Improving Data Locality in Applications through Execution Delegation

by

Bryant J Curto

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With the slowing or even death of Moore's Law, computer system architectures are trending toward more CPU cores. This trend has driven systems researchers to explore novel ways of utilizing this computational power for improved efficiency and performance. One such approach is to use this power to help alleviate the memory wall problem through execution delegation. The memory wall problem describes the issue whereby system performance hits a wall that is dictated by the latency of accessing main memory. Using execution delegation, the execution of the application on one core is delegated to another core. The desired result is that the cores of the system are specialized to access mostly disjoint sets of data. In this way, data locality and, therefore, performance are improved.

The aim of this work is to develop tools and methods for predicting situations in which execution delegation via user thread migration is useful for improving an application's data locality. To this end, a microbenchmarking tool named Accesstest is used to perform a systematic study of execution delegation via user thread migration. Further, an approach, which makes use of a working set characterization tool named Accessprof, is developed to predict the qualitative impact of delegating an execution sequence. This prediction approach is verified and used to improve the Apache HTTP server's performance by as much as 11%.

Acknowledgements

I would like to thank all the people who made this thesis possible.

I would like to express sincere thanks to my supervisor, Dr. Martin Karsten. His guidance throughout my master's program has been invaluable. He enabled and encouraged me to work harder and smarter in search for hidden truths. My time in his lab has been a great learning experience.

I would like to thank my current and former labmates, Teodor (Alex) Ionita, Peter Cai, and Gan Wang. Thank you for your support, shared expertise, and the memorable days we spent together in Waterloo.

I would like to thank Dr. Peter Buhr and the entire C∀ team for their insight, support, and kindness.

Finally, I would like to thank my wife, family, and friends for celebrating with me during my successes and supporting me during my failures.

Dedication

This thesis is dedicated to the scientists in my life – my wife, Soo Jung Oh, and mother, Lynne Ierardi-Curto – for their love, support, and begrudging willingness to participate in discussions about CPU caches.

Table of Contents

List of Figures

Chapter 1

Introduction

Modern day computer architectures are vastly different from those of five decades ago (or even one decade ago). Generation after generation, computer architectures have grown smaller, cheaper, and more powerful. For example, since 1978, processor performance has grown a tremendous 50,000-fold [22]. Processors can be found in most daily items ranging from cars to greeting cards. This growth has enabled developers to be less stingy with hardware resources, with which many have traded performance for productivity. As of September 2022, productivity-oriented programming languages like Python and Java outrank more performance-oriented languages like C and C++ in terms of programmer interest [7].

Much of the growth of processor performance came from an increase in processor transistor counts. Moore's Law, stated by Gordon Moore in 1965, predicts that the number of transistors per chip would double roughly every two years and cost less [32]. Moore's Law has remained relevant largely because of Dennard scaling [15], observed by Dennard et al. in 1974, which states that, as transistors are reduced in size, their power density stays constant. This means that more, smaller transistors fitting into the same area use the same amount of power. However, this rapid growth of processor performance appears to be slowing down or to have recently come to an end.

As of today, transistor counts are off from that predicted by Moore's Law by a factor of ten [22]. This is due, in part, to a breakdown in Dennard scaling because of the challenges encountered when attempting to shrink current transistor sizes any farther [15]. This has led experts to argue as to whether or not Moore's Law and Dennard scaling are dead. Nevertheless, in an attempt to achieve higher performance, focus has shifted to architectures with multiple energy-efficient CPU cores rather than one power-hungry CPU

core. While many systems contain a single processor package of CPU cores, systems with two or more processor packages (each containing multiple CPU cores) are also common. In all such scenarios, in order to utilize these cores, developers must divide their programs into parts such that the parts can be executed by a system's cores in parallel.

One approach of utilizing this computational power is to use it to alleviate the memory wall problem [31], a growing pain that arose during the period of rapid growth. This growing pain causes memory access latency to bottleneck computation speed. It is caused by the growing disparity between CPU speeds and main memory speeds. Each CPU core has a cache, in which it stores recently accessed data. Accessing data stored within the cache is much faster than accessing data stored in main memory.

However, caches are beneficial only when they contain the data being retrieved. Typically, there are two situations when data is not in the cache: the cache is too small or multiple cores are operating on the same data. Through specializing CPU cores to perform certain tasks, both of these issues can be alleviated. One approach for performing this specialization is called spatial execution reorganization or, more simply, execution delegation. Numerous studies have been published presenting approaches and systems with which performance is improved through execution delegation. However, there are only a few, rigid situations in which execution delegation is applied and studied, namely the execution of critical sections and system calls.

The aim of this work is to develop tools and methods for determining when execution delegation is beneficial to application performance. To this end, a subclass of applications fitting the thread-per-session paradigm are studied. A microbenchmarking tool named Accesstest is developed to study the impact on performance of execution delegation via user thread migration. A characterization tool named Accessprof and prediction approach is developed that can be used to qualitatively indicate whether or not an execution sequence is amenable to execution delegation via thread migration. This prediction approach is applied to and verified using several system calls performed by an Apache [3] HTTP webserver instance[1].

The remainder of this thesis is organized as follows. Chapter 2 reviews previous research that is related to this work. Chapter 3 describes background information necessary for understanding this research. Chapter 4 outlines the session-loop pattern: the characteristics of the class of applications that benefit from the results of this research. Chapter 5 overviews the design and implementation of the Accesstest microbenchmarking tool along with experimental results. Chapter 6 outlines the steps in the execution delegation

[1]Source code and results data can be found at https://gitlab.uwaterloo.ca/bcurto/accesstest.

prediction approach including the Accessprof characterization tool. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Related Work

2.1 Execution Delegation

Execution delegation is a useful software solution for situations where the cost of moving data to the computation is higher than moving the computation to the data. The functionality needed to perform execution delegation has existed in a range of software systems for many years: actors and SmallTalk [20], active messages [44], autonomous objects [14], and messengers [19]. Thread migration is one approach for performing execution delegation. Kogge et al. [28] recently present evidence demonstrating the strength of thread migration for improving data locality of memory bound applications where computation is dominated by memory access and movement.

Reif et al. [38] introduce migration-based synchronization. Thread migration is used to synchronize access to shared resources within a critical section and achieve improved data locality, and can be used as a replacement for locks. Each shared resource in the system has a corresponding *synchronization core*, which is uniquely permitted to access the shared resource. For a thread to acquire exclusive access of a shared resource, it migrates to the resource's corresponding synchronization core. For the thread to release exclusive access of the shared resource, it migrates back to its original core. Preemption and implicit migration (e.g., load balancing) are not provided on synchronization cores. A thread starts or stops executing on a synchronisation core only if it explicitly requests. In this way, execution of the critical section is delegated to the synchronization core. Data locality is improved since shared resources remain within the synchronization core's local cache. Additionally, by preventing thread preemption on the synchronization core, mutual exclusion is guaranteed since a core can execute only one thread at a time.

Dysart et al. [18] introduce *Emu*, a system architecture that uses kernel thread migration, facilitated by hardware, to improve the data locality of data-intensive applications exhibiting weak-locality, or locality within a large memory region. Within their system, the global address space is partitioned across *nodelets*, or disjoint sets of cores. When a thread attempts to access memory that is not local to the nodelet on which it is currently scheduled, it is automatically migrated by hardware to the nodelet to which the memory is local. In effect, threads migrate to the data they are accessing rather than having that data brought to them. For the applications previously described, data locality and therefore performance is improved. Kogge [29] and Springer et al. [42] further discuss applications that benefit from such an architecture.

