local to each core.

The team management table is responsible for forming teams of similar threads. Each entry consists of: a unique numerical ID, a type ID, a team ID, index within a team, and a time stamp. For this work
Table 4.1: SLICC’s hardware component storage costs.

<table>
<thead>
<tr>
<th>Component (SLICC-SW &amp; SLICC-Pp)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Missed-Tag Queue (MTQ)</strong></td>
<td>60-bits</td>
</tr>
<tr>
<td><strong>Miss Shift-Vector (MSV)</strong></td>
<td>100-bits</td>
</tr>
<tr>
<td><strong>Cache Signature (Bloom Filter)</strong></td>
<td>2K-bits</td>
</tr>
<tr>
<td><strong>Thread Queue</strong></td>
<td>30-entries (12-bits numerical ID, 48-bits pointer to thread context, 4-bits core ID)</td>
</tr>
<tr>
<td><strong>Team Management Table</strong></td>
<td>60-entries (12-bits numerical ID, 32-bits timestamp, 4-bits type ID, 4-bits team ID, 8-bits team index)</td>
</tr>
</tbody>
</table>

**Total**
- **Cache Monitor Unit**: 2208 bits (276 Bytes)
- **Thread Scheduler**: 1920 bits (240 Bytes)
- **Team Formation (SLICC-SW & SLICC-Pp)**: 3600 bits (450 Bytes)

**Grand Total**: 7728 bits (966 Bytes)

we simulated a centralized copy at one of the cores and modeled the necessary traffic.

With an over-provisioned thread queue of 30 threads, and a copy of the team management table, per core, SLICC requires a maximum of 966 bytes in addition to logic. All logic operations for SLICC are not on the critical path.

### 4.3.9 Remote Cache Segment Search Activity

As per the description of Section 4.2.2, a thread that wants to migrate has to find which cache, if any, holds the next code segment. Our results, thus far modeled this searching by including separate messages for the corresponding miss messages using separate broadcasts. We do so to obtain an upper limit of the overhead these messages may induce. We report the frequency of these messages as Broadcasts per Kilo Instructions (BPKI) and find that it is very low. For TPC-C, BPKI is 2.204 for SLICC and 0.28 for SLICC-SW and SLICC-Pp. For TPC-E, BPKI is 1.328 for SLICC and 0.367 for SLICC-SW and SLICC-Pp. As Section 4.2.2 explained, these requests are required anyhow for normal miss processing. The ownership information required by SLICC is either already available or should be possible to piggyback on existing responses.

### 4.4 Analysis of Transaction Latency

Similar to software transaction batching schemes [55, 126, 154], SLICC improves the overall throughput but may increase transaction latency; the length of time elapsed between when the transaction starts execution until it ends. This section analyzes SLICC’s behavior from a transaction latency perspective.
and highlights potential areas of improvement.

In a typical OLTP system, a transaction executes on a dedicated core, and remains uninterrupted on that core until completion (assuming no preemption). In contrast, SLICC migrates transactions to cores which hold the forthcoming code segment, incurring migration-related overheads. These overheads include the thread context switching, and queuing delays which are higher on cores which hold hot code segments. Figure 4.10 shows a distribution of transaction latencies for the Baseline system and SLICC, for TPC-C (left) and TPC-E (right). The graphs confirm the prior hypothesis that SLICC increases transaction latency; SLICC almost doubles the average latency of transactions, while increasing the tail latency for TPC-C. The rest of this section studies: Why does SLICC increase latency that much? Can we improve it?

### 4.4.1 Performance of Individual Transactions

To better understand the sources of the latency increase, we study how individual transactions perform. As Chapter 3 explained, transactions of the same type are similar in code structure and memory access behavior, hence, we group transactions by their type. Figure 4.11 shows the average latency of TPC-C and TPC-E transaction types, comparing the Baseline and SLICC. Furthermore, SLICC’s latency is broken down into the time transactions spend actively executing on cores, and the migration-related overheads (context switching and queuing). The results suggest that for some transaction types (e.g., Delivery and Trade Status), SLICC increases latency significantly, while for the others the increase is smaller. Similarly, the amount of performance improvement gained from SLICC varies with transaction types. For example, the time Delivery transactions spend executing on cores is higher than the corresponding Baseline latency, indicating that the performance of Delivery transactions is not improved by SLICC. The opposite is true for New Order transactions where SLICC seems to cut their execution time by half. These results suggest that a selective approach might work better, where SLICC is only applied on a subset of the transactions. The rest of this section is organized as follows. Section 4.4.2 studies SLICC’s impact on instruction and data misses per transaction type, and shows that not all transactions benefit equally from SLICC. Section 4.4.3 builds on these insights and shows that a system which groups and migrates only a subset of the transactions can reduce latency and increase overall performance.
Figure 4.11: Average latency per transaction type for the Baseline and SLICC. SLICC’s latency is broken down into active execution time and migration-related overheads.

Figure 4.12: Instruction (left) and data (right) misses per transaction type, for the Baseline and SLICC.

### 4.4.2 SLICC-Friendly Transactions

Figure 4.12 shows the L1 instruction misses (I-MPKI, left) and data misses (D-MPKI, right) per transaction type, for the Baseline and SLICC. While SLICC consistently reduces I-MPKI significantly for all transaction types, its impact on D-MPKI varies. Transactions with relatively high Baseline data misses see either a mild or no increase in D-MPKI with SLICC (e.g., New Order and Market Watch). Vice versa, SLICC significantly increases D-MPKI for transactions with relatively low Baseline D-MPKI (e.g., Delivery and Security Detail).

These findings suggest that some transaction types are more prone to the data-miss-increase scenarios described in Section 4.3.5. These scenarios can be attributed to either (a) grouping similar transactions together, or (b) due to thread migration. To identify which of these hurt data misses more, we perform two synthetic experiments. To test for (a), we evaluate a scheduler that groups similar transactions together into teams, similar to SLICC, however it does not perform any thread migrations (Baseline-grouping). To test for (b), we modify SLICC to execute one transaction at a time, such that a transaction is allowed to migrate among all cores without preemption and without interfering with other threads (SLICC-migration-only).

Evaluation shows that SLICC primarily increases data misses due to transaction grouping, and thread migration mostly reduces data misses. Figure 4.13 plots the L1 D-MPKI for the Baseline, Baseline-grouping and SLICC-migration-only, showing that transaction grouping increases data misses the most.
Furthermore, thread migration increases data misses for only Delivery and Security Detail, while reducing data misses for the rest. These results imply that data accesses are sometimes correlated with code segments, thus migrating threads to improve instruction locality also improves data locality. Chapter 5 presents STREX which exploits both instruction and data locality among same-type transactions.

### 4.4.3 SLICC-Fr Transaction Scheduler

**SLICC-Fr** is a modified version of SLICC where only SLICC-friendly transactions, those which are expected to improve performance, are grouped into teams and allowed to migrate, while the remaining transactions execute individually and do not migrate. SLICC-Fr is a two-phase hardware transaction scheduler. First, it profiles transactions to determine which types are SLICC-friendly and which are not. Profiling includes collecting statistics per transaction type while executing in the normal Baseline mode (no grouping, no migration) and in SLICC mode. Second, based on this profiling, SLICC-Fr applies the normal SLICC transaction grouping and thread migration algorithm only on SLICC-friendly transactions, while the remaining transactions are treated as stray threads.

In this work, we opt to use a simple empirical model to determine SLICC-friendly transactions, which proves sufficient. The collected statistics include the time transactions spend executing on cores (active), and spend in thread queues (queuing). For a transaction type to qualify as SLICC-friendly, SLICC must improve its execution time, while not increasing its latency by more than double the Baseline latency (Formula 4.1). Given this formula, SLICC-Fr executes the following transactions as stray threads: Delivery, Order Status, Security Detail and Trade Status.

\[
(\text{active}_{\text{SLICC}} < \text{active}_{\text{Baseline}}) \land ((\text{active}_{\text{SLICC}} + \text{queuing}_{\text{SLICC}}) < 2 \times \text{active}_{\text{Baseline}})
\]

Stray transactions may negatively interfere with SLICC-friendly transactions for two reasons: (1) Stray threads execute on cores from start to end thrashing the caches. (2) They limit the number of cores available on which SLICC-friendly transactions would otherwise spread their instruction footprint. To limit and measure this interference, SLICC-Fr puts an upper bound on the number of stray threads concurrently executing in the system.

SLICC-Fr significantly outperforms the Baseline and SLICC. Figure 4.14 compares the I-MPKI, D-MPKI (left Y-axis) and performance (right Y-axis), for the Baseline, SLICC, and SLICC-Fr. For the
Chapter 4. SLICC: Exploiting the Aggregate Cache Capacity on CMPs

latter, different configurations are shown in the form SLICC-Fr-sX, where X is the maximum number of concurrent stray threads. As expected, SLICC-Fr incurs 44% more instruction misses compared to SLICC, but is still 51% lower than the Baseline, on average. SLICC-Fr’s impact on data misses varies with the number of concurrent stray threads: more concurrency leads to more data misses. Figure 4.13 demonstrated that the data miss rate for some transaction types is significantly impacted by concurrency. SLICC-Fr-s1 prevents such transactions from executing concurrently, hence reducing data misses compared to the Baseline and SLICC by an average of 20% and 30%, respectively. These results account for the profiling overhead. For the simulated 1.2B instruction sample, TPC-C and TPC-E spend 13% and 20% in profiling, incurring a performance loss of 6% and 10%, respectively. However, real systems would execute these workloads for much longer periods amortizing these overheads, since profiling does not need to be repeated often.

Figure 4.14 shows a distribution of transaction latency of the Baseline, SLICC and SLICC-Fr-s1. SLICC-Fr-s1 manages to bridge the latency gap between the Baseline and SLICC. Compared to the Baseline, SLICC-Fr-s1 increases the average latency for TPC-C and TPC-E by only 31% and 19% (compared to 88% and 86% with SLICC), respectively.
Summary
Evaluation shows that the proposed modifications to SLICC improve performance and reduce transaction latency. SLICC-Fr only applies transaction grouping and migrations to those transactions that are expected to see benefit. In doing so, SLICC-Fr achieves a better balance of instruction and data miss reduction. SLICC-Fr uses a simple formula to classify transaction types which proves sufficient. However, transactional systems other than the evaluated standard benchmarks might require different formulas.

4.5 Related Work

Section 2.2.2 reviewed prior instruction cache prefetching designs. This section reviews prior work specifically targeting OLTP instruction cache misses and thread migration.

4.5.1 Synchronized Transactions through Explicit Processor Scheduling

STEPS aims to minimize OLTP instruction misses from the software side [54, 55]. Like SLICC, it groups threads executing similar transactions into teams. It, either manually or by using a profiling tool, breaks each transaction’s instruction footprint into smaller instruction chunks in a way that each chunk can fit in the L1-I cache. Then, rather than executing the whole transaction without any interruption, all the threads in the same team execute the first chunk on the same core; when one transaction completes the execution of the chunk, it context switches to allow another thread, and so on. STEPS repeats this process for all the chunks, allowing instruction reuse across many threads for each chunk. SLICC exploits the same way of re-using the instructions already brought into the cache by previous threads. However, rather than context-switching on the same core, SLICC migrates threads to another core so that they can continue their execution. Moreover, SLICC dynamically detects the synchronization points in a transaction rather than using a priori manual or profiling based software instrumentation. Chapter 5 studies combining the time-domain pipelining of STEPS with the space-domain pipelining of SLICC.

4.5.2 Computation Spreading

Chakraborty et al. demonstrate a high-degree of redundancy in system-level code fragments across threads concurrently running on multiple cores [25]. They propose CSP, which employs thread migration to distribute the dissimilar instruction code segments and group the similar ones together. For system code which is commonly used by multiple threads, CSP fragments and distributes the code across a group of dedicated cores. CSP then migrates threads to these dedicated cores to execute system code. When threads are done, they return back to their original cores to resume execution for the user-level code. Thus CSP is limited to fragmenting OS code, losing opportunities of fragmentation within user code. SLICC generalizes thread migration to include interleaved user-OS code fragmentation points. In addition, thread migration in SLICC is managed by the hardware, while with CSP, the OS performs the migrations.

4.5.3 Advanced Instruction Chasing for Transactions

Recent work by Töziin et al. [142] proposes ADDICT, which builds on SLICC’s concept of spreading a transaction’s instruction footprint across multiple L1 caches. ADDICT is a scheduling technique which
operates at the finer granularity of database actions, as opposed to the coarser granularity of transaction scheduling. ADDICT segments the transactions into database operations, and further segments these operations into smaller actions, where the instruction footprint of each action fits in a single L1 instruction cache. Then, it assigns these actions to specific cores, and migrates the transactions to the cores which hold the next action to be executed. To operate at this fine granularity, ADDICT requires information about database operation boundaries within the code. Using this information, ADDICT first profiles the application to determine segmentation points (instruction PCs) within a database operation that would result in actions small enough to fit in a single L1 cache. At runtime, when a segmentation point is encountered, ADDICT migrates transactions to the core specifically assigned to the next action.

4.5.4 Data-Oriented Transaction Execution

DORA indirectly affects the instruction footprint of a transaction [109]. It divides a transaction into smaller actions based on the data being accessed at a particular transaction part. Then, each of those actions is sent to its corresponding worker thread, reducing the overall number of instructions executed by a single thread per transaction. Such a design might not necessarily break a transaction into instruction parts that can fit in L1-I. However, if combined with SLICC, it can give better hints on where to migrate or reduce the total number of migrations needed to be done per worker thread.

4.5.5 Other Thread Migration Techniques

Other recent thread migration proposals target power management, data cache, or memory coherence [97, 118, 90, 128], but do not focus on instruction misses.

Rangan et al. [118] propose multi-core systems with heterogeneous power-performance capabilities. They propose Thread Motion, a method to migrate threads among cores depending on the program’s time-varying compute density. When a thread is compute-intensive, it migrates to the higher-performance cores. Vice versa, when its computation requirements fall back, it migrates to the less-power consuming cores.

Michaud et al. partition the data working-set into smaller subsets which fit in a single private cache [97]. Then, these subsets are distributed on multiple cores, thus better utilizing the aggregate cache capacity of a chip-multiprocessor. Lis et al. [90] and Shim et al. [128] try to localize data accesses by fine-grained instruction-level thread migration decisions. On a shared-memory system, where data accesses might reside on remote cores, instead of fetching the data to the local cache, a thread migrates to the remote core and hence accesses to the data become local accesses.

4.6 Concluding Remarks

Previous works tackled OLTP’s instruction footprint problem in software or hardware, but they either require code instrumentation or relatively large on-chip data structures. This chapter presented a solution based on thread migration, SLICC. Similar to CSP [25] and STEPS [55], SLICC exploits the code commonality observed across multiple concurrent threads. Unlike CSP, SLICC does not limit code reuse to OS code segments. Unlike STEPS, instead of context switching on the same core, SLICC distributes the instruction footprint across multiple cores and migrates execution. SLICC is a low-level hardware
algorithm that requires no code instrumentation and efficiently utilizes available cache capacity, by improving intra- and inter-thread locality.

Over the baseline, on average, the best variant of SLICC, SLICC-Fr, reduces the instruction and data misses for OLTP by 51% and 20%, respectively, resulting in an overall performance improvement of 125%. When tested on MapReduce, a cloud workload that has a relatively small instruction footprint, SLICC was robust and did not affect the L1 miss rates or performance.