While there are examples of thread migration being used to perform execution delegation, use of an alternate approach (typically some form of message passing or remote procedure call) is more common. Soares et al. [41] propose an operating system mechanism named FlexSC. This mechanism provides exception-less system calls for system intensive workloads (and especially highly threaded server applications). Using FlexSC, the execution of a batch of system calls is delegated to a predetermined set of cores. When a *FlexSC-Thread*, which is a user thread provided by FlexSC's M:N user threading library, wants to perform a system call, it writes the system call number and associated arguments to a *syscall page*. The thread then continues executing or blocks until it has received a result. Once a threshold of system calls have been recorded, all system calls are executed by a *syscall thread* sequentially on one of a predetermined set of cores. The result of each system call is written back to the syscall page, which is later consumed by the requesting FlexSC-Thread. Through batching and delegating the execution of system calls to specialized cores, data locality is improved, cache pollution is reduced, and the number of user-kernel boundary crossings is reduced.

io_uring [12] is a Linux kernel system call interface for performing asynchronous IO operations introduced in Linux kernel version 5.1. io_uring is designed with the aim of being easy to use, extendable, feature rich, efficient, and scalable. It has two single producer, single consumer queues: a *submission queue* and a *completion queue*. The application enqueues IO requests on the submission queue to be completed by the kernel. At a later point in time, the application dequeues the result of each request off of the completion queue. io_uring can be used to achieve similar benefits (i.e., reduced user-kernel boundary crossings and improved data locality) as FlexSC. Using io_uring, an application can reduce the number of user-kernel boundary crossings by batching IO requests. It can also be configured so that the kernel polls for IO requests, thereby eliminating nearly all user-kernel boundary crossings. Further, the execution of IO operations is delegated possibly to specialized cores.

Similar to the previously described migration-based synchronization are the works of Lozi et al. [30], Roghanchi et al. [39], and Srinivasan [43]. In these works, the execution of critical sections is delegated to a single specialized core. This is implemented using remote procedure call for the first and message passing for the others. Lozi et al. additionally design a dynamic profiling tool to determine which critical sections would benefit most from delegation. The percentage of execution time spent in critical sections for each lock is computed. The authors show that, if the percentage of execution time of a critical section for a given lock is over 20%, then their locking technique performs better than a POSIX lock. Further, if it is over 70%, then their locking technique outperforms all other lock algorithms known at the time of publication.

Hendler et al. [21] propose *flat combining*, a technique for synchronizing accesses to a shared data structure by multiple threads through cooperation. Flat combining is purported to have better data locality and to be simpler than fine-grained locking and lock-free mechanisms. In essence, when multiple threads attempt to concurrently operate on a shared data structure, they delegate the execution of their operations to one of the modifying threads. In turn, this thread executes each delegated operation and reports each result back to the requesting thread. The delegatee thread is not typically predetermined and can change as time passes such that a thread executes a batch of operations before another takes over the responsibility.

The work presented in this thesis differs from previous work in one key way. Previous work proposes methods or infrastructures by which predetermined operations of an application can be delegated. Some also describe approaches for determining which, if any, of these predetermined operations would benefit from delegation. The work in this thesis develops an approach for predicting which execution sequences (i.e., sequences of instructions that are not necessarily contained within a single operation) of an application would, if delegated, result in improved application performance. This is accomplished through a detailed analysis and characterization of the memory locations accessed by the application.

2.2 Improved Data Locality by Design

To improve data locality, execution delegation is unobtrusive when compared to designing systems from the ground up with locality in mind. Nevertheless, if machines continue to trend toward higher CPU core counts, systems, data structures, and algorithms must be designed with data locality in mind.

The following are prominent examples of such whole-system, data-locality-minded redesigns of the operating system. Baumann et al. [13] develop a novel OS structure, called

the *multikernel model*. A multikernel named *Barrelfish* is also implemented. In the multikernel model, the machine is treated as a distributed system of CPU cores that communicate over a network explicitly through message passing (and through shared memory as an optimization). Note that, while the OS is designed to not use shared memory, applications are not prevented from sharing memory among cores. Replication is used to handle OS state that is traditionally accessed and modified by all cores of the system. Cores access and update this shared state as if it were a local replica. Consistency is maintained through message passing. The multikernel model enables improved data locality in several ways. The OS traditionally makes use of high performance data structures requiring a minimal amount of data movement per access. However, the number of cores in a typical machine and the number of such data structures accessed while performing any given operation are large enough that message passing is found to be more efficient. Further, by treating the machine as a network, well-known networking optimizations (such as pipelining and message batching) are applied to improve data locality. Current trends indicate that core counts will continue to increase. As a result, so too will the benefits of the above design choices.

Peter et al. [35] design and build a new OS, named *Arrakis*, in which the traditional roles of the OS have been split into the control plane and data plane. In the control plane, the kernel provides network and disk protection without intervening in every IO operation. Meanwhile, through the data plane, applications directly access IO devices that (partially or fully) support virtualization. Each application receives its own network stack and the cores that are executing an application also process the application's packets. As a result, amongst other benefits from this split, data locality is greatly improved in the network stack through reduced lock contention and cache effects.

There has also been much research into several classes of data structures and algorithms designed with the cache in mind [9]. One class, referred to as cache-aware, consists of a set of algorithms and data structures that take the characteristics of the cache (e.g., capacity, cache line size, associativity, and number of levels) as arguments. Using this information, the algorithm or data structure better utilizes the cache in order to reduce the number of cache misses. Another class, referred to as cache-oblivious, similarly attempts to reduce the number of cache misses. However, these data structures and algorithms require no knowledge of the cache's characteristics. An approach for implementing cache-oblivious algorithms is through recursive divide-and-conquer such that the problem is repeatedly subdivided into smaller and smaller subproblems. Through subdivision, each subproblem becomes small enough to take advantage of the cache independent of its characteristics.

Chapter 3

Background

3.1 Caches

3.1.1 Overview

In general, a cache is used to store data for the purpose of reducing the cost (e.g., time, energy) needed to retrieve this data at a later point in time. Modern day CPUs typically have multiple caches: hardware components physically residing on the processor die used to temporarily store data residing in main memory for the purpose of reducing the latency of future accesses. CPU caches (hereafter referred to as simply caches) are typically several orders of magnitude smaller than main memory. However, their reduced capacity is the key feature that enables them to reduce data access latency. As a result of their size, caches can reside physically closer to the CPU cores requesting and using the data. Further, they can be made from more premium technologies, which has a higher cost per byte, as compared to main memory.

Caches take advantage of the fact that programs obey the principle of locality [17]. This principle asserts three things. First, over any interval of time, the distribution of memory accesses of a program is nonuniform. Second, the frequency with which a program accesses a memory location changes slowly. Third, memory locations accessed in the immediate past are highly likely to be correlated with memory locations that will be accessed in the immediate future. There are two main types of locality: temporal and spatial. Temporal locality refers to the tendency of a program to reuse data within a small window of time. Spatial locality refers to the tendency of a program to use data that is stored close to previously used data in the address space. Caches take advantage of temporal locality by

storing recently accessed data. They take advantage of spatial locality by prefetching and storing data located near previously accessed data in the memory space.

Caches are pivotal to mitigating the effects of the memory wall problem, which was first described in 1994 [31]. The memory wall problem describes the issue whereby system performance hits a wall that is dictated by the latency of accessing main memory. One can think about the memory wall problem through the lens of Amdahl's law [11]. Fundamentally, it is caused by the disparity in improvements to CPU speeds compared to the speed of accessing main memory.

Below is the equation for the average time to access memory:

$$t_{avg} = p \times t_c + (1 - p) \times t_m$$

where t_c and t_m are the cache and main memory access times respectively. p is the probability of a cache hit, which is when requested data is retrieved from the cache instead of main memory. As the time to access the cache (t_c) approaches 0, the average time to perform a memory access (t_{avg}) is dominated by the time it takes to access main memory (t_m). With no clear solution to aligning the speeds of the CPU and of accessing main memory, much work has been put into decreasing the probability of a cache miss ($1 - p$). A memory access results in a cache miss if the requested data cannot be found in the cache.

Besides caches, other strategies are employed to mitigate the memory wall problem including out-of-order execution and speculative execution [22]. These strategies typically entail attempting to perform useful work while waiting for data to be retrieved from memory.

3.1.2 Memory Hierarchy

Figure 3.1 shows the structure of a common memory hierarchy, which is broken down into four main parts: the CPU registers, cache hierarchy, main memory, and stable storage. The first three levels of the memory hierarchy (i.e., the CPU registers, cache hierarchy, and main memory) are typically volatile, meaning that they retain information only while powered on. This is in contrast to the fourth level (i.e., non-volatile storage), which retains information even when powered off.