The following chapter presents an alternative scheduling policy, STREX [12], which also improves instruction cache locality, however instead of spreading the instruction footprint across multiple cores, it dynamically pipelines and time-multiplexes transactions on a single core. STREX has the advantage of operating on a single core, thus its benefits are not affected by the available runtime aggregate L1-I cache capacity. In contrast, SLICC is effective only if there is enough aggregate cache capacity to fit the transactions’ instruction footprint.
Chapter 5

STREX: Stratifying Transaction Execution

Chapters 3 and 4 demonstrated that OLTP application behavior and CMP integration offer opportunities to improve instruction cache locality. Specifically, SLICC [11] exploits the following observations: (1) beyond a certain core count, the aggregate L1 capacity of a chip multiprocessor becomes sufficiently large to capture not only the footprint of each transaction individually, but the footprints of all of the concurrently running transactions. (2) There is significant overlap in the instruction code segments executed within and across transactions [55]. SLICC dynamically migrates transaction execution across cores to avoid instruction cache thrashing, allowing multiple transactions to reuse cached instructions. Through thread migration SLICC exploits both intra- and inter-transaction overlap by converting it into instruction reuse in the caches. SLICC was demonstrated on a 16-core CMP, but this chapter demonstrates that it is not as effective and may hurt performance when the footprint of all concurrently running transactions exceeds the aggregate L1-I capacity.

While mainstream 16-core server CMPs are around the corner and the number of cores on-chip is expected to grow in the future, the number of cores per application, and hence the aggregate L1-I cache capacity, may not always be sufficient for SLICC to be beneficial. Data center design and application trends influence the available per application core count. (1) Current data center design trends are toward consolidating more virtual machines on servers as this increases utilization, improves security and energy efficiency, and reduces costs and management overheads [23, 64, 153]. (2) A data center may run multiple OLTP workloads, each with different transactions. (3) While L1-I capacities remain cycle-time limited, OLTP transaction instruction footprints are increasing. Transactions are becoming more complex and thus larger as a result of additional functionality such as data analytics (e.g., WebSphere [63], WebLogic [105]) or more complex logic. Accordingly, it is desirable to avoid SLICC’s performance cliff and to develop an instruction stall reduction technique that is effective irrespective of the number of available cores.

Section 3.2.3 demonstrated significant temporal code overlap among similar OLTP transactions, suggesting that their execution order can be stratified to increase instruction reuse in the caches. Motivated by this observation, this dissertation proposes STREX [12], a technique that exploits inter-transaction locality by grouping and synchronizing the execution of similar transactions into time slices. During each time slice, a lead transaction brings into the L1-I an instruction code segment that fits in the L1-I
and that all other transactions ought to reuse. Ideally, when the transactions within a group overlap perfectly, only the lead transaction incurs all necessary misses; these are the misses that a transaction would incur on a conventional system anyhow. As a result of STREX’s time slicing, the remaining transactions find all the instructions they need in the L1-I avoiding misses completely. STREX uses local information at each core observing cache block allocations and evictions to orchestrate transaction execution. STREX substantially reduces instruction stall time when running OLTP workloads, while remaining effective even when the aggregate L1-I capacity is insufficient to store the footprints of all concurrently executing transactions.

STREX’s approach to reducing instruction misses is similar in spirit to STEPS [55]. STEPS is a software technique that was demonstrated on single cores. STEPS relies on manual code instrumentation, a high-overhead error prone process, and produces platform dependent code that is not portable. STREX does not suffer from these limitations as it is a programmer transparent technique.

Experiments demonstrate that while STREX performs substantially better than SLICC when the number of cores is limited, the opposite is true when there are enough cores available. Accordingly, this chapter presents a mechanism for dynamically selecting between the two techniques. The selection mechanism uses the available core count while periodically and transparently sampling transactions to measure their footprint needs.

When compared to the baseline system, which does not include any mechanism to improve cache performance, STREX exploits transaction code overlap and reduces L1 instruction and data misses respectively by 37% and 13%, on average for two to 16 cores. When compared to a model of a state-of-the-art instruction prefetcher, PIF [37], STREX’s performance is within 5% for TPC-C [146] and 9% better for TPC-E [147]. If the available core count, and hence the available aggregate on-chip capacity, does not fit the transactions’ instruction footprint, STREX outperforms SLICC by an average of 49%. The experiments show that the hybrid mechanism closely follows the performance of the best-performing technique (SLICC or STREX) given the number of cores available and the workload’s footprint. Finally, the experiments show that STREX is more effective at reducing instruction misses compared to the replacement policies described in Section 3.1.3.

The rest of the chapter is organized as follows. Section 5.1 discusses how transaction behavior can be potentially exploited to improve instruction reuse in caches. Section 5.2 describes STREX and its implementation. Section 5.3 demonstrates experimentally the performance benefits of STREX. Section 5.4 illustrates and evaluates a hybrid system which incorporates SLICC and STREX. Section 5.5 reviews related work, while Section 5.6 concludes.

5.1 Exploiting Transaction Instruction Overlap

Given that OLTP transactions exhibit significant instruction footprint overlap over time (Section 3.2.3), their execution order can be stratified to increase instruction reuse in the caches. This section contrasts two different ways of exploiting stratified execution to improve instruction cache reuse. Both methods break transaction execution into slices, where during each slice, a transaction executes through an L1-I sized code segment. The first method that is the basis for STREX, time-multiplexes and synchronizes the execution of slices from multiple transactions over the same core. The second, previously presented method, SLICC (Chapter 4), does so across multiple cores using thread migration. Both methods increase instruction reuse by having slices that touch the same code segment execute in sequence over
Chapter 5. STREX: Stratifying Transaction Execution

Figure 5.1: Scheduling three identical transactions. Each transaction executes in order code segments A, B and C. (a) Conventional Execution: each transaction executes until completion. (b) STREX: transactions are multiplexed on the same core. (c) SLICC: code segments are distributed among multiple cores, and transactions migrate to the cores holding the code segment to be executed.

the same cache where that code segment resides.

**Conventional Execution:** Figure 5.1(a) shows how three perfectly overlapping transactions would execute under a conventional OLTP system. The example transactions execute three code segments A, B, and C in order. Each segment fits in the L1-I, but any two segments exceed its capacity. When these transactions execute in a conventional OLTP, they take turns thrashing the cache since each executes segments A through C in order before allowing the next transaction to execute. Each segment incurs an overhead due to instruction cache misses.

**STREX:** Figure 5.1(b) shows a better way of utilizing the L1-I where the first, lead transaction executes segment A incurring an overhead as previously. However, instead of proceeding to execute segment B, transaction 1 context switches allowing in turn, transactions 2 and 3 to execute instead. Transactions 2 and 3 find segment A in the L1-I and thus incur no overhead due to misses. Once all three transactions execute the first segment, execution proceeds to segment B and so on. Overall, transaction throughput increases because there are fewer L1-I misses. STREX attempts to achieve the execution order depicted in Figure 5.1(b) automatically.

**SLICC:** CMPs enable the migration-based approach used by SLICC. As long as there are enough cores so that the aggregate L1-I capacity can hold all code segments, a transaction can migrate to the core whose L1-I cache holds the code segment the transaction is about to execute. For example, as Figure 5.1(c) shows, the lead transaction can execute segment A first on core 1, then migrate to core 2 where it would execute segment B, then migrate to core 3 where it would execute segment C. Transactions 2 and 3 can follow in a pipelined fashion, finding segments A, B, and C, in cores 1, 2, and 3, respectively. While transaction 1 incurs an overhead when fetching the segments for the first time, the other transactions do not.

STREX and SLICC exploit the same inter-transaction temporal overlap. SLICC has the additional advantage that it can exploit intra-transaction far-flung locality. For example, if the three transactions in Figure 5.1 executed A, B, and C in a loop, SLICC would only fetch each segment once storing...
them over three separate L1-Is. STREX would necessarily fetch each segment once per iteration as the
segments cannot fit into a single L1-I. However, Section 5.3.3 shows that SLICC performs well only
when there are enough cores to fit the footprint of all concurrently running transactions. In addition,
when there are fewer cores in the system, SLICC may even cause performance degradation. The rest of
this chapter investigates a practical implementation for STREX, which can be used when the number
of available cores is insufficient for the thread-migration-based SLICC to work well. A system can also
decide which method to use while exploiting the best of both worlds. Section 5.4 presents a mechanism
that dynamically selects between SLICC and STREX.

## 5.2 STREX

The key to the success of STREX is the ability to dynamically detect the points at which a transaction
is ought to be context-switched in order to keep inter-transaction execution synchronized, thereby max-
imizing instruction cache reuse. If a transaction executes for a long time, it will end up evicting cache
blocks that other transactions could have reused. If a transaction executes for a short time, the over-
heads of context switching and of a potential increase in contention in the data caches will overwhelm
performance. For these reasons, context switching at regular intervals would perform sub-optimally at
best. An optimal synchronization algorithm has to rely on dynamic information: a transaction should
be allowed to execute as long as it benefits from data and instruction locality, however, it should not be
allowed to evict any blocks that will be useful for other transactions. Moreover, care must be taken to
amortize the costs of context switching over the benefits gained as a result of the increase in instruction
reuse.

Figure 3.5 demonstrated significant temporal overlap among same-type transactions, but also showed
divergence that results from loops and conditional statements. Thus, breaking down the instruction
footprint of several transactions into smaller chunks will not generally result in identical code segments.
Optimally scheduling those chunks in order to maximize instruction locality is akin to job scheduling,
an NP-complete problem [43]. However, an optimal algorithm exists for the simpler case of a group
of identical transactions, i.e., transactions that follow the exact same execution path. STREX applies
this algorithm to the general case of partially overlapping transactions resulting in a simple, inexpensive
solution that performs well.

The rest of this section is organized as follows: Section 5.2.1 describes and demonstrates the potential
of the optimal algorithm for the special case of identical transactions. Section 5.2.2 presents STREX’s
core synchronization algorithm. Section 5.2.3 presents STREX’s implementation. Section 5.2.4 discusses
practical challenges and qualitatively compares STREX to SLICC and prefetching.

### 5.2.1 Optimal Synchronization for Identical Transactions

For the special case of multiple, perfectly overlapping transactions, there is an algorithm for detecting
the optimal context switching points. Let us consider the case of two identical transactions T1 and T2,
and let us assume that initially the L1-I is empty. T1 starts execution fetching instruction blocks. Since
the transactions are identical, all blocks touched by T1 will be touched by T2 in the exact same sequence.
T1 should continue execution up to the point where it would be forced to evict a cache block it brought
in. Evicting this block would force T2 to refetch it. Once T1 stops, T2 starts execution touching exactly
the same blocks that T1 did. All these blocks already reside the L1-I. We just finished one code segment
that was brought in by T1 and reused by T2. T2 is now about to evict a block from this segment. So, we are moving now to a new segment which may partially overlap with the existing one. As the discussion explains next, to make sure that we appropriately identify when this new segment ends we need to differentiate its blocks from the ones that already exist in the cache. Accordingly, we will use a phase tag to paint the blocks as they are touched. In more detail, T2 marks all blocks presently cached with a phase number, say 0. Then, T2 continues execution but it marks any block it currently touches with phase(1). T2 continues execution up to the point where it would be forced to evict a block marked as phase(1). Evicting such a block would force T1 to refetch it. Execution continues with T1, up to the point where T1 would now have to evict a block tagged as phase(1). T1 then proceeds to fetch additional blocks, marking them with phase(2). This process continues incrementing the phase number every time both threads have had a chance to execute with the same phase number.

More formally, the synchronization algorithm for \(N\) multiple, identical transactions is as follows:

1. The transactions execute in phases, starting with phase(0). The first transaction is deemed the lead.
2. At phase\((i)\), all cache blocks accessed by the lead thread are tagged with “\(i\)”, irrespective of whether they are in the cache or not.
3. A transaction, including the lead, executes phase\((i)\) as long as it does not evict cache blocks tagged with “\(i\)”.
4. When the last thread completes phase\((i)\), execution proceeds to phase\((i + 1)\).

**Potential for Identical Transactions:**

As Figure 5.2 shows, the synchronization algorithm has the potential to work well in practice. The figure shows the reduction in I-MPKI for the L1-I for TPC-C and TPC-E transactions. This experiment uses ten randomly chosen instances for each transaction type. Each of these instances is replicated ten times thus forming a hypothetical workload of 100 transactions. Figure 5.2 depicts the instruction misses with the optimal synchronization algorithm for each transaction type (Sync-Identical) compared to that of the baseline. The results demonstrate that STREX reduces I-MPKI significantly in all cases.

However, in practice and as the results of Section 3.2.3 suggest, rarely different instances of same-type transactions have completely identical instruction streams since their control flow depends on the data they process. Accordingly, the results of Figure 5.2 only serve as an indication of what may be possible.
5.2.2 STREX Synchronization Algorithm

The synchronization algorithm STREX uses to improve instruction cache reuse for the general case of multiple, non-identical transactions is as follows:

1. Given a pool of transactions, STREX groups transactions of the same type into *teams*. STREX places each team into a hardware thread queue in an available execution core. Then, it flags the first transaction in the queue as the *lead*.

2. STREX synchronizes transaction execution using a per core *phase*\(_{ID}\) counter. As a transaction touches instruction blocks it tags the block with the current *phase*\(_{ID}\) value no matter whether the access was a hit or a miss. Whenever the *lead* resumes execution, it increments the *phase*\(_{ID}\) counter.

3. STREX continuously monitors *victim* cache blocks. Upon encountering a victim block tagged with the current *phase*\(_{ID}\) value, STREX context switches the current executing transaction and places it at the end of the thread queue. The next ready transaction resumes execution.

4. If the *lead* transaction terminates, the next thread in the queue becomes the *lead*.

5. Threads keep running in a round robin order until they all complete execution.

6. Once all the threads in a team complete execution, the core becomes available for another team to execute.

Since transactions diverge at runtime (as Figures 3.4 and 3.5 show), the *lead* may not touch all blocks that the other, subsequent threads need. Hence the algorithm allows *non-lead* transactions to fetch new cache blocks.

5.2.3 Implementation

STREX’s implementation requires the following components per core: (a) thread execution queue, (b) a *phase*\(_{ID}\) counter, (c) a *phase*\(_{ID}\) tag per L1-I cache block, (d) a victim block monitoring unit, (e) a thread context switching unit, and (f) STREX’s control logic.

STREX tags all cache blocks with *phase*\(_{ID}\) values. These *phase*\(_{ID}\)s can be maintained separately in a table (PIDT) to avoid impacting the L1-I design and latency. The PIDT contains a *phase*\(_{ID}\) entry per cache block and is accessed in parallel with the L1 tag and data arrays. This work uses 8-bit *phase*\(_{ID}\) tags and an 8-bit, modulo *phase*\(_{ID}\) counter per core. The area overhead of the PIDT is small as it uses only eight additional bits per cache block. A PIDT does not contain any address tags or any additional block related information.

This work sizes the scheduling structures empirically in order for STREX to find a sufficient number of similar transactions to improve throughput. The OLTP system can provide up to 30 pending transactions at any given point in time\(^1\). Similar to SLICC-Pp (Chapter 4), STREX identifies similar transactions by examining the address of the header instructions, and groups them into *teams*. The maximum number of transactions that a team can have (*team\_size*) is fixed system-wide. STREX assigns teams in the arrival

---

\(^1\)Typical database systems allow one or two orders of magnitude more worker threads than the available processing cores to mitigate I/O related stalls [73, 107, 98].
order of the oldest thread in a team. When transactions that are not part of a team (stray transactions) become the oldest, they are scheduled individually.