Registers are located at the top of the memory hierarchy and are the fastest memory structures in the hierarchy. Each CPU core has its own set of registers and the registers are directly connected to its core. However, this proximity comes at the cost of size: registers are small and few in number. Unlike all other parts of the memory hierarchy, registers

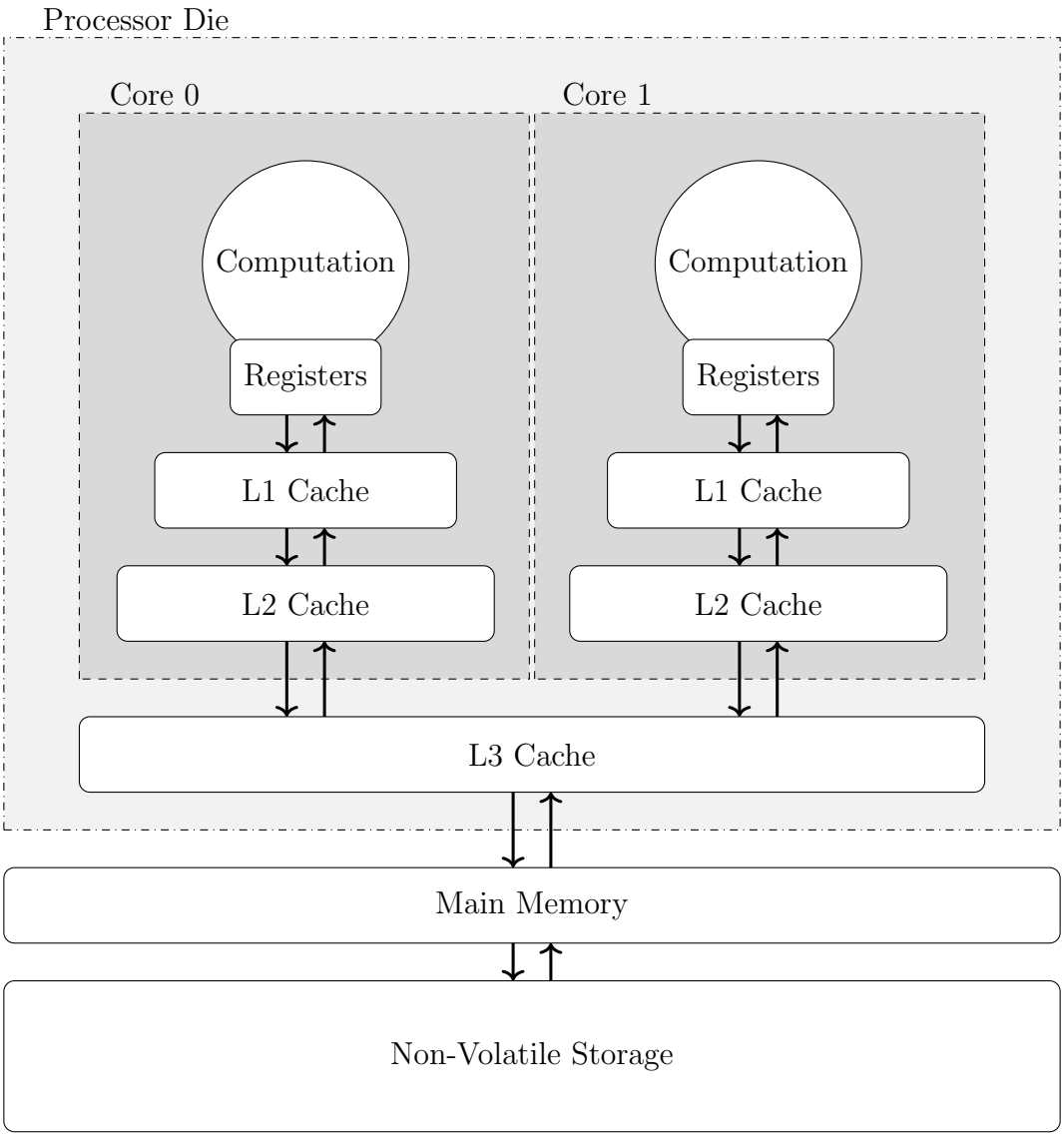


Figure 3.1: Common memory hierarchy.

are CPU private. This means that the contents of a set of registers can only be modified by its corresponding CPU core. In most instruction set architectures, registers are split into two groups: special purpose and general purpose registers. Special purpose registers store register specific values (e.g., a pointer to the instruction currently being executed). General purpose registers are used for temporarily storing values during computation: the inputs and outputs of machine code instructions. During execution, a general purpose register is reused once the value it stores is no longer needed. However, if such a register is needed but none are available, a register's contents is temporarily stored in the executing process' memory space (hereafter referred to as simply memory) to be retrieved when needed. Modifying a value stored in memory often requires reading the value into a register, modifying the value in the register, and writing the value back to memory. Intel64 and AMD64 architectures have 16 general purpose registers each 64 bits in size [25, 10].

The cache hierarchy comes after the registers in the memory hierarchy. It typically contains three separate levels and is designed to mask the memory wall problem. Each level is referred to as the LN cache where $N = 1, 2, 3, \ldots$. The value of N indicates the level's distance from computation (i.e., CPU core). Unlike registers, caches are CPU public. Even though a cache may have only one corresponding CPU core, its contents can be modified indirectly by other CPU cores through the cache coherence protocol. For modern processors, the cache hierarchy is located on the CPU die. In general, the farther from computation, the larger the capacity but the higher the access latency. The L1 cache is the cache level located closest to computation. It has the lowest access latency and has the smallest capacity. There is typically one L1 cache per CPU core. Further, it is typically a split cache. This means that the cache is divided into an L1d cache and an L1i cache, which store exclusively data and instructions respectively. Splitting the L1 cache has been found to increase parallelism, simplify caching logic, and reduce access latency [40]. Each subsequent level (i.e., L2 and L3) has a progressively larger distance from computation, larger capacity, and higher access latency. The L2 and L3 caches are typically unified caches. This means that they contain data and instructions intermixed. While each core typically gets its own L2 cache, the L3 cache is typically shared amongst all cores of a CPU. All levels of the cache are commonly composed of SRAM.

Data stored in one level of the cache may or may not also be in the lower levels of the cache. This is referred to as the cache inclusion policy. If the data in a higher level of the cache is also required to be present in a lower level, then the lower level is inclusive of the higher level. If the data in the higher level must not be present in the lower level then the lower level is exclusive of the higher level. Otherwise, the relationship among the levels is called non-inclusive non-exclusive. In the Intel Sandy Bridge microarchitecture, the L3 cache is inclusive of the L1 and L2 caches. Further, the L2 cache is non-inclusive of the L1

cache (suggesting that it is non-inclusive non-exclusive) [24].

Main memory follows the cache in the memory hierarchy. It is typically composed of dynamic random-access memory (DRAM), a high density and low cost memory technology. Finally, stable storage resides at the bottom of the hierarchy and may be one of many memory technologies: e.g., hard disk drive (HDD) or solid-state drive (SSD).

Moving down the hierarchy, distance from computation and storage capacity increase. As a result, so too does access latency. This is because more time is needed for signals to propagate a farther distance. Further, a larger size typically means more complexity, which also means more time. Smaller memory structures benefit more from being placed closer to computation. As a result, larger memory structures are pushed farther from computation.

3.1.3 Cache Access

A cache line is the smallest unit of data that can be stored within a level of the cache. A cache line is typically 64 bytes in size and is uniform across all levels of the cache. As a result, even if only one bit of data is desired to be stored in the cache, a cache line sized region must be stowed instead. The data stored in a cache line corresponds to a cache line sized region of memory whose base address is a multiple of the cache line size. In this work, *cache line* refers to the storage location in the cache and *data line* refers to the data that can be stored in a cache line. At any given time, a cache line will store at most one of any of a multitude of data lines.

A cache hit occurs when a memory location is accessed and its value is found in the cache. A cache miss occurs when a memory location is accessed and its value is not found in the cache. There are four main types of cache misses in modern multiprocessors: compulsory, capacity, conflict, and coherence. A compulsory cache miss occurs when a program performs a memory access to data that it has never previously accessed. These cache misses are unavoidable unless the data is prefetched. A capacity miss occurs when a program accesses data that was previously in the cache but is replaced with other data because of the cache's limited capacity. A conflict miss occurs when a program accesses data that was previously in the cache but is replaced with other data even though not all cache lines are utilized at the time of replacement. (Fundamentally, this is caused by the restrictions on which data lines can be stored in which cache lines.) Lastly, a coherence miss occurs when a program accesses data that was previously in the cache but is invalidated in order to maintain a coherent view of memory by the CPU cores.