Section 5.3.5 shows that by controlling the maximum allowed team size, STREX can trade-off between overall throughput and per transaction latency. Software database management scheduling schemes that batch transactions exhibit a similar tradeoff [55, 126, 154].

STREX context switches threads by saving and restoring their architectural state to/from the L2 cache slice nearest to the core. STREX requires support for hardware scheduling of multiple threads. Several proposals exist for implementing hardware-level thread scheduling and context switching [32, 124, 125]. STREX serves as additional motivation for further investigating how hardware-level thread scheduling ought to be supported.

5.2.4 Discussion

This section discusses some of the implications of STREX for corner cases, its overheads, and contrasts it against state-of-the-art instruction miss reduction techniques.

Forward Progress Guarantees

STREX’s effectiveness is limited by the amount of temporal overlap available across transactions of the same team. More precisely, the lead transaction has the largest impact on locality. For example, in a scenario where the lead transaction has minimal temporal overlap with the rest of the team, only the lead thread will make forward progress, while the others will have to wait until the lead finishes. There is no possibility of deadlock or starvation as the lead is guaranteed to finish, and in the worst possible scenario, the rest of the threads will become leads in order. Since the lead always starts execution with a new phaseID, it has the highest authority to evict cache blocks. If other threads do not touch these blocks and try to evict them, they will be context switched too early. Since STREX selects the lead randomly, that scenario is unavoidable. Yet, with our examined workloads, this scenario has never happened due to the inherent temporal overlap across transactions of the same type. An extension to STREX might investigate placing lower limits on the amount of forward progress a thread should make before context switching.

Context Switching Overhead

STREX incurs an overhead for context switching among team members. The architectural state of each transaction has to be saved and restored. In this work, thread contexts are saved in the second level cache to avoid thrashing the L1-D. STREX amortizes this overhead by improving instruction and data locality, which result in overall throughput improvement (see Section 5.3.3). A portion of the physical address space is reserved for storing thread contexts. For the workloads studied, context switches were sufficiently infrequent enough so that the overhead of context saving and restoring was never a significant fraction of the overall execution time. An implementation may choose to enforce a minimum number of instructions or cycles that a transaction ought to execute before a context switch is allowed.

Transaction Latency

Software transaction batching schemes [55, 126, 154] and SLICC improve performance while increasing transaction latency. STREX is no exception. STREX batches transactions into groups, and pipelines
their execution on a single core with the goal of reducing the total execution time for these transactions. Hence, the latency observed by a particular transaction includes a portion of the execution time of other transactions in the same group. In other words, a transaction’s latency is a function in the number of transactions within its group. Section 5.3.5 measures transaction latency as a function of STREX’s team size configuration parameter, suggesting its use to calibrate the tradeoff of throughput vs. latency, similar to the batch size parameter in a commercial DBMS, VoltDB [154].

Interaction with Prefetching

Prefetching reduces the latency observed for instruction cache misses by anticipating future instruction fetch requests. The simplest next-line prefetcher works well for streaming code regions. Other more sophisticated prefetchers can predict complex execution paths by recording previously seen instruction streams, and then prefetching cache blocks when a part of a recorded stream is touched again. Ferdman et al. propose prefetchers that record temporal streams of accessed instruction blocks, TIFS [39], or streams of committed instructions, PIF [37]. PIF is highly accurate for OLTP, but it requires resources that exceed that of a typical L1-I, e.g., \(~40KB\) per core. When compared to PIF, STREX is expected to require less bandwidth, area, and power. STREX improves locality by increasing instruction and data reuse, rather than by redundantly fetching cache blocks from the lower cache levels at every transaction. STREX and PIF are not two exclusive solutions to the same problem. PIF could reduce execution time for the lead transaction, thus improving performance when used in conjunction with STREX. On the other hand, STREX requires less cache bandwidth, and does not require PIF’s storage tables to be active all the time. Future work may investigate a possible combination of the two techniques.

Simultaneous Multithreading

Simultaneous Multithreading (SMT) improves transaction throughput by executing multiple transactions concurrently over the same core. SMT has been shown to improve performance for OLTP workloads but at the expense of additional L1 misses; on real hardware, 2-way SMT increases instruction (TPC-C 15% / TPC-E 7%) and data (TPC-C 10% / TPC-E 16%) misses [36, 144]. Moreover, the core remains idle 80% of the time suggesting that most stalls remain. Future work might look into using STREX to synchronize thread execution under SMT and thus improve locality and performance.

5.3 Evaluation

This section demonstrates experimentally that STREX reduces instruction stalls improving performance for OLTP workloads. Specifically, Section 5.3.2 demonstrates STREX’s impact on instruction and data misses as compared to SLICC [11]. Section 5.3.3 shows the throughput improvement with STREX in comparison with a Next-Line prefetcher [131], a state-of-the-art instruction prefetcher (PIF [37]), and SLICC. Section 5.3.5 highlights that STREX can be tuned to trade-off transaction latency for higher overall throughput. Section 5.4 presents and evaluates a solution combining STREX and SLICC. Section 5.3.6 discusses the hardware cost of STREX. Section 5.3.7 investigates the effectiveness of state-of-the-art cache replacement policies, and their interaction with STREX.
Chapter 5. STREX: Stratifying Transaction Execution

5.3.1 Methodology

The evaluation in this section replicates the methodology described in Section 4.3.1. With \( N \) cores, the Baseline architecture has \( N \) hardware contexts with the OS making thread scheduling decisions. SLICC in this section refers to the transaction type-aware variant which detects on-the-fly the transaction types using a dedicated scout core (SLICC-Pp, see Section 4.2.3). STREX detects transaction types in a similar way. STREX or SLICC form teams over a pool of up to 30 virtual contexts. Unless otherwise noted, each core maintains a thread queue of up to ten threads. STREX forms teams of up to ten threads whereas SLICC forms teams of up to \( 2N \) threads.

5.3.2 L1 Miss Rate

This section shows that STREX improves instruction and data locality consistently for OLTP workloads irrespective of the number of cores. Figure 5.3 reports the L1 I-MPKI and D-MPKI incurred by the Baseline system, SLICC, and STREX for two to 16 cores. For MapReduce, the I- and D-MPKI with STREX is within 1\% of the Baseline as context switches rarely occur for this workload. The next section shows that performance is virtually identical as well.

The I-MPKI of the Baseline is practically independent of the number of cores. This is expected as the Baseline makes no explicit effort to increase instruction reuse across threads; adding more cores improves throughput through increased parallelism. STREX consistently reduces I-MPKI over the Baseline with the I-MPKI remaining practically constant (the variation is less than 2\%) no matter how many cores are available. STREX reduces I-MPKI compared to the Baseline by an average of 37\%, 29\%, and 44\% for TPC-C-1, TPC-C-10, and TPC-E, respectively, and mostly independently of the number of cores.

Figure 5.3 shows that for the Baseline, data misses increase with the number of cores; more concurrency increases coherence misses, since transactions share data. Much of the sharing is due to same type transactions. They tend to access the same metadata and locks of the same tables, as well as the same index roots during index probes, and they tend to do so in the same sequence. STREX improves data locality by synchronizing their execution. The higher the data miss rate of the Baseline, the greater the D-MPKI improvements with STREX. For 16 cores, STREX reduces D-MPKI by 18\%, 35\%, and 11\%, for TPC-C-1, TPC-C-10, and TPC-E, respectively.

SLICC behaves differently than STREX improving instructions misses more, as more cores become available. When there are at most eight or four cores for TPC-C and TPC-E, respectively, SLICC does
Chapter 5. STREX: Stratifying Transaction Execution

5.3.3 Throughput

This section compares the overall throughput of STREX relative to the Baseline, a Next-Line instruction prefetcher [131], a state-of-the-art instruction prefetcher (PIF [37]), and SLICC. Figure 5.4 reports overall throughput normalized over the 2-core Baseline. STREX consistently improves throughput over the Baseline by an average of 41–55%, and by 19–32% over the Next-Line prefetcher, for 2–16 cores. Contrary to SLICC, STREX is insensitive to the number of cores and always improves performance.

SLICC either degrades or barely improves performance over the Baseline for the TPC-C workloads with up to eight cores and for the TPC-E workloads with up to four cores. For the same configurations the Next-Line prefetcher consistently outperforms SLICC. SLICC outperforms STREX and does so considerably only when there are at least eight and 16 cores respectively for TPC-E and for the TPC-C workloads. With 16 cores, SLICC outperforms STREX by 7%, 10% and 18% for TPC-C-1, TPC-C-10 and TPC-E. This result serves as motivation for Section 5.4 which presents and evaluates a hybrid system that combines STREX and SLICC. The system selects dynamically which method to use depending on the number of available cores and the cache demands of the target OLTP workload.

PIF is a state-of-the-art instruction prefetcher that has been reported to have nearly perfect instruction coverage of L1-I misses, and its performance is close to that of a perfect-latency cache which always responds with the hit latency [37]. Thus we model PIF as a 100% hit rate L1-I cache, with externally-observed demand traffic similar to the Next-Line prefetcher [37]. For 2–16 cores, STREX achieves on average 94% of PIF’s performance for TPC-C, and outperforms PIF by 9% for TPC-E, with less than 2% of the overhead storage.

MapReduce, which has an instruction footprint that fits in the L1-I cache, remains almost unaffected with all techniques.
5.3.4 Energy

This section reports the power, energy and energy-delay products for STREX compared against the no-prefetching Baseline and SLICC. Figure 5.5 plots the static and dynamic components of energy and power for 16-core systems. As illustrated earlier in Section 4.3.7 (which reports power and energy for SLICC), improving throughput increases core activity, thereby increasing dynamic power, however overall energy is reduced. STREX increases power by up to 40%, while reducing energy by up to 29%. STREX reduces ED and ED^2 metrics by up to 59% and 76%, respectively.

The relation between the dynamic power of SLICC and STREX varies for TPC-C and TPC-E; STREX’s dynamic power is higher for TPC-C, while for TPC-E SLICC’s dynamic power is higher. Several factors lead to these differences with the combination of L1 instruction and data misses contributing the most. The resulting overall energy for STREX is higher than SLICC by 15% for TPC-C, and lower by 2% for TPC-E.

5.3.5 Transaction Throughput vs. Latency

Similar to software transaction batching schemes [55, 126, 154] and SLICC (Section 4.4), STREX improves the overall throughput but may increase transaction latency. By adjusting the maximum number of transactions per team (team_size), it is possible to control this trade-off as Figure 5.6 suggests. Figure 5.6(left) shows the distribution of transaction latencies for TPC-C for the Baseline and STREX. For STREX the figure reports latency distributions as a function of team_size, noted as STREX-xT, where x is a team size in the range of two to 20. In all preceding experiments teams had up to ten threads. With STREX, the transaction latency is mostly independent of the core count, hence Figure 5.6 shows latencies only for the 16 core system. The trends are similar for TPC-E and the figure omits these measurements for clarity. Figure 5.6(right) plots the average latency for TPC-C and TPC-E normalized to the Baseline latency. The results show that the latency is almost linear in the size of teams (note that the Baseline can be thought of as STREX with team_size = 1).

A transaction’s latency is the number of cycles elapsed from the moment it enters the transaction queue until it completes execution. For STREX, as team_size increases, the distribution tends to move toward longer transaction latencies. Figure 5.7 shows the corresponding normalized throughput for TPC-C and TPC-E demonstrating that throughput also increases with the team_size. With up to 20 threads per team, throughput improvements are the highest at 59% and 80% over the Baseline, for
Figure 5.6: Transaction latency distribution (left) and average latency (right) as a function of \textit{team\_size}.

Figure 5.7: Overall throughput for a range of \textit{team\_size} values.

TPC-C and TPC-E, respectively. It would be straightforward to make the \textit{team\_size} configurable by the system, which can then set \textit{team\_size} according to its specific needs. This is similar to the request \textit{batch\_size} parameter in VoltDB, a commercial DBMS whose throughput and latency balance can be tuned \cite{154}.

### 5.3.6 Hardware Cost

Table 5.1 shows a cost breakdown of STREX’s hardware components. STREX utilizes two main units: a team formation unit and a thread scheduler unit. The team formation unit is used to group similar transactions into teams. In this work STREX searches through a window of 30 threads. The team management table maintains information about threads until they are dispatched to a core. Each entry consists of: a unique numerical ID, a type ID, a team ID, an index within a team, and a timestamp.

The thread scheduler unit is responsible for incrementing the \textit{phase\_ID} counter, tagging cache blocks with the current \textit{phase\_ID} value, keeping track of the lead thread, monitoring instruction cache block victims, and context switching threads. The thread queue is a circular FIFO buffer. Each entry consists of a unique ID, a pointer to the thread’s context in the L2 cache, and a lead flag bit. The size of the thread queue should be the maximum value allowed for the \textit{team\_size} configuration parameter. Most experiments set \textit{team\_size} to 10 with 20 being the maximum considered. Assuming one team management table per core, the total storage required per core by STREX is 665.5 bytes, in addition to the logic.
Table 5.1: STREX’s hardware component storage costs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Storage Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread Scheduler</strong></td>
<td></td>
</tr>
<tr>
<td>Thread Queue</td>
<td>20-entries (12-bit ID,</td>
</tr>
<tr>
<td></td>
<td>48-bits pointer to thread context,</td>
</tr>
<tr>
<td></td>
<td>1-bit lead flag)</td>
</tr>
<tr>
<td>phaseID Counter</td>
<td>8-bit</td>
</tr>
<tr>
<td>Auxiliary phaseID Table</td>
<td>8-bit per cache block (512</td>
</tr>
<tr>
<td></td>
<td>cache blocks)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5324 bits (665.5 Bytes)</td>
</tr>
<tr>
<td><strong>Team Formation</strong></td>
<td></td>
</tr>
<tr>
<td>Team Management Table</td>
<td>30-entries (12-bit ID,</td>
</tr>
<tr>
<td></td>
<td>32-bit timestamp, 4-bit type ID,</td>
</tr>
<tr>
<td></td>
<td>4-bits team ID, 8-bit team index)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1800 bits (225 Bytes)</td>
</tr>
</tbody>
</table>

Figure 5.8: State-of-the-art replacement policies.

### 5.3.7 Replacement Policies

This section studies the interaction of STREX with several state-of-the-art replacement policies, described in Section 2.1.3. OLTP workloads behave similarly to streaming applications and thus foil LRU replacement. BRRIP [70] was shown to reduce instruction misses by 8% on average, leaving ample room for improvement (Section 3.1.3). Figure 5.8 reports the I-MPKI for eight cores, with the following replacement policies: (1) LRU, (2) LIP, and BIP [115], (3) SRRIP and BRRIP [70], and (4) STREX coupled with LRU, BIP, and BRRIP.