3.1.4 Cache Coherence

Many modern (multicore and multiprocessor) computer systems support shared memory. This means that all CPU cores of the system can read and write to any memory location in the shared memory space. This complicates the usage of caches, which store a mapping from data address to the value currently stored at that address. This mapping needs to be made coherent across the caches of possibly several CPU sockets, where each cache is hierarchical in nature. With possibly multiple copies of each address/value mapping stored across several caches, an update to any mapping must be communicated to all caches storing a copy. Otherwise, the view of a core may be incoherent, or out-of-date with respect to the rest of the shared memory system. Any mechanism for keeping the caches coherent needs to make sure that each core sees the writes of all other cores. Further, for each memory location, there must be total ordering of all reads and writes made of all cores. Most importantly, this all must be done quickly. A cache coherency protocol is used to ensure that each core has a coherent view of the shared memory system. To ensure timeliness, this protocol is implemented in the hardware.

There are two main approaches to maintain coherence: snooping and directory-based. Within each cache is a coherency controller that keeps the cache coherent. In a snooping cache coherence protocol, each cache's coherency controller snoops the memory bus in order to inspect transactions (reads and writes). When the coherency controller observes a transaction on a memory location that its cache contains, then the controller modifies its cache's contents to ensure coherence. When a core performs a transaction on a cache line, the coherency controllers of all caches learn of this modification and update their cache accordingly.

In a directory-based cache coherence protocol, a directory exists that contains information on which caches contain which cache lines. When a cache line is newly placed in a cache, its coherence controller updates the directory to indicate that the cache contains the cache line. When a cache line in a cache is modified, the coherence controller queries the directory to determine which other caches contain the cache line. The controller then notifies the others of the modification so that they can maintain coherence. Transactions are ordered at the directory, meaning that each operation on the directory happens sequentially. When a coherence controller learns of a modification to a cache line that its cache contains, a common action is to simply invalidate the modified cache line. When the cache line is next accessed, a cache miss occurs and the cache line must be retrieved from another memory structure (e.g., a structure lower in the memory hierarchy or possibly another cache). This is the general behavior of write-invalidate protocols. However, snoopy coherence protocols in particular can instead use a write-update protocol. When a

coherence controller learns of a modification, it updates the cache line stored in its cache with the updated value that it learned from snooping on the bus.

While cache coherence is typically very fast as it is implemented in hardware, it can introduce non-trivial overheads in certain situations. To study one of these situations that is relevant to this thesis, the MESI protocol is examined in further detail. It is a snoopy and write-invalidate cache coherence protocol and is widely used in one form or another. Both Intel Core i7 and AMD Opteron processor families use extended versions of the MESI protocol [22]. The observations drawn from this examination of the MESI protocol are applicable to most snooping and directory-based cache coherence protocols.

The MESI protocol gets its name from the four states that a cache line can be in: modified, exclusive, shared, and invalid. A cache line in the modified state is dirty, meaning that it differs from the value in main memory as a result of a modification. The modified cache line needs to be written back to main memory. At that point, the cache line is no longer in the modified state. A cache line in the exclusive state is both clean (i.e., matches main memory) and is present in exclusively the current cache. A cache line in the shared state is both clean and present in the current cache and possibly other caches. A cache line in the invalid state can no longer be used and is no longer considered in the cache.

Figure 3.2 presents the state diagram of a cache line following the MESI protocol. In the diagram, each of the four states of a cache line in some cache are depicted as circles. The arrows represent the possible state transitions of the cache line. Transitions are labeled with the corresponding `input/output`. The solid lines are the transitions caused by a core, to which the cache is local, reading from (PrRd) or writing to (PrWr) the cache line. The dashed lines are the transitions caused by coherence messages, produced because another core read (BusRd) or wrote (BusRdX) to the cache line, that the coherence controller snooped on the bus. The output of these state transitions is a flush operation, meaning that the cache line is written back to main memory.

Using the MESI protocol, successive writes to the same cache line can have a large overhead. A cache line under high contention that is updated by multiple cores in parallel may have one core write to the cache line immediately after another core has written to it. In this scenario, each time a core writes to the cache line, it is not in the core's local cache because it was invalidated by another core's write. Further, each write results in the cache line getting invalidated in all other core local caches. Each write involves multiple steps including flushing the cache line to main memory, invaliding the cache line in all other caches, and retrieving the cache line from either main memory or another cache. This pattern is particularly time consuming on modern multiprocessor systems, where communicating caches may not be on the same processor die. As a result, there has been

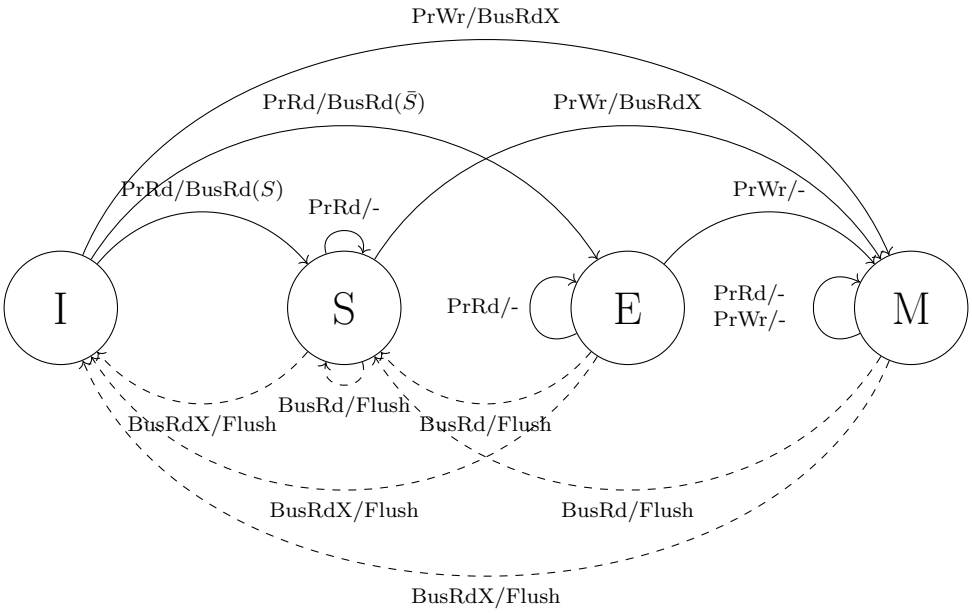


Figure 3.2: State diagram of the MESI protocol.

much research on detecting and avoiding such patterns.

3.2 Concurrency

3.2.1 Overview

In a typical computer system, the operating system (OS) acts as a middleman between the user and the hardware. It is responsible for managing the hardware resources of a computer and serving client requests in the form of running requested programs. An OS typically must balance several often conflicting objectives including fairness and security while keeping efficiency high.

One of the most important of these hardware resources is the CPU cores. Without a core, a program never "runs" as the CPU core executes the instructions of the program. The OS manages access to the cores of a system through kernel threads. In general, a thread is an execution context. It is all of the information needed by a core to pause and resume execution of a sequence of instructions. In practice, this minimally consists of CPU register values and a stack. More specifically, a kernel thread is a thread that is managed in kernel space by the OS. Each program has at least one kernel thread.

Most modern OSes manage access to the cores by scheduling (with input from the user) when, where (i.e., which core), and how long each thread executes. The OS assigns each thread a slice of time during which it executes on a given core. Once its time slice has passed, the thread is preempted (interrupted) and supplanted with another thread, which executes for its time slice. This is called preemptive scheduling. Through kernel thread scheduling, the OS allows for the appearance of multiple processes running at once, which is called multiprocessing. Further, the OS chooses a thread scheduling algorithm that satisfies its objectives (e.g., fairness). For example, Linux's default thread scheduler is called the Completely Fair Scheduler [1]. It is designed to model an ideal, precise multi-tasking CPU no matter the underlying hardware. Put another way, threads are scheduled such that it appears that all threads are running in parallel and are receiving an equal proportion of CPU resources.

3.2.2 Threads

Besides the kernel thread, another type of thread is the user thread. A user thread is similar to a kernel thread. However, instead of being managed in kernel space by the OS,

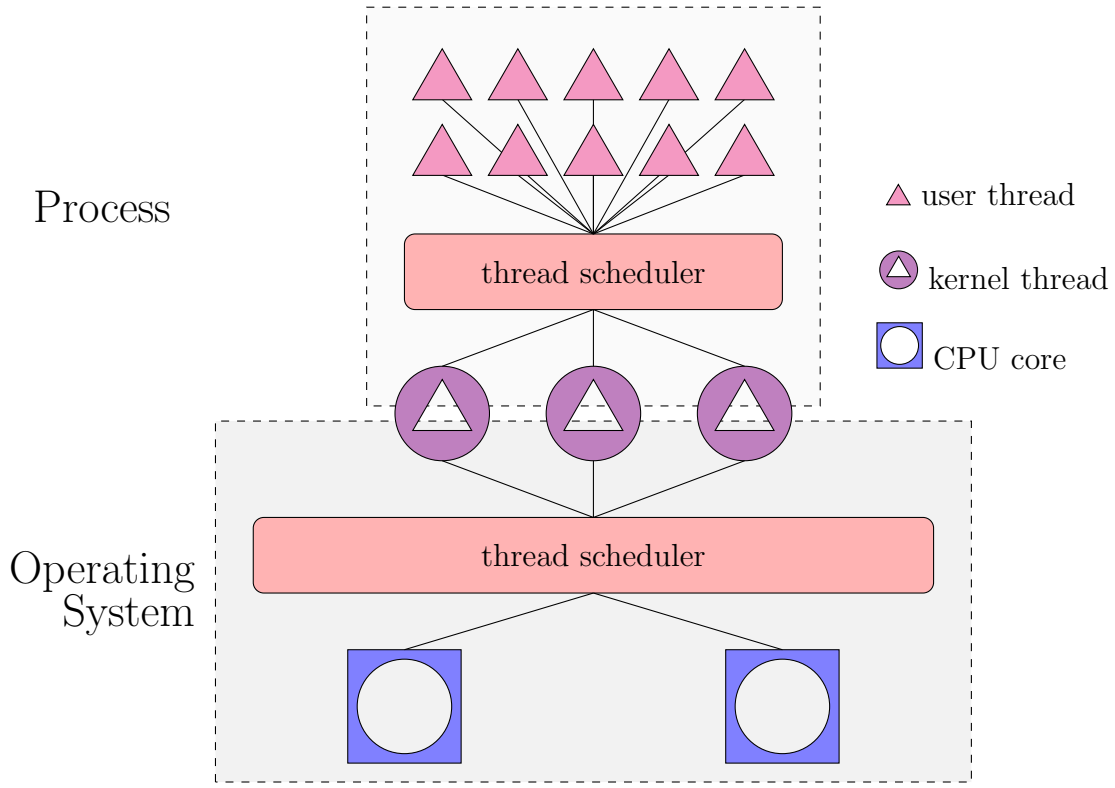


Figure 3.3: Visualization of the relationship between user threads, kernel threads, and CPU cores.

it is managed in user space by a user threading runtime. Figure 3.3 shows the relationship between each of these types of threads and the CPU cores. User threads can be thought of as executing on kernel threads in the same way that kernel threads execute on CPU cores. By using a user threading runtime, an application has more control over how its threads get scheduled. As a result, it can select a scheduling algorithm that better matches its objectives and, therefore, achieves better performance. In applications that use a user threading runtime, it is common for the number of kernel threads spawned by a process to be roughly equal to the number of cores assigned to execute the process. Further, it is common for the number of user threads to be several orders of magnitude greater than the number of kernel threads.

A common characteristic of user thread scheduling algorithms is that they are (mostly) cooperative. This means that executing threads periodically, voluntarily yield access to CPU resources to allow other threads to execute. This is in contrast to preemptive scheduling, which not only ensures fairness amongst threads but also allows the OS to rapidly handle events requiring immediate attention. Cooperative thread scheduling is used, in part, to give the application developer more control over thread scheduling as they will know how to ensure fairness amongst the threads of their application. Additionally, context-switching and synchronization overheads introduced by preemptive scheduling can be avoided. Many applications do not need to handle events with as much immediacy as the OS and, therefore, don't require the ability to preempt executing tasks. However, cooperative scheduling can (intentionally or unintentionally) permit unfairness between threads. In extreme cases of unfairness, this can lead to deadlock.

The threading model defines the relationship between user threads, kernel threads, and cores. Knowing the threading model of a user threading runtime is important as it puts restrictions on where user threads can execute. The relationship can be denoted by `X:Y` where `X` denotes the number of user threads and `Y` denotes the number of kernel threads. Figure 3.4 shows a visual representation of several different threading models. A threading model of `1:1` describes the scenario where one user thread maps to one kernel thread. More generally, a threading model of `N:N` describes the scenario where each of `N` user threads uniquely maps to one of `N` kernel threads. In both scenarios, user threads and kernel threads are indistinguishable. Meanwhile, `M:1` describes the scenario where `M` user threads all map to one kernel thread. This threading model does not permit parallelism and, as a result, is infrequently or no longer employed. For example, the Java Runtime on Solaris version 2.6 used a `M:1` threading model [2]. Lastly, `M:N` describes the scenario where each of `M` user threads maps to all of `N` kernel threads. Put another way, any user thread can execute on any kernel thread. Unsurprisingly, the scalability of this threading model has made it the most frequently implemented by user threading runtimes: Go [6], Libfibre [26],

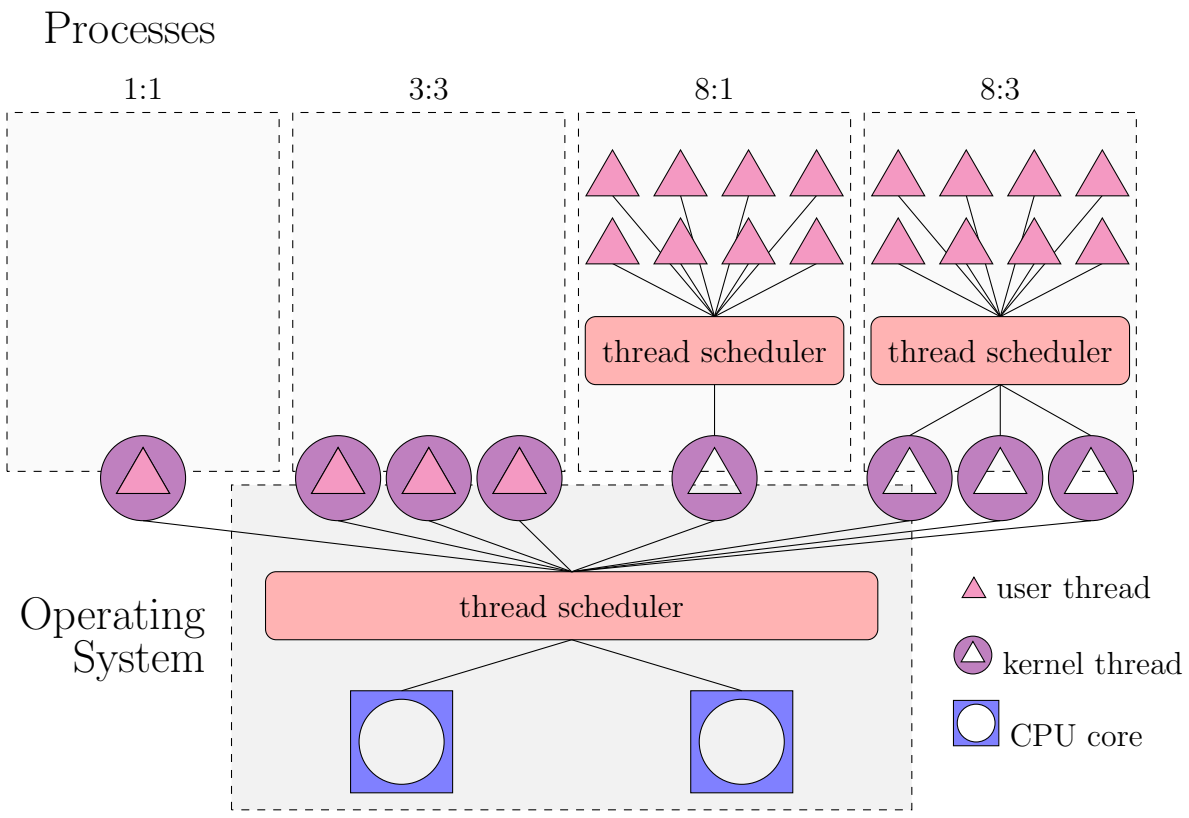
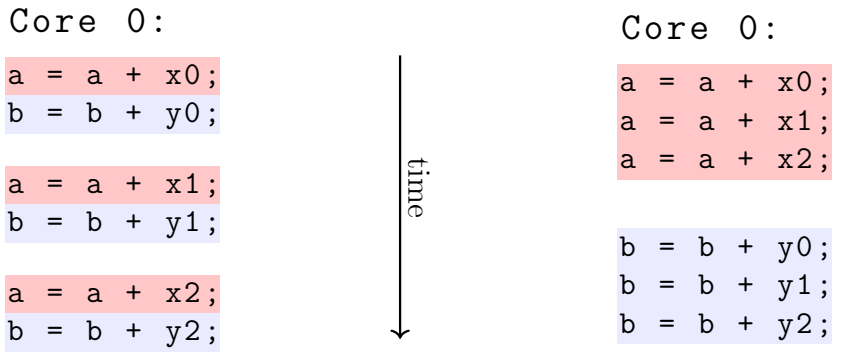