Figure 5.8 shows that for the Baseline, BBRIP performs the best as it is specifically designed for streaming applications. However, STREX with LRU reduces I-MPKI more than 35% over the best replacement policy for TPC-C and more than 45% for TPC-E. When STREX is combined with the alternate replacement policies little improvement results if any. The alternate replacement policies try to keep the first segment in the cache and thus quickly force evictions when execution proceeds to the second segment with STREX. This results in much more frequent context switching and thus degrades performance.
Table 5.2: \textit{FPTable}: instruction footprint size per transaction in L1-I size units.

| TPC-C | Delivery = 12, New Order = 14  
|       | Order = 11, Payment = 14, Stock = 11 |
| TPC-E | Broker = 7, Customer = 9,       
|       | Market = 9, Security = 5,       
|       | Tr\_Stat = 9, Tr\_Upd = 8, Tr\_Look = 8 |

5.4 Combining STREX and SLICC

Data centers often vary their configuration at runtime to maximize the utilization of their resources, and to maintain QoS levels for different users. One such configuration option is changing the number of cores assigned to a particular application. Section 5.3.3 demonstrated that STREX performs much better than SLICC when the number of cores is not sufficient to hold the footprint of all transactions, whereas the opposite is true otherwise. Thus combining STREX and SLICC is a desirable option to enjoy the best of both worlds. A hybrid mechanism could make scheduling and migration decisions at the granularity of a code segment thus switching between SLICC and STREX depending not only on the specific number of available cores but also on the specific mix of transactions running at any moment. Such a mechanism could also take advantage of both inter- and intra-transaction overlap by using SLICC only for a portion of the transaction (e.g., the inner-loop in transactions of type \textit{New Order} as shown in Figure 3.4). This work considers a simpler alternative that instead dynamically chooses between STREX and SLICC for all transactions based on the aggregate L1-I capacity available at runtime. If there is enough aggregate L1-I capacity to fit the workload’s footprint, the system will decide to schedule all transactions using SLICC, otherwise, it will use STREX.

The mechanism operates as follows: (1) The system profiles the workload measuring the instruction footprint of each transaction type. The system records the measurements in a transaction footprint size table (\textit{FPTable}). The goal is to measure the average footprint size, in L1-I size units, for all transaction types. (2) When a transaction group is ready for scheduling, the system decides, based on knowledge of the available number of cores and the contents of \textit{FPTable}, whether to use STREX or SLICC. (3) \textit{FPTable} updates are triggered at: system startup, workload change, new transactions types, and system reconfiguration (e.g., increasing or decreasing the available cores); all rare events.

The aforementioned system requires a mechanism to measure the instruction footprint of each transaction type. This work implements this mechanism re-using STREX’s phase$_{ID}$ table while executing using SLICC. The system periodically switches to SLICC and executes a short profiling phase. In the profiling phase, a group of similar transactions is scheduled using SLICC. A random transaction is selected as a sample for this transaction type. The profiling phase counts the number of cache blocks touched by this thread on all cores. The profiling process works as follows. (1) All phase$_{ID}$ tables are reset to zero on all cores. (2) The sample thread is assigned with a non-zero phase$_{ID}$ value. (3) Any cache block touched by the sample thread is tagged with the pre-assigned phase$_{ID}$. (4) A cache block counter is incremented whenever the sample thread touches a block, and had to change its phase$_{ID}$ value. (5) This process repeats for different transaction types. The resulting cache block counts are rounded off to L1-I cache size units and are recorded in the \textit{FPTable}. Once the profiling phase completes, \textit{FPTable} holds the number of cores required by each transaction when running under SLICC. Section 5.4.1 evaluates the proposed hybrid solution.
Table 5.3: Extra hardware cost for the hybrid STREX/SLICC system.

<table>
<thead>
<tr>
<th>Hybrid System: SLICC’s Cache Monitor Unit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Missed-Tag Queue</td>
<td>60-bits</td>
</tr>
<tr>
<td>Miss Shift-Vector</td>
<td>100-bits</td>
</tr>
<tr>
<td>Cache Signature</td>
<td>2K-bits</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2208 bits (276 Bytes)</td>
</tr>
</tbody>
</table>

5.4.1 Evaluation of the Hybrid Solution

Table 5.2 reports the footprint values recorded in \(FPTable\), per transaction type during a profiling phase. For these experiments, the hybrid system updates the \(FPTable\) at system startup and on system reconfiguration, since the transactions’ footprints remain unchanged for a given system configuration. Thus, generating the \(FPTable\) once is sufficient. The profiling period is 0.2% of the overall execution time. When TPC-C and TPC-E run on more than 12 and eight cores, respectively, the hybrid solution selects SLICC, otherwise it selects STREX. Figure 5.4 reports the hybrid system’s throughput. The hybrid system selects STREX on 2–8 cores for TPC-C, and 2–4 cores for TPC-E, and SLICC when there are more cores. Accordingly, the hybrid solution matches the best performing scheduler per core count. For TPC-E on eight cores, three transactions require nine cores. With SLICC, these transactions incur a few extra misses, however, the resulting throughput is still slightly higher than STREX.

To accommodate SLICC, the hybrid mechanism requires extra hardware components. Table 5.3 reports SLICC’s cache monitoring unit components. The total storage for the hybrid system is 1166.5 bytes, per core.

5.5 Related Work

STEPS [55], a software solution whose approach is identical in spirit to STREX. STEPS relies on manual code instrumentation, which is a cumbersome task that requires high level of expertise, prone to many errors as it is manual, and results in code that is not portable since it is platform dependent. A slightly improved version, autoSTEPS, automates several components of the instrumentation process. However, the user must excite the DBMS through a client to determine where to place CTX calls manually. In addition, autoSTEPS may introduce races. In contrast, STREX is a programmer-transparent technique that behaves similar to STEPS, in terms of benefits.

Section 4.5 presents an overview of other prior art related to improving cache locality for OLTP workloads.

5.6 Concluding Remarks

This chapter presented STREX, a programmer-transparent, single-core technique that exploits the available temporal overlap among similar OLTP transactions. STREX groups similar transactions into teams, and time-multiplexes their execution on a single core, thus improving instruction and data locality. Ideally, a lead transaction encounters cache misses required to fetch a code segment, and its associated data, and the rest of the team hit on that code segment, and the shared data. As opposed to SLICC (Chap-
Chapter 5. STREX: Stratifying Transaction Execution

Strengthened by Chapter 4, STREX was demonstrated to be insensitive to the available number of cores, thus outperforming SLICC when the available aggregate cache capacity is not sufficient.

Experimental evaluation demonstrated the following. When compared to a conventional CMP, STREX consistently reduces L1 instruction misses by 37% on average, for 2 to 16 cores, and data misses by up to 35%. When compared to a state-of-the-art instruction prefetcher, PIF [37], STREX’s performance is within 6% or up to 9% better, with STREX having less than 2% of PIF’s storage requirements. When the core count is less than SLICC’s needs, STREX outperforms SLICC by an average of 14%. This work also presented a hybrid mechanism that incorporates both STREX and SLICC, and dynamically switches to the better alternative based on the footprint size and available core count. The hybrid solution closely followed the performance of the best-performing technique.

Future work might look into interactions between STREX and (a) instruction cache prefetchers, which can mitigate the overhead of re-filling the cache by the lead transaction, and (b) SMT, where STREX can be used to synchronize the execution of simultaneously executing transactions on the same core to minimize contention for the limited cache resources.
Chapter 6

EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

The previous three chapters addressed a crucial class of instruction-intensive workloads, Online Transaction Processing (OLTP), proposing techniques to mitigate front-end instruction-related stalls. This chapter tackles data-related stalls, specifically focusing on emerging data-intensive workloads with hard-to-predict access patterns.

The volume of digital data increases at a fast pace with 2.5 quintillion\(^1\) bytes produced daily [62]. The nature of this data is changing as well due to new services and applications such as Facebook and Twitter. Digital data is increasingly semi-structured or unstructured exhibiting intricate patterns. To make sense of this data deluge and to extract value from it, a series of so-called big data applications have emerged [157]. Such applications use algorithms from several domains such as data mining, machine learning, scale-free graphs, linear algebra, mathematical programming, and financial analysis. Most of these algorithms exhibit irregular, non-repetitive patterns generated by traversals of massive data structures, such as graphs and sparse matrices. As these applications operate on large volumes of semi-structured data they exacerbate the effect of the much slower main memory, a critical performance and power consideration in modern architectures. Worse, their irregular access streams tend to be hard to predict stressing existing data prefetching mechanisms.

Traditionally, prefetching has successfully hidden the much slower latency of main memory by fetching in advance the data the program may need. Numerous hardware and software prefetching techniques exist targeting various memory access patterns such as linear progressions [41, 61, 108], or repeating patterns in the access streams or in the relative distance of subsequent accesses [68, 74, 104, 132]. As this work corroborates, there are access patterns that elude such prefetchers.

A precomputation-based prefetcher (PbP) can in principle predict any arbitrary access pattern [8, 9, 26, 42, 82, 88, 93, 117, 123, 162, 164]. A PbP executes as a separate thread a precomputation slice (p-slice), a skeleton of the program that ideally is sufficient to generate the addresses for the most delinquent loads (DLIs), and do so early enough. For the most part, past PbPs favor p-slices with much fewer instructions than the corresponding main program slice. Using shorter p-slices was viewed as the key to allowing p-slices to run sufficiently ahead of the main program. Unfortunately, such p-slices sacrifice control flow and memory dependencies, and thus accuracy, triggering frequent aborts.

---

\(^1\)Quintillion = \(10^{18}\) in U.S., Canada and modern British.
and restarts. Specifically, past PbPs constructed p-slices either manually [164] or automatically using compiler techniques [82, 88, 117], or operated on application binaries [8, 9, 26, 42, 123, 162]. Compiler-based PbPs require source code, have to manage runtime overheads when orchestrating p-slice execution, and contend with lack of runtime information. Binary-based PbPs construct p-slices at runtime but have to contend with high monitoring overheads and reduced accuracy since their p-slices exclude all or part of the control flow. Past works suggested that constructing p-slices using just the most frequent path is sufficient, emphasizing p-slice speed at the expense of accuracy [8, 26, 162].

This dissertation revisits PbP design choices advocating a “slow and steady wins the race” approach showing that p-slices that are not much shorter than the main thread are able to run ahead in large datasets as long as the p-slice accurately predicts the relevant control flow. This observation motivates the Ekivolos PbP [10], EK for short, which automatically constructs highly accurate p-slices using just the application’s binary via a lightweight, trace-based approach. EK is closest in spirit to the work by Zilles and Sohi that manually constructed accurate p-slices [164], an onerous and error-prone task. Key to EK’s success is that p-slices only need to include the execution paths encountered on a test run of the application.

EK uses a two-phase, runtime system-supported approach, similar to runtime optimization systems for managed languages such as Java. (1) During the first offline profiling phase, a software runtime system profiles the application’s instruction stream capturing traces leading to DLIs and constructs control-flow and memory-dependence inclusive, self-sufficient p-slices. The binary is enhanced with these p-slices along with a set of triggering instruction addresses. (2) When encountering trigger DLIs during execution of the enhanced binary, a hardware component launches p-slices on prefetching cores. The p-slices run ahead independently of the program generating accurate and timely prefetches improving performance and energy. Introducing such hardware assists is now more likely, if not a necessity, as Dennard scaling has ceased [21, 35].

EK uses a novel p-slice construction algorithm and borrows from past proposals [26, 82, 103, 162], resulting in a unique combination of features. EK constructs p-slices dynamically which include enough control flow, including biased branches, in order to faithfully replicate the original program’s execution, similar to compiler based methods [82]. EK’s p-slices include memory store operations similar to run-ahead execution [103] benefiting from stable memory dependencies. However, EK operates on application binaries [26, 162], and constructs p-slices dynamically allowing it to be more aggressive in folding branches, and circumventing memory dependencies that are typically treated very conservatively when done statically [82, 158].

EK proves effective on various data access patterns representing emerging data-intensive workloads. Compared to a PbP that constructs single-path p-slices similar to past work [26, 162], EK’s prefetches are 3× more accurate. Over a baseline with no prefetching, EK reduces L3 misses in the range of 0% to 99%, improves performance by 0 – 12×, and reduces energy by 22% on average. Depending on the workload, a system with EK is faster and more energy efficient than one using a simple linear prefetcher, SMS [132], AMPM [67] or PC/AC [104]. EK’s prefetching cores can be shut off or perform other computation when not used for prefetching.

The conclusions drawn are: (1) P-slices do not have to be much shorter than the main program to be effective. (2) Including just enough control flow and data dependencies in p-slices results in accurate and timely prefetching requests for workloads that past single-path p-slices are ineffective. (3) A simple

---

2From ἐκηβόλος, shooting far and accurately, Iliad, Homer.
novel algorithm that observes the execution traces that lead to DLIs during a profile run can construct such p-slices that are generally leaner than those a static p-slice analysis would produce. (4) A simple, stripped down in-order processing core is a sufficient p-slice execution engine.

The rest of this chapter is organized as follows. Section 6.1 motivates control flow inclusive p-slices. Section 6.2 describes EK’s design. Section 6.3 considers several workloads identifying which are memory bound and how they interact with existing approaches. Sections 6.4 and 6.5 present the experimental methodology and results highlighting EK’s strengths and weaknesses. Section 6.6 reviews prior work, and Section 6.7 concludes.

6.1 Motivating Example

For precomputation prefetching slices to be effective on certain workloads, considering just the dominant execution path is insufficient. Consider sparse vector-matrix multiplication (SpVM), a kernel essential in emerging application domains such as machine learning [17, 28]. SpVM performs poorly on modern architectures with performance as low as 10% of peak [156]. SpVM may, at first glance, be dismissed as having an innocuous memory access pattern, a justified reaction if it were not for the matrices being sparse. Figure 6.1 shows a simplified implementation where the $n \times 1$ vector $V$ and the $n \times m$ matrix $M$ use the Compressed Sparse Row (CSR) format, a preferred format when the sparseness degree is undetermined in advance [155]. The outer loop scans $V$ one non-zero element at a time, while the inner loop iterates over the non-zero elements of the corresponding row $M[\text{outer}][\text{inner}]$ multiplying with $V[\text{outer}]$, accumulating this partial product to $RV[\text{inner}]$.

As Figure 6.2 shows, $V$ is represented as two arrays: $Vv$ containing, in index order, the non-zero values ($sv\_val$), and $Vi$ ($sv\_idx$) storing their column indexes. Similarly, $M$ is represented by three arrays: (1) $Mv$ containing in row-major order the non-zero values ($sm\_val$), (2) $Mi$ containing the corresponding column indexes ($sm\_idx$), and (3) $Mb$, a row beginnings array ($sm\_beg$) pointing to where each of the $n$ rows start in the other two arrays.

Figure 6.2 shows the accesses for two outer loop iterations. The inner loop indexes $Mb$ using $Vi$ to find where in $Mv$ and $Mi$ the corresponding $M$ row is stored. It reads the row elements, an element per iteration using $Mi$ to index and update an $RV$ element. When executed (see methodology in Section 6.4), the three loads accessing (1) $Mv$ (Figure 6.1, line 15), (2) $Mi$ (line 14), and (3) $RV$ (line 16) prove the most problematic. Figure 6.1 reports their average memory latencies assuming zero queuing delays, with the $RV$ DLI being the slowest.

Only the accesses to $Vi$ and $Vv$ are linear. The accesses to $Mb$ depend on $Vi$’s values, those to $Mi$ and $Mv$ depend on $Mb$ and those to $RV$ depend on $Mi$. The accesses to $Mi$ and $Mv$ thus form a progression comprising bursts of linear sequences with the distance between bursts depending on the $Vi$ values. The $RV$ access sequence depends on the values of $Mi$, $Mb$ and $Vi$. Any regularity in the resulting $RV$ access stream will be the result of happenstance for sparse matrices. As Section 6.5 demonstrates, SpVM’s access pattern eludes two state-of-the-art hardware prefetchers.