Figure 3.4: Visualization of different threading models.



(a) Without execution batching.

(b) With execution batching.

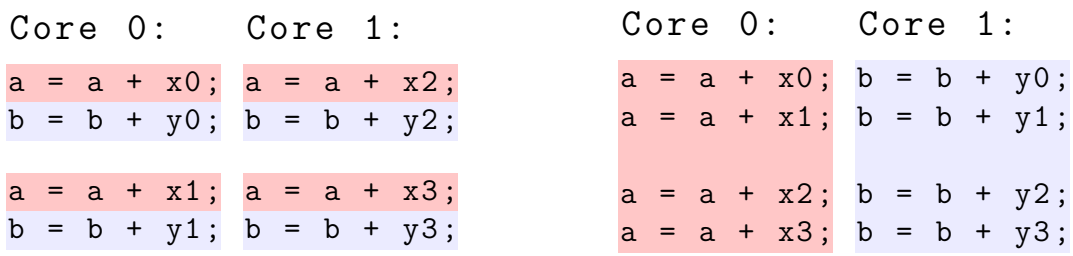
Figure 3.5: Instructions without and with execution batching.

C∀ [4], μC++ [16], Shenango [34], and Arachne [37].

3.3 Execution Delegation

3.3.1 Execution Reorganization

Execution reorganization is the general technique of restructuring the execution of a program in order to improve data locality. Put another way, execution reorganization aims to move computation to where the data resides instead of moving the data to the computation. Within a single computer system, improved data locality stems from reduced data movement both among levels of the memory hierarchy and across memory hierarchies. Data often moves among the different levels of the memory hierarchy when the duration between accesses to the data by the core is large. Between accesses, the core accesses other data and this data is also stored in the core's cache. The longer the duration between accesses, the higher the likelihood that data is removed from the cache in order to make room for new data. Data often moves across memory hierarchies when multiple cores of the system access and modify the same data. If multiple cores store some data in their local caches and at least one core modifies the data, then the data is invalidated from the cache of all non-modifying cores. Each of these cores must then retrieve the data again. In order to reduce these two types of data movement, execution reorganization can be performed in two corresponding ways: temporally and spatially.



(a) Without execution delegation.

(b) With execution delegation.

Figure 3.6: Instructions without and with execution delegation.

Temporal execution reorganization, referred to as execution batching or simply batching, aims to decrease the amount of data movement among different levels of the memory hierarchy. This is done by modifying the schedule of when a core executes operations in order to group operations that have similar working sets. In this context, the working set is the set of locations in memory accessed during the execution of an operation. As a result, the average duration between accesses to the same data is shortened. For example, Figure 3.5 depicts a sequence of instructions executed by a core without (3.5a) and with (3.5b) batching applied. In the figure, all operations on variable `a` are batched together such that they are executed one after the other. The same happens to operations on variable `b`.

Spatial execution reorganization, hereafter referred to as execution delegation or simply delegation, aims to decrease the amount of data movement between the local caches of different cores. This is done by having the cores delegate the execution accessing the data to a subset of the cores. As a result, data moves only between the local caches of the subset of cores where execution is delegated. If execution is delegated to only one core, then no data movement occurs. For example, Figure 3.6 depicts a sequence of instructions executed by two cores without (3.6a) and with (3.6b) delegation applied. When delegation is applied, Core 0 delegates its operations on variable `b` to Core 1. Similarly, Core 1 delegates its operations on variable `a` to Core 0. In this work, delegation is the only form of execution reorganization under examination. A study of batching is left to future work.

3.3.2 Execution Delegation Approaches

Delegation is ubiquitous throughout computer system design. This is possibly because, once a task has been decomposed into distinct parts (as programmers often do), it is beneficial to delegate one or more of those parts. Depending on the situation, delegation

can be used to improve security, maintainability, scalability, and performance.

The remote procedure call (RPC) is a prominent example of delegating execution between separate processes. An RPC is used when one program wants to request the services of another process by having it execute a certain procedure with specified arguments. Both processes are typically located on separate computer systems. Once an RPC is performed, the caller can block or can continue executing in parallel with the procedure until the result of the procedure is required. RPC is a request-response protocol implemented through message passing. One process (the client) sends a message containing the procedure to be executed and its arguments to another process (the server). Once finished executing the procedure, the server packs the result into a new message and sends it back to the client.

Within a single process, thread migration can be used to facilitate delegation across cores of a single computer system. Thread migration is the process of pausing the execution of a thread on one core and resuming execution on another core. Thread migration is a common occurrence on most computer systems. The operating system's thread scheduler attempts to balance work across the cores of a system to achieve its objectives (e.g., maximum resource utilization). If a thread could request of the thread scheduler as to when and where it would like to be migrated, then thread migration could be used to implement execution delegation. To perform execution delegation, the thread would only need to make a scheduling request before and after the execution to be delegated. User threading runtimes are a straightforward choice for performing execution delegation using thread migration. Since both the user threads and user threading runtime belong to the same process space, user threads can easily and safely make such requests of the user thread scheduler.

Chapter 4

Session-Loop Pattern

4.1 Overview

In order to study the effect on performance of execution delegation, it is important to understand how delegation affects cache usage behavior. To this end, the *session-loop pattern* is proposed. Through this pattern, the class of applications fitting this pattern can be studied.

The session-loop pattern is defined as follows. Each session repeatedly performs the same sequence of actions. Put another way, each session can be thought of as an independent execution of a loop of actions. This loop is referred to as the *session loop*. Depending on the context, session loop is used in this work to refer to either a sequence of actions or the corresponding sequence of instructions performing the actions. Figure 4.1 visualizes an example session loop with two sessions repeatedly performing corresponding sequences of actions. Further, typically, an application fitting the session-loop pattern is highly concurrent such that the number of concurrent sessions vastly outnumbers the number of parallel resources (i.e., CPU cores).

The session-loop pattern can be observed in applications fitting the thread-per-session paradigm. In such applications, each session is represented by a software thread with its own execution stack. For many applications that could adopt the thread-per-session paradigm, event-driven programming is typically used instead. Following this pattern, the application must explicitly manage the set of sessions by reacting to external events such as I/O events reported by the OS. However, Karsten et al. [26] demonstrate that it is possible for the thread-per-session paradigm to have equal or superior functionality, efficiency,

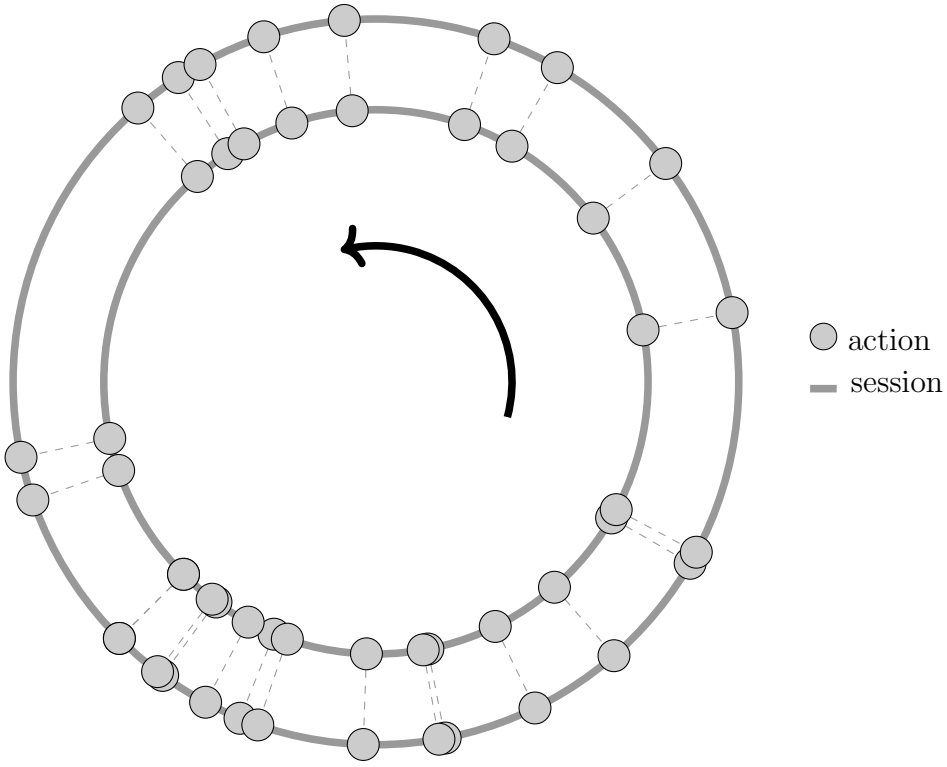


Figure 4.1: Visualization of a session loop.

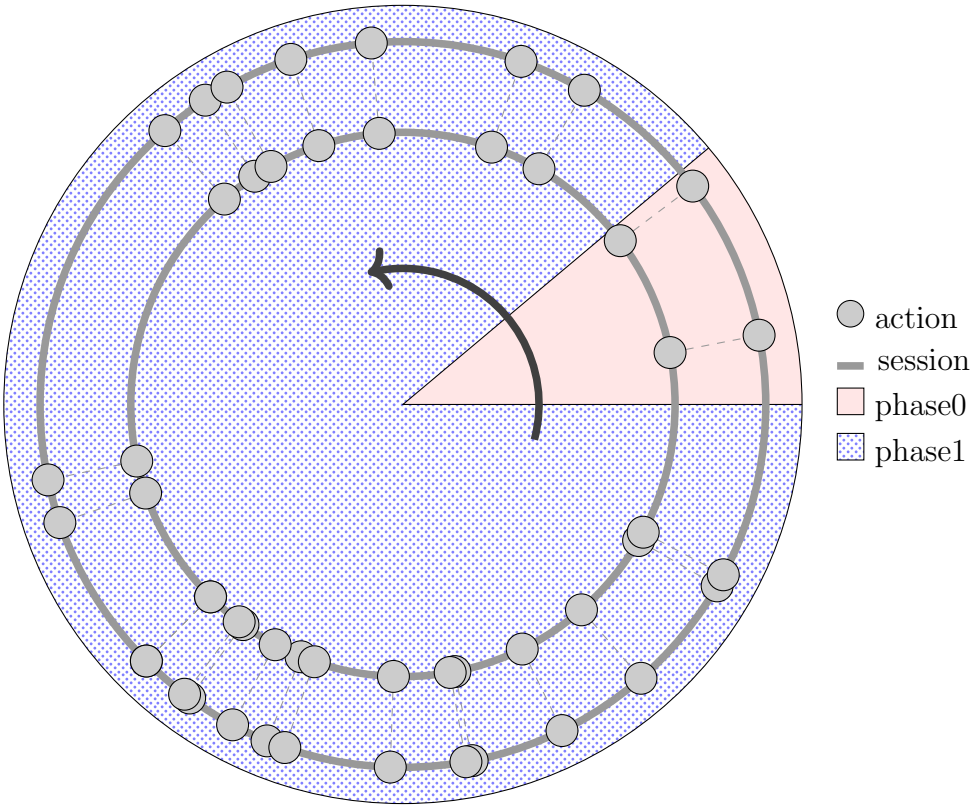


Figure 4.2: Visualization of a session loop split into two phases.

performance, and scalability compared to event-driven programming. In particular, they found this to be true when sessions are represented using user threads.

Various network-based server applications satisfy these requirements. For example, databases and webservers handle stateful sessions that number several orders of magnitude greater than the number of cores. Each session consist of repeated request/reply interactions.

The session loop can be broken down into one or more *phase*s, where a phase is a contiguous slice of the session loop. Put another way, if the session loop is a circle, a phase is a sector of that circle. Figure 4.2 represents Figure 4.1 after being split into two phases. Like before, depending on the context, phase is used in this work to refer to either a subsequence of actions or the corresponding subsequence of instructions performing the actions.

For example, consider a webserver. A client connects to the webserver in order to

operate on the data managed by it. From the perspective of the webserver, a session can consist of all requests received from, processed, and responses sent to a given connected client. Figure 4.2 may represent the execution of the webserver having established two sessions. The session loop of the webserver is composed of receiving a request, processing the request, and sending a reply. In Figure 4.2, phase 0 may represent the sector of the session loop related to receiving a request. Phase 1 would then represent remainder of the session loop (i.e., processing the request and sending a reply).

Chapter 5

Microbenchmark

5.1 Overview

This work investigates execution delegation via user thread migration in the context of an application fitting the session-loop pattern. To do this, a microbenchmarking tool named *Accesstest* is used. The impact on the performance of Accesstest from delegating a phase of the session loop is studied.

5.2 Design and Implementation

Accesstest is implemented in approximately 4,000 lines of C++ code. The code makes heavy use of object oriented design and templates. This allows for highly extensible and flexible code. Experiments run using Accesstest are also highly configurable, as it accepts a wide range of command line options. Accesstest is composed of three stages: path construction, session creation, and execution. At the completion of all three stages, performance statistics are reported. Further documentation can be found in Appendix B and in the code repository.

5.2.1 Path Construction

At the core of Accesstest is a circular pointer chasing path. Pointer chasing is the process of performing a sequence of memory accesses that follow an irregular access pattern. A

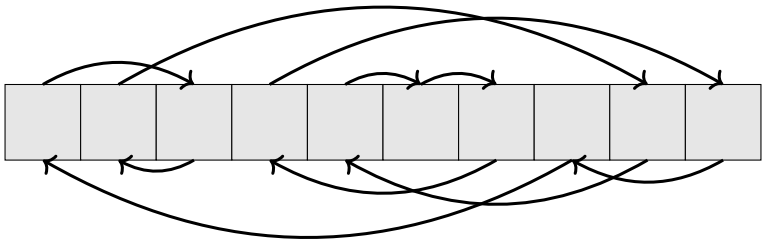


Figure 5.1: Example of a circular pointer chasing path.

pointer chasing path is therefore a set of memory locations such that each memory location holds a pointer to the next memory location in the path. It is circular in the sense that, once all elements in the path have been traversed, the traversal starts again at the first memory location. Figure 5.1 depicts an example circular pointer chasing path and the order in which elements are accessed as the path is traversed.

A traversal of the pointer chasing path models the memory accesses made while executing the session loop. In this way, Accesstest simulates the memory accesses to an application's working set performed by sessions executing a session loop in parallel. Accesstest additionally simulates reads and writes to these memory locations. Such pointer chasing is employed by Pingali et al. [36] to study the effects of temporal reorganization on program components to improve the data locality of memory intensive applications. Further, it is employed by Srinivasan [43] in microbenchmarks to study the effects of delegating the execution of critical sections via asynchronous message passing.