As density increases, the more elements of $RV$ will be accessed and the more the resulting pattern resembles a linear stream that can be effectively prefetched even with a stream prefetcher. Section 6.5.3 presents a sensitivity analysis with respect to the density of the matrix showing that SpVM eludes state-of-the-art prefetchers for matrix densities of less than 10%. Such densities are found in Webgraph [22], circuit simulation, DNA analysis, social networks, graph partitioning and clustering [29] and
fluid dynamics applications [31].

6.1.1 Precomputation Prefetching to the Rescue

Precomputation-based prefetchers (PbPs) try to compute the address of an access instead of predicting it. They construct a precomputation slice (p-slice) – a slice of code extracted from the program that calculates the target load’s address – which is launched concurrent to the main thread. If the p-slice is accurate and timely, it can prefetch data effectively.

Challenges with Single-Path P-Slices

A single-path PbP constructs p-slices considering only one execution path, whereas a multi-path PbP considers multiple paths. Single-path PbPs typically extract p-slices over a single dominant trace during
execution. Constructing optimized, fast running p-slices was seen as the key in being able to run ahead of the main thread. Past works opted to omit all or part of the control-flow from p-slices and to mitigate the resulting p-slice inaccuracy, they relied on monitoring mechanisms to reactively stop, restart or revise p-slices [9, 26, 162]. This approach proved sufficient for the workloads studied.

A p-slice lacking control flow cannot accurately prefetch SpVM. Figures 6.1(b) and 6.1(c) show respectively the dominant execution trace (the inner loop) and the corresponding p-slice. The p-slice increments the matrix index \( m_{\text{idx}} \), loads the result array index \( \text{res}_{\text{idx}} \), and reads the result element \( \text{res}[\text{res}_{\text{idx}}] \). This p-slice accurately prefetches the elements of a single \( M \) row, but, due to lack of control flow, it will continue to prefetch subsequent \( M \) rows, even though they might not be needed because the corresponding \( V \) elements are zero. A multi-path PbP would naturally include the control flow instructions needed.

**P-Slice Accuracy and Effectiveness**

Figure 6.3 explains via an execution sketch, how shorter yet inaccurate p-slices can be ineffective, while longer but accurate p-slices can run ahead of the main thread. The main thread is shown to progress at a steady pace as represented by the dark continuous line (in practice the main thread will accelerate due to prefetching). The shorter, inaccurate p-slice is restarted every time it goes astray resulting in several “acceleration” phases where it starts at the same point as the main thread, quickly running ahead. Soon thereafter it is stopped and restarted resulting in a sawtooth pattern. A “slower” more accurate p-slice, will slowly yet successfully run far ahead (see Section 6.2.5) reaching the maximum allowed run-ahead distance, if it exists, issuing timely prefetches. This simplified sketch illustrates that accuracy can be more important than acceleration.

**Multi-Path Precomputation Prefetching**

This is not the first work to recognize the need for multi-path p-slices. Past work constructed such p-slices using either compiler techniques [82, 93, 117] or manually [164], or using only a subset of the control flow. Compiler techniques operate on the application’s source code and typically incur heavy overheads at runtime as they modify the original thread to orchestrate p-slice execution. Manual p-
Chapter 6. EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

Figure 6.3: A sketch displaying the relation between the program’s main thread (dark), and concurrently running p-slices. Accurate slower p-slices (blue) can be more effective than faster inaccurate p-slices (red), if the latter is frequently restarted. $\theta$ represents acceleration.

Slice construction is an onerous and error-prone task. Alternatively, Illusionist constructs control flow inclusive p-slices by operating directly on application binaries, however it omits biased branches in an attempt to build shorter p-slices [9]. The SpVM example shows that omitting such branches would lead to ineffective p-slices. The need for a light-weight binary-only approach that does not omit branches motivates EK which is described next.

6.2 Ekivos Prefetch System

EK uses a two phase approach: First, the application is profiled identifying its DLIs and constructing the appropriate p-slices. Second, any subsequent run can use the p-slices to prefetch data from memory. The rest of this section details (1) the profiling phase, (2) the prefetching subsystem and how prefetching unfolds, and then (3) discusses EK’s limitations and design choices.

6.2.1 P-slice Construction Process

We initially assume a single DLI and detail (a) the DLI selection metric, and (b) the p-slice construction algorithm, concluding with (c) support for multiple DLIs.

Identifying DLIs

Different metrics were proposed to identify DLIs [26, 88, 93, 100, 117, 162, 164]. At each profiling run, EK selects the load with the longest total memory delay among those whose average latency is higher than the last level cache latency. Average latency indicates whether the load goes off-chip while a load’s total memory delay, the sum of the latencies observed by its instances, is a proxy for its relative performance impact. All latencies are measured using hardware support [26, 93, 162].

P-Slice Construction Algorithm

Figure 6.4 shows the steps of p-slice construction: (1) Collect all unique traces, or p-traces, leading to the DLI as they appear. (2) Trim unnecessary branches. (3) Construct a p-slice for each p-trace.
Chapter 6. EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

(4) Remove unnecessary instructions. (5) Merge the per p-trace p-slices. (6) Map instruction addresses and branch targets into a p-slice address space. The rest of this section details each step.

Step 1. P-Trace Collection: A p-trace is the sequence of instructions between two subsequent occurrences of the DLI. EK collects all unique p-traces as encountered. For the simplified SpVM, there will be two p-traces: one containing the inner loop iteration only (Figure 6.1(c)) and another containing also the outer loop (Figure 6.1(b)).

Step 2. Conditional Branches: For each encountered branch, EK considers the directions it followed in all p-traces excluding always-not-taken branches (e.g., 0x9586) and converting always-taken branches to unconditional branches [44].

Step 3. Per P-Trace P-Slice Construction: For each p-trace, EK identifies the instructions that should be included in the p-slice via a data-dependency-only backward slicing algorithm [93, 162] starting from all branches left in Step 2 and the DLI. As memory dependencies tend to be stable, EK follows an approximate approach and includes only those stores that have a dependent load in the same p-trace [44]. The instructions marked by this step are tagged with an asterisk (*) in Figures 6.1(b) and 6.1(c). This slicing algorithm may include branches that do not affect the DLI. Section 6.5 shows that this simplified approach was sufficient for the studied workloads. A static backward slicing algorithm [140] will produce sparser p-slices when applied on the observed program dependence graph (PDG) which considers only the control-flow edges and memory dependencies that were observed in the p-traces. Generally, this observed PDG will be a subset of the PDG which considers all execution paths and possible dependencies [40].

Step 4. Per Trace P-Slice Optimization: P-slice optimization has been studied extensively [162]. We apply two optimizations: (1) Eliminate stack push/pop instructions pairs for registers that have not been modified in-between, and (2) combine back-to-back unconditional branch instructions. We do not apply further optimizations potentially underestimating the benefits of EK. This simplifies p-slice
contribution and can prove useful if a hardware only approach was to be followed.

**Step 5. P-Slice Merging:** The per-p-trace p-slices are merged into one p-slice using a sorted merge on the instruction address space on all p-slices.

**Step 6. Remapping Instruction Addresses and Determining Exit Points:** Finally, EK maps the p-slice instruction addresses and branch target addresses into a separate p-slice address space. Since the p-slice omits instructions, this step is necessary for correct sequencing during p-slice execution. Figure 6.1(e) shows the final p-slice. Branches with targets not in the p-slice form exit points and their target is mapped to a reserved end-of-execution address (e.g., instruction 0x0036).

### Multiple DLIs

Currently, EK follows an automated iterative approach processing a single, additional DLI per profiling run. Profiling repeats until no further DLIs can be found or when the performance has plateaued. This iterative approach reduces the number of prefetched DLIs as several loads may depend on one DLI. Processing at each step multiple DLIs whose p-traces span non-overlapping code regions can further reduce the steps needed. Using shorter running inputs, as done in this work, or sampling, results in shorter profiling runs.

Given the already processed DLIs, a new DLI can be dependent or independent and its p-traces may share some instructions with existing p-traces. DLI $Y$ depends on DLI $X$ if $Y$ uses $X$’s value. Naturally, a p-slice which targets $Y$ will include $X$. Instructions 0x956e and 0x9576 in Figure 6.1(c) are dependent. Independent DLIs, such as instructions 0x9572 and 0x9576, are included iteratively, one at a time. For each independent DLI whose p-traces do not overlap with any other DLI, EK creates a p-slice with the DLI being the trigger instruction. EK merges overlapping p-slices, repeating Steps 5 and 6 while considering all p-traces. The resulting p-slice will have multiple triggering DLIs. Figure 6.5 depicts the different scenarios of multiple DLIs and how EK handles each.

This section presented EK’s p-slice construction algorithm and demonstrated how it captures control-flow and memory dependencies, and how it deals with multiple DLIs. Next, Section 6.2.2 describes the overall prefetcher design and discusses EK’s limitations. Sections 6.4 and 6.5 evaluate EK experimentally.

### 6.2.2 Overall Architecture and Execution

EK uses a p-core, a separate general-purpose processing core to execute the p-slices. No effort is made to optimize the p-core in this work as the goal is to show that an accurate p-slice is more effective than a shorter one. A simple in-order core supporting a subset of the main core’s (m-core) instruction set using

<table>
<thead>
<tr>
<th>Dependent DLIs (Y depends on X)</th>
<th>Overlapping P-slices</th>
<th>Non-overlapping P-slices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>P-slice of Y must include X</strong></td>
<td><strong>N/A</strong></td>
</tr>
<tr>
<td>Independent DLIs</td>
<td>Merge two P-slices into one. Two trigger DLIs for a single p-slice.</td>
<td>Two p-slices, one trigger for each.</td>
</tr>
</tbody>
</table>

Figure 6.5: Taxonomy of DLI dependency vs. and p-slice overlaps.
a small local cache proves sufficient. Further p-core enhancements (e.g., leader-follower TLB [19]) or organizations warrant further attention in a broader context (e.g., [83]). Our goal here is to demonstrate EK’s effectiveness for one reasonable configuration. The p-core’s local cache prefetched into the m-core’s L2.

To execute a p-slice, EK must: (a) determine when to launch it, (b) handle p-slice stores, (c) control the run-ahead distance, and (d) restart execution if needed. Figure 6.6 depicts the lifespan of a p-slice. When the m-core executes a trigger DLI, EK spawns the corresponding p-slice on the p-core. M-core register and L1 TLB contents are copied to the p-core which starts by executing the instruction directly following the DLI. As long as the p-slice is running on the p-core, further instances of the same DLI on the m-core will not trigger another p-slice launch. Only one p-slice runs at any given point of time.

P-core stores write their value in a fully-associative LRU, byte-addressable local store buffer (LSB) that takes precedence over the rest of the memory hierarchy for p-core loads. A 64-Byte LSB was sufficient. The LSB contents are discarded when the p-slice exits.

Prefetching too early can hurt performance either by evicting prematurely other useful data or by having the prefetched blocks not survive long enough. EK employs a run-ahead saturating counter (RAC) which it increments for p-core prefetches, and decrements for m-core loads. A saturated RAC throttles the p-core. Figure 6.3 depicts the resulting two phases of p-slice execution: acceleration and run-ahead steady state. Initially, the p-slice “accelerates” running ahead. Once the p-core reaches a predetermined run-ahead distance it throttles, slowing down maintaining a mostly constant run-ahead distance. After the p-slice terminates, and the m-core has consumed all prefetches (i.e., RAC equals zero), EK may spawn a new p-slice. Ignoring the RAC would often trigger a p-slice for the same DLI that would just re-generate the addresses of the p-slice that just terminated. The RAC auto-resets when its value freezes at a non-zero value for more than 100K cycles, an empirically chosen value. While not observed in our experiments, this can happen if the p-slice prefetched more items than it should have.
Comment on Using Ekivolos with SMT

A common PnP configuration executes the main thread and the corresponding p-slice on the same core which supports Simultaneous Multi-Threading (SMT) [26, 88, 94, 162, 164]. In such configurations, the p-slice shares execution resources, and higher cache levels and TLBs with the main thread. Consequently, the effectiveness of EK on SMT depends on the amount of destructive interference that may happen as a result of this resource sharing. For the memory-bound workloads this work targets, utilization of execution resources is typically low, enabling a p-slice to execute smoothly without impacting the main thread. However, the sharing for the L1 data cache and TLB is more concerning. On one hand, the p-slice tries to run as far ahead of the main thread as possible to effectively hide the access latency of the target DLI. On the other hand, prefetching too much data may pollute the relatively small L1 cache and TLB. Hence, executing EK on SMT requires some experimental evaluation with respect to the saturation value of the RAC (i.e., maximum run-ahead distance), and the impact on cache and TLB misses.

6.2.3 Constructing P-slices Dynamically

EK constructs p-slices using dynamic information, thus allowing it to be more aggressive in folding branches and circumventing memory dependencies that are typically treated conservatively when done statically [82, 158]. For example, in SpVM, all conditional branches will be treated equally and the branches corresponding to array-boundary error checking (0x9586 and 0x956c) will be included, together with their leading instructions, in such p-slices. In practice, these branches are rarely executed, if ever. In contrast, EK will recognize during profiling the insignificance of these branches and ignore them (Section 6.2.1, Step 2) [50, 117]. Out of the 16 instructions which comprise the dominant execution path in SpVM (Figure 6.1(c)), EK’s p-slices include 7 instructions, while their static counterpart will include 13 instructions.

6.2.4 When and How Can the P-slice Run Ahead

Two conditions enable the p-core to run ahead of the m-core. (a) The DLI is not on the p-slice’s critical path. (b) The main thread’s critical path includes other instructions that are not included in the p-slice, which slow down the m-core with respect to the p-core.

The main thread includes DLI-dependent instructions, those which depend on the target DLI’s outcome, thus putting the long-latency DLI on the critical path. A p-slice which omits DLI-dependent instructions will not block on the DLI. The SpVM code exemplifies this condition: while Figure 6.1(c) shows that for the main thread, the target DLI (0x9576) produces a value which has a consumer (0x957a), Figure 6.1(e) shows that the p-slice omits this consumer. Hence, while the m-core may stall on the target DLI, the p-core can pseudo-retire the DLI [163] and can advance to subsequent loop iterations, virtually treating the DLI as a non-blocking prefetch.

Other reasons may cause the m-core to stall allowing the p-core to run ahead, even if the DLI is on the latter’s critical path. Consider the pointer-chasing access pattern in graph500, where the p-slice executes DLI-dependent instructions. The m-core however executes a number of memory operations which clog up its load-store queue buffers. The p-slice omits these instructions, gaining a relative execution speed advantage.
The p-slice may fail to run sufficiently ahead in at least the following two cases: (1) For *effectively dense* p-slices that cannot accelerate ahead of the main thread as their latency is the same as that of the main thread. This happens when the p-slice does not omit any instructions or when there is enough instruction level parallelism in the original stream so that the m-core can hide the latency of all the instructions that are not in the p-slice. In some cases with multiple DLIs, EK could remove some of these DLIs resulting in a leaner p-slice. (2) For p-slices that terminate before running ahead by a sufficient number of iterations which happens when the DLI does not execute a sufficient number of times. This scenario can be detected in the profiling phase or at runtime by measuring the DLI latency. This behavior appears in *canneal* (see Section 6.5.1). Unless remedy is possible, EK should not create p-slices for such DLIs.

6.2.5 Limitations

EK strives to construct p-slices that are accurate and timely but it will not always succeed. The key assumption in EK is that the memory dependencies and control flow behavior observed during profiling is representative of the actual runs. For example, if an execution path was never excited during profiling (e.g., code dealing with corner cases), the constructed p-slice(s) will exclude that path. This may affect EK’s prefetch accuracy, depending on the frequency of the missed path, and whether the address calculation on the included paths depends on that missed path. That said, for the evaluated workloads, relatively small profiling inputs were sufficient to capture all execution paths. Inaccurate p-slices should ideally be detected and fixed by repeating the p-slice construction. The following are some indicators of inaccuracy: (1) A p-slice frequently stopping/restarting such as a p-slice stopping earlier than it should. (2) A p-slice that is stuck in the run-ahead state without forward progress. This happens when the main thread has exited the loop including the target DLI. (3) The target DLI latency remains high indicating that the p-slice is failing to issue correct prefetches.

While automated, EK requires an iterative offline profiling-based process and needs to augment the execution binary. The cost of this process is low and amortized over multiple, longer runs of the program. Complexity is $O(n \times t)$; where $n$ is the number of DLIs and $t$ is the time of a single profiling run. For the evaluated workloads we found $n < 10$ and $t < 1B$ CPU cycles (except SVM-RFE $t = 3.7B$). An online runtime approach where p-slices are extracted and deployed on-the-fly warrants a separate study due to challenges including: (1) Slice construction overhead, and (2) incrementally adding control-flow paths.

EK uses a hardware p-core as opposed to a pure software approach [83]. While appearing heavy handed, such p-cores may already exist for other types of prefetching [83]. A pure software approach faces two challenges: (1) Performing store operations in the p-slice to maintain memory dependencies – implementing those stores in software is not obvious. (2) Software thread spawning imposes overheads on the main thread.

Techniques which use an extra execution context [9, 26, 42, 88, 162, 164] are seen as extravagant. However, improving single-threaded performance is sometimes preferred over parallelism. For example, when it becomes increasingly difficult to parallelize kernels or distribute data, as with Google’s workloads [101]. Furthermore, Section 6.5.6 shows that a simplified p-core suffices.
6.3 Workloads

EK targets memory-bound workloads that exhibit irregular, hard-to-predict off-chip memory accesses. Unfortunately, few of the well-established workload suites that are used for architectural studies are memory-bound while not being predictable by existing hardware prefetchers. Moreover, there are application domains that are increasing in importance that are not well represented in any one benchmark suite. For this reason, we considered a variety of workloads: (1) All PARSEC 3.0 applications [20]. (2) MCF, LBM, Soplex, three of the most memory-bound SPEC2006 [57, 69] applications using the reference dataset. (3) SVM-RFE a memory-bound data mining benchmark from NU-MineBench [114]. (4) Cholesky [30] and Hashjoins [15] two commonly used memory-bound kernels. (5) Graph500, radix, SpVM, and SSCA2 that are representative of emerging memory-bound algorithms and similar to those included in MachSuite [120] a suite targeting accelerators. MachSuite targets small-scale embedded applications and thus limits the applications to 32KB memory footprints. We consider equivalent implementations using larger datasets [13, 48, 49].

Figure 6.7 reports the L3 MPKI for these workloads for a baseline system without prefetching (NoPref), a simple linear prefetcher (NL), or with the state-of-the-art prefetchers, AMPM [67] or SMS [132]. Section 6.4 details the experimental methodology and the configurations used. Several workloads either exhibit low L3 MPKI or can be serviced well by an existing prefetcher. However, there are workloads that foil existing prefellers and that exhibit relatively high L3 MPKI: canneal, freqmine, MCF, graph500, SpVM, SSCA2, and SVM-RFE. The rest of this work considers these applications along all those that exhibited a noticeable L3 MPKI on the baseline system. Table 6.1 briefly describes these workloads and comments on their access patterns. Collectively they exhibit a variety of access patterns and thus have the potential to expose the strengths and weaknesses of the various prefellers.

6.4 Methodology

6.4.1 Experimental Setup

All experiments use ESESC [77], an ARM 32-bit ISA cycle-accurate processor simulator that uses a modified QEMU [16] as its emulation engine. All workloads were compiled using gcc or g++ v4.7.3 using the -O3 optimization level. Table 6.2 details the baseline system architecture and in parentheses
Table 6.1: Workloads and their access patterns.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes [20]</td>
<td>calculates the prices for a portfolio of stock options analytically with the Black-Scholes partial differential equation. It maintains two sets of arrays, which are accessed linearly. Overall, it is dominated by floating-point operations, not memory accesses.</td>
</tr>
<tr>
<td>Canneal [20]</td>
<td>uses cache-aware simulated annealing to optimize the routing cost of a chip design. Simulated annealing approximates the global optimum in a large search space. Canneal pseudo-randomly picks pairs of elements and tries to swap them resulting in a random access pattern.</td>
</tr>
<tr>
<td>Cholesky [30]</td>
<td>factorization of a positive-definite matrix into a lower triangular matrix and a conjugate transpose, is used for efficient numerical solutions and Monte Carlo simulations, a common method in optimization, numerical integration and probability distribution problems. The access pattern consists of both linear and random accesses.</td>
</tr>
<tr>
<td>Facesim [20]</td>
<td>takes a model of a human face and a time sequence of muscle activations and computes a visually realistic animation of the modelled face by simulating the underlying physics. The computation solves non-linear systems of equations, represented as matrices with linear accesses.</td>
</tr>
<tr>
<td>Ferret [20]</td>
<td>a content-based similarity search of feature-rich data such as images. The algorithm segments the input image into feature vectors, which are used to search an indexed database. Image segmentation dominates execution, and exhibits data locality within the image.</td>
</tr>
<tr>
<td>Freqmine [20]</td>
<td>Frequent Itemset Mining that identifies patterns in a transactional database. The database is represented by a prefix tree and two sparse lookup tables (table entries may include linked lists). The access pattern includes linear table scans, and linked list and tree traversals.</td>
</tr>
<tr>
<td>Graph500 [49]</td>
<td>constructs a graph and performs a breadth-first search to construct sub-graphs within the larger graph. Representative of computations used in cybersecurity, medical informatics, data enrichment, social networks, and symbolic networks. We target the breadth-first search component as the graph construction phase is synthetic. It performs pointer-chasing over a multi-dimensional data structure.</td>
</tr>
<tr>
<td>Hashjoins [15]</td>
<td>an optimized hash joining kernel configured to use the histogram-based joining algorithm. The input relation is first scanned to obtain a histogram over the hash values. Then, a prefix sum is used to help re-order relation $R$ (to obtain $R'$), such that tuples with the same hash value appear contiguously in $R'$. The access pattern of the re-ordering phase is random, but, the initial table scan and the joining are linear.</td>
</tr>
<tr>
<td>MCF [57]</td>
<td>performs single-depot vehicle scheduling in public mass transportation. It solves an optimization problem trying to reduce the number of vehicles required to cover certain aggregate trip requirements. The access pattern consists of chasing pointers with variable stride deltas.</td>
</tr>
<tr>
<td>Radix Sort [48]</td>
<td>an optimized implementation with a linear access pattern.</td>
</tr>
<tr>
<td>SpVM</td>
<td>Sparse vector-matrix multiplication is used to solve large, sparse, linear systems of equations and has many applications, e.g., in machine learning [17, 28]. SpVM is an implementation based on the Compressed Sparse Row format [155]. Unless otherwise mentioned, the input matrix is an $18M \times 18M$ web-page graph with an average of 16 non-zero elements per row [22]. The access pattern is a mix of fragmented linear and data-dependent accesses (see Section 6.1).</td>
</tr>
<tr>
<td>SSCA2 [13]</td>
<td>a graph computational kernel with applications in computational biology, network analysis, and security. It builds on top of graph500 to measure betweenness centrality; a measure of a node’s centrality in a network. The algorithm iteratively traverses a few nodes, performs some computation, then traverses more nodes. SSCA2’s overall access pattern is different from graph500 since it repeats several steps only one of which is a graph traversal.</td>
</tr>
<tr>
<td>Streamcluster [20]</td>
<td>clusters continuously streaming data to their nearest center. It is common in network intrusion detection, pattern recognition and data mining. The program is memory bound for low-dimensional data and becomes computationally intensive as the dimensionality increases.</td>
</tr>
<tr>
<td>SVM-RFE [114]</td>
<td>Support Vector Machines - Recursive Feature Elimination is a feature selection algorithm from NU-MineBench, a data mining benchmark suite. It is an iterative algorithm that works backward from an initial set of features to fit a linear SVM, rank the features based on their weight, and then eliminate the lowest-weight feature. The initial scan of features is a linear, however the ranking process embodies hard-to-predict accesses.</td>
</tr>
</tbody>
</table>
Table 6.2: System parameters.

<table>
<thead>
<tr>
<th>Baseline</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Core</strong></td>
<td>(6.5mm²)</td>
</tr>
<tr>
<td></td>
<td>64 L1-TLB, 512 L2-TLB, OGEHL* (10 tables)</td>
</tr>
<tr>
<td></td>
<td>IL1$: 32KB, 64B blks, 4-way, 2-cycles</td>
</tr>
<tr>
<td></td>
<td>DL1$: 32KB, 32B blks, 8-way, 4-cycles</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>256KB, 64B blks, 16-way, 7-cycles, MESI</td>
</tr>
<tr>
<td><strong>L3 Cache</strong></td>
<td>2MB, 64B blks, 32-way, 14-cycles</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>FR-FCFS**, open-page, 8M × 16 × 8 banks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prefetching Engines</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P-Core</strong></td>
<td>(4.2mm²)</td>
</tr>
<tr>
<td></td>
<td>64 L1-TLB, OGEHL* (10 tables)</td>
</tr>
<tr>
<td></td>
<td>Unified L1 I/D$: 4KB, 32B blks, 4-way, 3-cycles</td>
</tr>
<tr>
<td><strong>SMS[132]</strong> (1.4mm²)</td>
<td>DL1$: 512B regions, 4 × 16 AGT, 256 × 8 PHT</td>
</tr>
<tr>
<td><strong>AMPM[67]</strong> (1.5mm²)</td>
<td>DL1$/L2$: ASP: 16 filters, 16 histograms</td>
</tr>
<tr>
<td></td>
<td>L3$: AMT: 52-entry, 16KB zones</td>
</tr>
</tbody>
</table>

* Optimized GEometric History Length branch predictor [127],

** First-Ready First-Come-First-Serve scheduler [122].

the area for each major component for a 22nm technology as estimated by McPAT 1.2 [87]. McPAT was shown not to account for area of major core components such as the Instruction Fetch, Instruction Scheduling and Load Store units [161]. Hence the absolute area numbers reported for the Main Core and P-Core must be used with caution. Energy was also estimated using McPAT 1.2. Table 6.3 lists the profiling and measurement input datasets. Unfortunately only one dataset was available for SVM-RFE. Execution measurement samples were taken after skipping the initialization.

6.4.2 Evaluated Prefetchers

The following prefetch engines are considered: (1) EK, (2) DPP, a PbP representative of prior work [26, 162], (3) NoMEM, a stripped-down version of EK which does not maintain memory dependencies, (4) Next-Line, a simple linear prefetcher, (5) SMS, an address-correlation prefetcher that has been shown to work well for commercial workloads [132], (6) AMPM [67] the winner of the first data prefetching competition [138], (7) Idealized PC/AC, an idealized PC-localization address-correlation prefetcher [104, 68], and (8) EK+ASP that combines EK with a stream prefetcher from AMPM.

- **EK** executes on a separate in-order processing core with its own private unified L1 cache. The *m-core* and *p-core* share the L2 and L3 caches. Prefetches by the *p-core* appear at the L2 and the corresponding cache blocks are allocated as shared. The *p-core* uses a 64B fully-associative store buffer that never spills into the memory hierarchy. EK uses a fixed run-ahead distance threshold of 256 prefetches (8-bit RAC).

- **Dominant-Path Precomputation Prefetcher** or DPP, is similar to single-path PbPs [26, 162] in that it uses only the dominant execution trace per DLI. It uses the same prefetch core as EK. Since its p-slices may go astray, DPP monitors prefetching accuracy stopping and restarting the p-slice accordingly. DPP models the effect of the best to our knowledge previously proposed monitoring mechanism [162]
Chapter 6. EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

by checking whether the m-core uses the prefetched blocks within a number of iterations adjusting a confidence counter accordingly. The counter is initialized at 512 and it increases by half its value when prefetched addresses are used and decreases by one otherwise. The p-core stops if confidence reaches zero. When the m-core executes the DLI again, the p-slice restarts with the current input values. If the p-core stops three times within a window of 100M instructions, it suspends for 50M instructions to avoid polluting the cache. The confidence counter values and sampling periods were chosen experimentally.

- **No Memory-Dependencies** PbP or NoMEM, is a multi-path PbP which omits store instructions. It serves to demonstrate the benefits of including memory dependencies in EK’s p-slices. NoMEM employs monitoring similar to DPP, since its p-slices may go astray.

- **Next-Line** or NL, is a simple linear prefetcher which prefetches cache blocks with tag \( t + 1 \) when a request for tag \( t \) misses in the L1 cache.

- **Spatial Memory Streaming** or SMS, is an address correlation prefetcher that detects spatially-correlated memory accesses within fixed-size memory regions [132]. It collects the pattern of misses per region using an Active Generation Table (AGT) which it stores in a Pattern History Table (PHT) tagged with the initiating code or address region. Upon encountering the same code, the access pattern from the PHT is used to prefetch blocks within the relevant region. The SMS configuration uses 512B regions, \( 4 \times 16 \) AGT, and \( 256 \times 8 \) PHT, for a total of 31Kbits of SRAM [38].

- **AMPM** [67] uses two adaptive stream prefetchers (ASP) one for the L1 and another for the L2 [61]. At the L3, an additional access map pattern matching prefetcher performs coarse-grain access pattern detection [66]. The L1/L2 stream prefetchers have 16 stream filters for stream detection and 16 histograms for stream length prediction. The L3 pattern matching prefetcher is configured to track 52 entries of 16KB zones [67]. In total, AMPM uses 32Kbits of SRAM.

- **Idealized PC/AC**, an address-correlation prefetcher at the L2 representing an unrealistic upper bound of a Global-History Buffer (GHB) prefetcher, with unlimited PC indexes and GHB length [104]. We limit the GHB linked-list traversals per prediction to 10,000 [68].

- **EK+ASP** (EK+ASP). EK prefetches into the L2/L3 caches. EK+ASP combines the L1 stream prefetcher of AMPM with EK to also prefetch into the L1. The ASP can (a) prefetch into the L1, (b) more effectively prefetch linear streams, and (c) target additional loads. ASP uses 1.6Kbits of SRAM.

We assume a single cycle latency to all SMS, AMPM and PC/AC tables thus disadvantaging EK.

### 6.5 Evaluation

This section demonstrates EK’s advantages and weaknesses, compared to existing prefetchers. It also reports several characteristics of the DLIs and their p-slices, showing that a simple p-core suffices.

Sections 6.5.1 and 6.5.2 respectively report cache miss rates and performance with the various prefetchers demonstrating EK’s advantages and weaknesses. Section 6.5.3 presents a sensitivity analysis of DLI access latency vs. matrix density for the SpVM workload. Section 6.5.4 reports overall energy consumption. Section 6.5.5 reports several characteristics of the delinquent loads and their p-slices. Finally, Section 6.5.6 considers p-core complexity and the p-slice instruction mix showing that a simple p-core suffices.
Table 6.3: Workload inputs and execution samples.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Profiling Input</th>
<th>Measurement Input</th>
<th>Skip/Execute Instructions (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>simlarge</td>
<td>native</td>
<td>15.00/3.00</td>
</tr>
<tr>
<td>canneal</td>
<td>simlarge</td>
<td>native</td>
<td>15.00/3.00</td>
</tr>
<tr>
<td>cholesky</td>
<td>Cannizzo/sts4098</td>
<td>Boeing/bcsstk36</td>
<td>0.20/5.80</td>
</tr>
<tr>
<td>facesim</td>
<td>simlarge</td>
<td>native</td>
<td>29.3/3.00</td>
</tr>
<tr>
<td>ferret</td>
<td>simlarge</td>
<td>native</td>
<td>10.00/3.00</td>
</tr>
<tr>
<td>freqmine</td>
<td></td>
<td>2^20 nodes</td>
<td>4.00/3.00</td>
</tr>
<tr>
<td>graph500</td>
<td>2^10 nodes</td>
<td>2.5M×2.5M recs</td>
<td>4.44/3.56</td>
</tr>
<tr>
<td>hashjoins</td>
<td>0.5M×0.5M recs</td>
<td>ref</td>
<td>0.57/4.58</td>
</tr>
<tr>
<td>mcf</td>
<td>train</td>
<td>200M elements</td>
<td>14.17/3.83</td>
</tr>
<tr>
<td>radix</td>
<td>30M elements</td>
<td>18M×18M matrix</td>
<td>0.35/0.85</td>
</tr>
<tr>
<td>spmv</td>
<td>5K×2K matrix</td>
<td>2^20 nodes</td>
<td>58.00/5.00</td>
</tr>
<tr>
<td>scca2</td>
<td>2^10 nodes</td>
<td>simlarge</td>
<td>1.50/4.00</td>
</tr>
<tr>
<td>streamcluster</td>
<td></td>
<td>253(tissues)×15154(genes)</td>
<td>11.10/5.50</td>
</tr>
</tbody>
</table>

6.5.1 Effect on Misses

This section shows that: (a) EK is superior to the hardware prefetchers when the access patterns are hard to predict, and vice versa when the access patterns are predictable, (b) EK outperforms DPP when execution exhibits sufficient divergence, (c) including memory dependencies are crucial for some workloads, and (d) combining ASP and EK improves prefetch coverage. Figure 6.8 shows the L1, L2 and L3 Misses per Kilo Instructions, or MPKI, normalized to the no-prefetching baseline. Overall, EK reduces L2 and L3 MPKI on average by 67% and 70%, respectively. The following discussion first compares EK, SMS and AMPM focusing on the L2 and L3 because EK does not prefetch into the L1. Then EK is compared to PC/AC, DPP, NoMEM and EK+ASP.

Comparing EK to SMS and AMPM the following behaviors are observed: (1) Graph500 and SpVM benefit from EK only. These workloads are characterized by hard-to-predict non-recurring access patterns, and have multiple dependent execution paths. (2) Cholesky, freqmine, hashjoins, and SVM-RFE benefit from all prefetchers but more so from EK. They exhibit both predictable and hard-to-predict access patterns, and a single dominant path (SVM-RFE has two dominant paths). (3) Radix benefits from all prefetchers but more so from AMPM. It exhibits a stream which is highly predictable, so AMPM can issue prefetches for non-delinquent loads which are ignored by DPP and EK. (4) Blackscholes, facesim, ferret and streamcluster benefit only from the hardware prefetchers. They do not trigger trace collection during the profiling phase as their MPKI is low overall. (5) Canneal, SCCA2 and MCF benefit only from EK but still suffer from off-chip misses. Canneal’s p-slices iterate only a few times and thus do not manage to sufficiently run ahead. The p-slices either originate from loops that iterate a few times to start with, or appear on execution traces that exceed the upper limit on p-trace size (1000 instructions) used by EK’s profiling phase. SCCA2 repeatedly performs a relatively short graph traversal followed by a relatively longer compute phase and so the outer-loop is not captured in the p-slices due to a large intervening sequence of compute instructions limiting EK’s benefits. MCF’s behavior is similar.

Similar to the idealized PC/AC, EK proves effective on irregular predictable access patterns. The former outperforms EK only for freqmine which exhibits repetitive accesses, for both delinquent and
non-delinquent loads. EK targets DLIs only.

DPP works as well as EK only for \textit{radix} and \textit{hashjoins} which exhibit a single execution trace per DLI. DPP outperforms the existing prefetchers also on \textit{SVM-RFE} but not as much as EK. This workload exhibits a frequent yet not dominant execution path.

P-slice memory dependencies exist only in \textit{canneal}, \textit{cholesky}, \textit{graph500} and \textit{SSCA2}. NoMEM, which omits memory dependencies, fails with \textit{canneal} and \textit{cholesky} since their DLIs’ address calculation rely primarily on memory dependencies. NoMEM manages to reduce misses for \textit{graph500} and \textit{SSCA2}, since only a subset of their execution paths include memory dependencies.

When used alone, EK increases L1 misses on average by 1% due to cache invalidations caused by the prefetches that are allocated in the inclusive L2 and L3. When adding ASP, EK+ASP eliminates this increase in L1 miss rate, reduces the miss rate at all levels, and helps with workloads without DLIs.

### 6.5.2 Performance

Figure 6.9 reports the overall performance improvement over the baseline. EK performs the best for all workloads but \textit{blackscholes}, \textit{facesim}, \textit{ferret}, \textit{freqmine}, \textit{hashjoins}, \textit{radix} and \textit{streamcluster}. As Section 6.5.1 showed, \textit{blackscholes}, \textit{facesim}, \textit{ferret} and \textit{streamcluster} do not generate p-slices. \textit{Hashjoins} and \textit{radix} have predictable streaming accesses and non-divergent code paths. AMPM identifies and prefetches \textit{radix}’s
Chapter 6. EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

Figure 6.10: Access latency (excluding queuing delays) w.r.t. matrix density. Speedup is normalized to the 10^{-6} baseline.

access pattern. DPP performs as well as EK for these two workloads as they exhibit a single execution path per DLI. Idealized PC/AC slightly outperforms EK for freqmine because EK targets only DLIs. EK improves performance by 11\times or more for graph500 and SpVM, corresponding to the sharp drop in their L2 and L3 misses (Figure 6.8). The other prefetchers show almost no performance improvement for these two workloads. For the remaining workloads which are accelerated by EK (excluding graph500 and SpVM), EK improves performance by an average of 19%. EK outperforms SMS, AMPM and PC/AC for applications with non-recurring hard-to-predict access patterns. DPP, SMS, AMPM and PC/AC improve performance by an average of 6%, 3%, 8% and 12%, respectively (excluding graph500 and SpVM). EK+ASP outperforms EK by 5% on average. NL slightly reduces L2 and L3 misses for MCF and SVM-RFE, but at it degrades performance by up to 20% due to useless prefetches clogging the cache ports. This degradation is caused by an increase in data access latency due to congestion on the cache ports from useless prefetch requests.

6.5.3 Sensitivity to SpVM Sparseness

This section examines how the sparseness of the matrix in SpVM affects prefetching effectiveness. Section 6.1 explained that SpVM’s memory accesses comprise linear (Vv, Vi), fragmented linear (Mi, Mv) and data-dependent (RV) accesses. The position of the non-zero elements on each row, determines the result vector access pattern. The denser the rows, the more effective a stream prefetcher will be. At the extreme of a fully occupied row, the result vector pattern will be sequential and a stream prefetcher will perfectly predict it. For brevity we focus on the loads for Mi (similar to Mv) and RV.

Figure 6.10 depicts the average access latencies (left) for Mi and RV and performance (right) of the baseline, ASP and EK. Matrix density (fraction of non-zero elements) ranges along the x-axis from 10^{-6} to 1 (18M \times 18M matrix). ASP, which contains a stream prefetcher, reduces the access latency of Mi accesses for almost all matrix densities, however, it only hides the latency of RV accesses for relatively dense matrices (0.25 and higher). This behavior is a result of the sub-linearity of the Mi accesses, and the data-dependent, hard-to-predict RV accesses. EK is able to hide the latency of both Mi and RV accesses irrespective of the density. ASP fails to outperform the baseline for 0.1 and lower densities.

EK’s p-slices have the extra benefit of not only covering multiple execution paths, but also doing so in a self-contained manner where a single p-slice can change execution paths without referring back to
the main thread. To demonstrate this benefit, we evaluate a model of past work, which we refer to as separate p-slice, or SPS [88]. SPS constructs multiple p-slices where each represents a different execution path. Within each p-slice, control-flow instructions are included only to sequence through this path or terminate when execution is ought to switch path. This design limits run-ahead distance by the number of iterations executed before switching to another path. Instead, EK’s p-slices are self-contained and can toggle between multiple paths without termination.

For SpVM, SPS limits the run-ahead distance by the number of non-zero elements per matrix row. It constructs separate p-slices representing the inner and outer loops, each has its own termination condition. SPS shows only 1.7× speedup for the 10−6 matrix density. As density increases, so as the number of inner loop iterations, SPS’s performance improves, matching EK’s when density is 10−3 or higher.

6.5.4 Energy

Figure 6.11 reports the overall energy consumption and the EDP and ED²P energy-delay products. The figure separates the results for graph500 and SpVM from the rest of the workloads. Static and dynamic energy are included. These measurements underestimate the energy of SMS and AMPM by not including the energy consumed by their meta-data tables. This gives an advantage to SMS and AMPM over EK. EK’s measurements are broken down into the energy consumed by the p-core and the rest of the chip (m-core and memory hierarchy). With graph500 and SpVM, EK reduces overall energy by 74% on average; it reduces the energy of the m-core and memory hierarchy by 80%, but introduces overhead via the p-core. Adding ASP to EK reduces energy by an additional 1%. For the rest of the workloads, EK increases overall energy by only 1%; it reduces the energy of the m-core and memory hierarchy by 7%, but introduces overhead via the p-core. AMPM performs best reducing overall energy by 13%. Except for the workloads where EK does not trigger any p-slices, energy with EK is lower compared to the systems that use AMPM or SMS. For those workloads that AMPM and SMS can prefetch but EK does not, energy is similar to the baseline system as the p-core remains disabled.

6.5.5 P-Slice Characteristics

This section discusses the characteristics of the delinquent loads and their corresponding p-slices for those workloads that have DLIs identified during profiling.

P-slice Composition

This section shows that EK can effectively prefetch DLIs that depend on other DLIs or appear together with other DLIs and/or along multiple execution paths. Table 6.4 reports for each workload the number
Chapter 6. EKIVOLOS: Accurately Prefetching Hard-to-Predict Data Accesses

Figure 6.12: Delinquent load prefetch coverage.

of DLIs covered, p-traces extracted, and the corresponding p-slices constructed. For brevity, we highlight a few workloads reporting (1) whether their detected DLIs are dependent or independent, (2) whether the resulting p-slices are overlapping or non-overlapping, and (3) the number of p-traces used per p-slice construction. **Hashjoins** exhibits five independent DLIs, each of which has a single p-trace that does not overlap with the rest. EK maintains five separate p-slices. **SVM-RFE** contains four DLIs: two are dependent and appear along a single p-trace, one appears over two p-traces, and the fourth appears on a single p-trace. Accordingly, **SVM-RFE** has three p-slices, one of which is a merge of two p-traces. **Graph500** and **MCF** exhibit six and seven DLIs, respectively, all of which are either dependent, or their p-slices have overlaps. A single p-slice per workload covers all DLIs.

**P-slice Density**

Table 6.4 reports the p-slice static density, the ratio of the number of instructions retained in the p-slice over the number of instructions in the original sequence, and dynamic density, the average number of instructions executed per iteration by p-core over m-core. Combined with the performance results, these measurements reveal that the p-slices manage to run ahead even if they are relatively dense. EK’s p-slices include control flow, hence not all instructions must execute on every iteration. For example, Table 6.4 shows that **graph500**’s p-slices retain 94% of the main thread’s corresponding eight unique execution paths. On every iteration only one path is executed. Fortunately, one path dominates the execution and most of the instructions that the p-slice omits occur on this dominant path (the p-slice retains only 42% of the instructions on this path). Thus for most of the iterations, the p-core executes approximately 4 out of every 10 instructions executed by the m-core, while for some infrequent iterations the p-core executes almost all the instructions executed by the m-core. The instructions omitted by the p-slice are memory accesses which limit the load/store bandwidth at the main thread, preventing the m-core from catching the p-core. The average IPC for the m-core and p-core are 1.93 and 0.79, respectively.

**P-Slice Prefetching Accuracy**

Figure 6.12 reports coverage, the fraction of DLI targets that precomputation prefacters correctly predict demonstrating the importance of including all excited control-flow in the p-slices. We use coverage as a proxy for accuracy; the latter determines whether a p-slice issues correct target addresses or not. Correct prediction does not imply timeliness. EK is nearly 100% accurate in all cases. The profiling phase captured all paths that appeared in the measurement runs and memory dependencies proved
Table 6.4: P-Slice and DLI statistics.

<table>
<thead>
<tr>
<th>Workload</th>
<th>DLIs</th>
<th>P-traces</th>
<th>P-slices</th>
<th>P-slice Density</th>
</tr>
</thead>
</table>
| stable. This may not be the case for other workloads or other profiling datasets. DPP matches EK’s accuracy only for hashjoins and radix where only one execution path occurs per DLI. For workloads with multiple execution paths, one or a combination of the following scenarios may occur. (1) DPP fails when the address calculation on the included dominant path depends on the other less frequent ignored path(s). E.g., in SpVM, after the inner loop (dominant path) exits, the computation of the next row index (row idx) is performed in the outer loop (infrequent path). (2) DPP is only partially effective when the address calculation on the dominant path is independent of the other less frequent path(s), then DPP will miss some of the opportunities to prefetch on the less frequent path(s), but will still be accurate at prefetching on the dominant path. E.g., with Cholesky and SVM-RFE, DPP covers 86% and 49% of the prefetches, while missing the rest which are distributed on six and three other less frequent paths, respectively.

For workloads where there are multiple execution paths, DPP’s accuracy depends on whether one path appears frequently enough and whether the address calculation along this path is independent of the other less frequent paths. For Cholesky and SVM-RFE there is a frequent, independent path, but even then DPP’s accuracy trails that of EK as DPP fails to prefetch along the less frequent paths. DPP fails when there is no dominant execution path, e.g., in graph500 and SSCA2, or when there are data dependencies on the DLI address across iterations, e.g., in SpVM.

6.5.6 Prefetch Core

So far, EK’s p-slices were executed on a simple in-order core, with a unified 4KB L1 cache. Executing EK on an Out-of-Order (OoO) core with separate 32KB I/D L1 caches improves overall performance by only 1%, while exerting 84% more energy.

Table 6.5 shows a breakdown of the instruction types executed by the main thread and the p-slice. Some instruction categories never appear on the p-core, e.g., Vector/Floating and Synchronization instructions. Store and other classes appear much less frequently. Thus it may be possible to simplify the p-core further.
### Table 6.5: Total instruction counts (all workloads).

<table>
<thead>
<tr>
<th>Category</th>
<th>Main Thread</th>
<th>Ekivolos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>213</td>
<td>81</td>
</tr>
<tr>
<td>Co-processor</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Data Processing</td>
<td>227</td>
<td>85</td>
</tr>
<tr>
<td>Function Calls</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>Multiply</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Privilege</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Stack</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Load</td>
<td>190</td>
<td>101</td>
</tr>
<tr>
<td>NOP</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>Store</td>
<td>68</td>
<td>12</td>
</tr>
<tr>
<td>Synchronization</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>Conditional Branches</td>
<td>137</td>
<td>55</td>
</tr>
<tr>
<td>Unconditional Branches</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Vector/Float Arithmetic</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>Vector/Float Multiply</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>Vector/Float Data Processing</td>
<td>5</td>
<td>-</td>
</tr>
</tbody>
</table>

### 6.6 Related Work

Precomputation based prefetching (PbP) is a mature research topic. Zhang et al. present a summary of static vs. dynamic techniques [162]. This section reviews the p-slice construction and execution methods considered in past PbP work, in addition to other prefetching techniques which incorporate control flow.

#### Construction of P-slices

Manual [164], compiler-based [82, 88, 117], or hybrid [94] techniques can construct p-slices which include control flow, hence their p-slices can replicate the main program’s control divergence, if it exists. However, manual p-slice construction is a burdensome task, while a compiler requires source code. Past automatic techniques which operate only on the application’s executable [8, 9, 26, 42, 123, 162] purposely ignore all or part of the control-flow and target building optimized and fast p-slices, at the expense of accuracy. Some of these techniques incorporate reactive monitoring mechanisms to realign a p-slice that goes awry [26, 162, 164]. Illusionist constructs control flow inclusive p-slices by operating directly on application binaries, however, it omits biased branches in an attempt to build shorter p-slices [9]. The inner loop branch is biased yet tracking it is essential for prefetching SpVM. This work studied emerging workloads whose control divergence renders these approaches ineffective. EK instead automatically constructs p-slices which include enough control flow instructions, it does not require external monitoring nor imposes overheads on the main thread and operates on application binaries.

#### P-slice Execution

Several architectures have been proposed to execute the p-slices concurrent to the main program. Rabbah et al. [117] embeds the p-slice instructions alongside the main program’s binary. By optimizing the program’s ILP, the compiler statically injects these instructions ideally without lengthening the main program’s critical path. Hassanein et al. [56] execute the p-slice in-memory, where a memory processor
fetches the p-slice, executes it, and forwards the resulting load value to the main processor. Others propose executing the p-slice as a separate thread on a specialized prefetch engine [8], an SMT context [26, 88, 94, 162, 164], or a conventional core in a CMP [9, 42]. EK constructs fully functional p-slices, which can either be executed on a prefetching engine, an SMT thread, or a separate core.

**Other Control-Flow Based Prefetching Techniques**

There are prefetching techniques that incorporate some control flow effects [9, 88, 78, 136, 163] and may offer similar benefits as EK under certain conditions. Liao *et al.* [88] construct multiple p-slices corresponding to multiple execution paths for a given delinquent load. Each p-slice includes a terminating condition which signals a change in the execution path. Hence, the length of instructions a p-slice executes before shifting to another path limits its run-ahead distance. Instead, EK’s p-slices are self-contained and can toggle between multiple paths without termination. Dual-core Execution (DCE) [163] runs the same program on two cores, where a *front* core retires instructions in a FIFO queue, which are then fed as a stream into a *back* core. The front core pseudo-retires long-latency loads using invalid values, and may drift away from the correct execution path. The back core detects such cases and squashes both pipelines. The FIFO queue size and the rate of pipeline squashes constrain the front core’s run-ahead distance. Illusionist [9] accelerates an application, running on a lightweight core (LWC), by executing a subset of its instructions on an aggressive core (AGC). AGC provides branch and cache hints to LWC via a hint queue. The instruction subset executing on AGC is extracted using a dynamic runtime compiler, and is optimized by eliminating hint-irrelevant instructions and biased branches. B-fetch [78] uses an auxiliary prefetch pipeline to predict the program’s control flow, as a sequence of basic blocks, using branch predictions. Then, for those predicted basic blocks, it predicts the target addresses of memory operations within, by correlating register value variations with a basic block sequence. The techniques above have the advantage of operating at runtime on unmodified executables. However, workloads with data dependent control flow, such as SpVM, can be challenging for them due to the data dependent behavior of the inner-loop in SpVM. DCE may suffer from frequent back core squashes due to the use of invalid load data by the front core [163], while branch prediction may not be accurate enough for B-fetch [78]. Slipstream [136] bares some similarities and limitations with DCE: the front core generates predictions (not prefetches) and bypasses ineffectual instructions, ultimately deviating from the correct path. Slipstream shares the same limitations as DCE.

### 6.7 Concluding Remarks

This chapter targeted emerging workloads with irregular, hard-to-predict access patterns that were shown to foil existing hardware prefetchers. The proposed EK prefetcher builds upon the wealth of past work on precomputation prefetching, but specifically targets mitigating control-divergence induced inaccuracies.

The key conclusions are that when iterating over large quantities of data, having fast precomputation slices is less important than having accurate ones. Even the tiniest speed advantage per iteration soon adds up to a sufficient distance advantage as long as the precomputation slice faithfully recreates the relevant control flow. Simply restricting attention to the paths that are observed during test runs proved sufficient for generating slices containing just enough of the program’s control flow. Finally,
a simple, stripped down in-order processing core, with a relatively small private L1 cache suffices for precomputation prefetching.

Future work may investigate a hardware only implementation of EK, developing specialized prefetching cores for executing the precomputation slices, or off-loading computation on these cores similar in spirit to recent work [83].
Chapter 7

Summary

This dissertation proposed three techniques to improve performance of crucial classes of emerging applications by reducing their instruction and data cache-related stalls. For Online Transaction Processing (OLTP) applications, we propose two novel hardware transaction scheduling techniques to improve instruction cache locality. For a set of emerging data-intensive algorithms, commonly used in Big Data applications, we propose a precomputation-based prefetching solution that proves feasible and effective on hard-to-predict data access patterns, which foil state-of-the-art prefetching systems.

First, we propose SLICC, a hardware transaction scheduling technique to improve instruction cache locality for OLTP workloads. SLICC exploits the application behavior (inherent code overlap among concurrently executing transactions) and the aggregate L1 cache capacity on modern chip multi-processor systems (CMPs). SLICC co-schedules transactions and dynamically migrates them among cores, such that ideally one transaction fetches a cache block that is reused by many other transactions. SLICC is a low-level hardware algorithm that requires no code instrumentation. However, SLICC is effective only if the transactions’ instruction footprints fit in the aggregate L1 instruction cache capacity of a CMP.

Second, we propose STREX, an alternative transaction scheduling policy, which also tries to improve instruction cache locality, however, instead of spreading the instruction footprint across multiple cores, it dynamically pipelines and multiplexes transactions on a single core. STREX exploits the available temporal overlap among similar OLTP transactions. It groups similar transactions into teams, and stratifies their execution on a single core, thus improving instruction and data locality. As opposed to SLICC, STREX was demonstrated to be insensitive to the available number of cores, thus outperforming SLICC when the available aggregate cache capacity is not sufficient.

Third, we propose EKIVOLOS (EK), a precomputation binary-based prefetcher that is programmer-transparent and effective on non-repetitive, irregular, hard-to-predict data access patterns. The proposed EK prefetcher builds upon the wealth of past work on precomputation prefetching, but specifically targets mitigating control-divergence induced inaccuracies. This dissertation departs from the conventional notion that having fast precomputation slices (p-slices) is less important than having accurate ones. We show that when iterating over large quantities of data, the opposite is true. EK builds on this novel paradigm and constructs p-slices which include enough control flow and data dependencies to faithfully replicate the original program’s execution path, while still being able to run ahead.
7.1 Contributions

In summary, this dissertation makes the following contributions:

Understanding OLTP Performance

⇒ We characterize the memory behavior of OLTP workloads showing that their transactions suffer
from instruction misses, and that their instruction streams exhibit intra- and inter-transaction
recurring patterns leading to eviction of useful blocks that are re-referenced later.

⇒ We demonstrate a high-degree of temporal code overlap among concurrently executing transactions,
and show that same-type transactions exhibit the most overlap.

⇒ We show that the instruction footprint of OLTP transactions fit comfortably in the aggregate L1
instruction cache capacity on modern CMPs.

⇒ We demonstrate that recently proposed cache replacement and insertion policies reduce instruction
misses by a small fraction of what can be achieved using larger caches.

SLICC: Spreading Instruction Footprint in Space (Multiple Cores)

⇒ We propose SLICC, a hardware dynamic thread migration algorithm which aims to partition
the relatively large instruction footprint and spread it on multiple cores within a CMP, while
dynamically assigning transactions to cores.

⇒ We demonstrate that SLICC reduces instruction misses by 64% resulting in 67% overall perfor-
mance gain, on average, over a system with no effort to improve cache performance. SLICC
outperforms a state-of-the-art instruction prefetcher by up to 27%.

⇒ We further analyze individual transaction type behavior with SLICC showing that some trans-
actions do not benefit much from thread migrations. Accordingly, we propose SLICC-Selective,
an enhancement to SLICC which dynamically profiles transactions and limits migrations to only
threads that will improve performance.

STREX: Spreading Instruction Footprint in Time (Single Core)

⇒ We propose STREX, a hardware thread scheduling technique that exploits inter-transaction local-
ity, and groups and synchronizes the execution of similar transactions on a single core.

⇒ We demonstrate that, on average, STREX reduces instruction and data misses by 40% and 15%,
respectively, resulting in 41–55% performance gains for systems with 2–16 cores, over a system with
no effort to improve cache performance. STREX outperforms SLICC when the available aggregate
cache capacity does not fit the transactions’ instruction footprints.

EKIVOLOS: Effectively Prefetching Hard-to-Predict Access Patterns

⇒ We identify a set of algorithms commonly used in Big Data applications and demonstrate hard-
to-predict access patterns, which foil state-of-the-art prefetching techniques.
⇒ We single out precomputation-based prefetching as a potential solution to such access patterns, emphasizing the need for p-slices which include control-flow and data dependences.

⇒ We revisit precomputation-based prefetching advocating that p-slices do not have to be much shorter than the main program to be effective, and demonstrating that including just enough control flow in the p-slices results in accurate and timely prefetching requests for workloads that past single-path p-slices are ineffective.

⇒ We propose EKIVOLOS, a p-slice construction algorithm which operates directly on binaries. It observes the multiple execution traces that lead to delinquent loads during a profile run, then merges those traces to construct control-flow and data dependence inclusive p-slices.

⇒ For set of emerging applications/kernels, we demonstrate that EKIVOLOS reduces L2 and L3 cache misses by 67% and 70%, on average, over a no-prefetching baseline, outperforming state-of-the-art prefetching techniques and a model of past single-path precomputation-based prefetchers.

⇒ We demonstrate that a simple, stripped down in-order processing core is a sufficient p-slice execution engine.

7.2 Final Thoughts and Directions for Future Work

As Dennard scaling has ceased [21, 35], the high performance computing industry cannot simply rely anymore on advances in technology to scale performance. Innovations at the architectural level have become a necessity. Incorporating more and more cores on a single chip has been successful at increasing the theoretical peak throughput per unit area and energy. However, emerging memory-bound applications have not benefited that much and a key challenge remains: how can we mitigate the gap between memory latency and CPU clock speeds? Recent advances in memory architecture such as 3D stacked DRAM [116] or the Hybrid Memory Cube [71] are promising, however extracting benefits from them remains challenging.

Future work may target enhancements to the proposed techniques, alternate ways of addressing memory stalls or study other types of workloads. A potential enhancement to our transaction scheduling techniques is to enable controlling and maintaining Quality of Service\(^1\) (QoS) guarantees. A potential enhancement to our precomputation-based prefetcher is to eliminate the need for offline profiling. Alternate or complementary approaches include migrating the full computation threads, or slices thereof, to execute on specialized accelerators, or “in-memory” or “in-cache” compute engines.

SLICC and STREX focus mainly on improving the overall system throughput and not as much on individual transaction latency. Although in some scenarios the increase in overall throughput justifies the increase in average transaction latency [154], it may be desirable to have some form of administrative control. This control can be provided through the team\_size configuration parameter, where larger teams offer higher performance, and also higher latency. However, the granularity of this control is not at the level of individual transactions. Future work may investigate methods to detect individual transaction latency at runtime and interfere with the scheduling algorithm to ensure QoS requirements are met for particular transactions.

\(^1\)Quality of Service (QoS) is a general term that refers to an agreement between a service provider and the entity obtaining the service, where the former guarantees certain quality aspects of the delivered service. For example, a stock exchange server can guarantee the response time to a purchase request will be within a certain amount of time.
Profiling-based techniques are seldom deployed in practice because their benefits seem not to outweigh the burdens and overheads associated with profiling the applications offline. Ekivolos currently utilizes an offline-based profiling phase limiting its ability for adoption in production systems. However, Ekivolos improves performance for some workloads, e.g., graph traversal, by an order of magnitude making the solution more attractive, given the growing interest in kernels like graph analysis. To make Ekivolos more attractive, future research could investigate enhancements to avoid the offline profiling; one way might be to perform the profiling automatically online within a runtime framework, akin to Just-in-Time compilers [135].

Conceptually, this dissertation proposed techniques which alter the conventional on-chip flow of instructions and data. Future work may investigate alternative approaches to instruction- and data-flows. For example, as opposed to the conventional system where a thread fetches instructions and data to be executed on a core, SLICC moves the threads to where the instructions reside, and Ekivolos spawns a helper-thread to fetch the data for the main thread. OLTP systems, as demonstrated in Chapter 3, are composed of multiple DB operations. Future research might investigate accelerator designs which can host these operations, and ideally be more efficient at executing them [83]. Processing-in-memory (PIM) is an analogous concept that is re-emerging\(^2\) [14]. PIM allows computation to be offloaded to specialized units that reside closer to the data in main memory. Given the anticipated adoption of large on-chip stacked DRAM as last-level caches [116], future research may investigate “in-cache” processing, where simpler computational units are deployed nearer to the DRAM cache. Such designs could exploit the higher DRAM internal bandwidth, and reduce the amount of data that is inefficiently transferred back and forth between the DRAM and the processing cores, thus improving performance and/or saving energy.

Finally, future work may investigate other types of workloads, or create a standardized benchmarks suite. SLICC and STREX were tuned for transactional workloads, and Ekivolos targets some emerging data-intensive algorithms commonly used in “Big Data” applications. Future research may investigate migrating computation or prefetching for other types of instruction- or data-intensive workloads such as web serving or media streaming. Furthermore, given the growing interest in “Big Data” research in the computer architecture community [4, 76, 86], it is imperative to have a standardized suite of “Big Data” benchmarks. In this dissertation we have made an effort towards that direction by creating our own custom set of benchmarks. Ahn et al. made a similar effort [4]. However, a standardized suite with reference datasets would trigger lots of further micro-architectural studies, and save researchers the hassle of finding and setting up their own benchmarks.

\(^2\)Processing-in-memory (PIM) was originally presented in the late 90’s [34, 46], however it was never implemented in production systems. Technology advances which enabled integration of CMOS and DRAM spurred interest again in the concept of PIM [50]
Bibliography


[29] Peter Sanders Dorothea Wagner David A. Bader, Henning Meyerhenke. Graph Partitioning and Graph Clustering, volume 588. AMS Contemporary Mathematics, 3 2013.