The pointer chasing path is implemented as a path of elements stored within a buffer as depicted in Figure 5.2. The buffer is divided into data lines (columns labeled 0 through 5 in Figure 5.2) which are subdivided into elements (cells labeled 0 through 20 in Figure 5.2). The location in the buffer at which each element is stored is significant because each element represents a memory access to the data line containing the element. The number of elements in a data line corresponds to the number of times that the data line is accessed during each traversal of the pointer chasing path. The buffer and, in particular, the data lines constituting it represent the working set of the application. An element compactly holds both an offset to the next element in the pointer chasing path and auxiliary information. The offset is represented as the index from the base of the buffer. The auxiliary information consists of two boolean values: whether the next element in the path represents a write operation and whether the end of a phase has been reached. Since information is compactly stored with each element, as the path is traversed, extraneous and unexpected

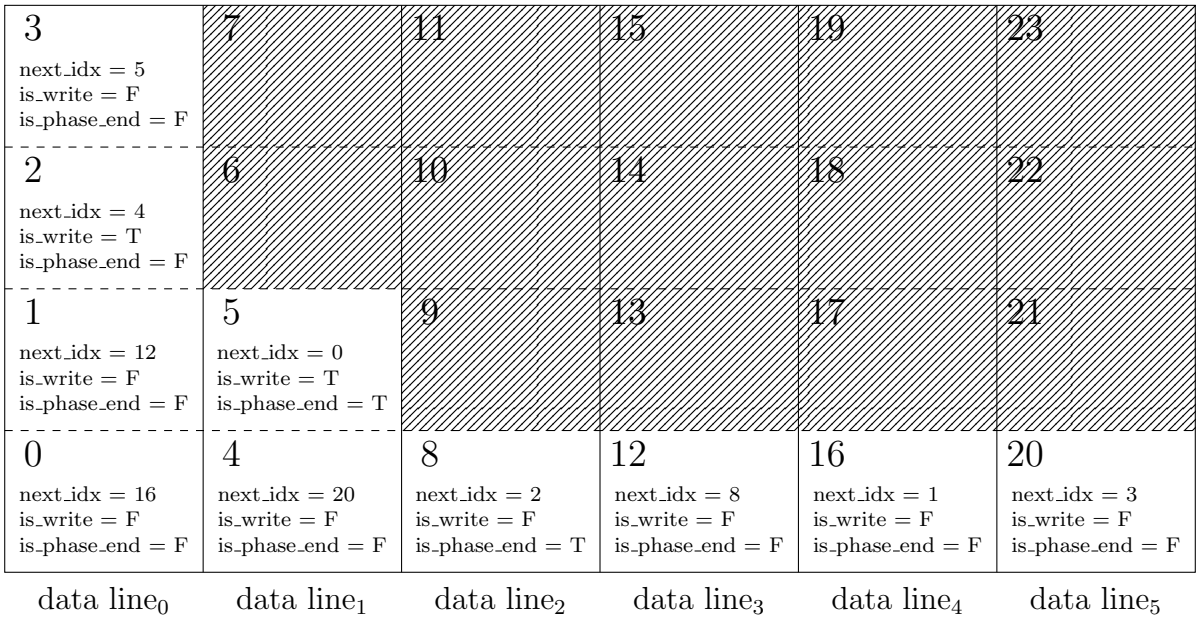


Figure 5.2: Example of a buffer containing a pointer chasing path.

memory accesses are prevented.

The number of elements in the pointer chasing path is restricted by the size of the buffer, which is specified by the user. When the buffer is allocated, it is done so with consideration of the cache. Put another way, the buffer is allocated such that the mapping between data lines of the buffer and cache lines is uniform. This means that the results reported by Accesstest are not influenced by conflict misses stemming from poor cache placement [27]. To this end, the buffer is allocated from contiguously physically addressed memory. Once allocated, the buffer is divided into two subbuffers along a boundary between two data lines. Each subbuffer holds the data lines accessed during one of the two phases of the session loop.

To install a pointer chasing path in each subbuffer, elements of the path are pseudo-randomly distributed amongst data lines of the subbuffer such that element counts approximate a real access pattern using a Zipf distribution with shape parameter 1. All data lines are accessed at least once per iteration of the pointer chasing path. Approximation is needed since each element in the path can only point to one next element. Further, the number of accesses made to a data line is bounded by the number of elements that can fit in a data line. For example, for data lines of size 64 bytes, only 16 Elements can fit in each data line since Elements are 4 bytes in size.

The two resulting pointer chasing paths represent the two phases of the session loop. As such, the last element of each of these paths is marked to indicate that the end of the corresponding phase has been reached. Using this mark, a session traversing the pointer chasing path knows when a phase has ended and can act accordingly. For example, recall that, in order to implement execution delegation using a user threading runtime, user threads migrate back and forth across the cores of a system. Using this mark, a user thread knows when to migrate (either back or forth) to perform execution delegation. Finally, the ends of the paths are concatenated to form a circular pointer chasing path.

For example, in Figure 5.2, the first phase consists of the path of elements at indices 2, 4, 20, 3, and 5. The second phase consists of the path of elements at indices 0, 16, 1, 12, and 8. Recall that each element represents a memory access to the data line containing the element. Therefore, the first phase consists of accesses to data lines with indices 0, 1, 5, 0, and 1. The second phase consists of accesses to data lines with indices 0, 4, 0, 3, and 2.

All elements in the pointer chasing path are pseudorandomly assigned to be either memory reads or memory writes. The preceding element in the path is marked to indicate this information. Using this mark, a session traversing the pointer chasing path knows if the next element in the path represents a read or write memory access. With this information, a session can simulate the corresponding memory access. In this way, Accesstest incorporates the effects of cache coherence in its results. How a session makes use of this information is described in detail later in this section. The percentage of elements assigned to be writes is specified by the user on the command line and is called the *update ratio*.

After the pointer chasing path is constructed, at least one of the CPU core's cache hierarchies contains pointer chasing path entries. To prevent these cached entries from influencing the results of Accesstest, the cache hierarchy of all cores is wiped immediately before the execution stage using the following technique. The cache hierarchy is assumed to have a least recently used replacement scheme. The cache wiping algorithm is implemented by allocating and accessing every byte of a contiguously physically addressed region of memory that is unrelated to the pointer chasing path. Using a sufficiently large contiguously physically addressed region of memory, all cache lines (including cached pointer chasing path data) are evicted no matter what the cache's associativity. In all experiments, a 1 GB hugepage is allocated since the cache hierarchy has a capacity of less than 1 GB in size.

5.2.2 Session Creation

Accesstest makes use of Libfibre [26], a M:N user threading runtime without preemption, for creating user threads to represent the sessions. Accesstest executes the two phases of the session loop either synchronously or by delegating the execution of a phase. If synchronously, the Libfibre runtime is initialized so that user threads execute any phase of the session loop on any kernel thread.

However, if delegation is performed, then only certain phases of the session loop are executed on each kernel thread. In this way, kernel threads, and therefore cores, are specialized to execute certain phases. To achieve this, kernel threads in the Libfibre runtime are grouped into scheduling domains corresponding to the phases of the session loop. Within a domain, work stealing is permitted. Across domains, work stealing is permitted if enabled by the user through the command line. Additionally, if delegation is performed, each session is informed of the delegated kernel thread before the execution stage begins.

5.2.3 Execution

In order to traverse the pointer chasing path during the execution stage, each session executes a function similar to the `loop` function presented in Figure 5.3. For completeness, a simplified definition of the data structure representing an element is also presented. Sessions execute the session loop by traversing the pointer chasing path and simulating the memory operations (i.e., read or write) specified in the elements. The assembly code generated by the compiler is also manually inspected to ensure that each step in traversing the pointer chasing path requires touching only one memory location: the location storing the next element in the path.

An element assigned to represent a memory read operation simulates the operation in the obvious way: the element is read from memory and traversal of the pointer chasing path continues. However, it is not as simple for an element assigned to represent a memory write operation since each element must be *read* to determine the next element in the pointer chasing path. A read operation with the side effects of a write operation (i.e., cache line invalidation in all other core's private cache) is ideal. In cache coherency protocols, such an operation exists and is called read for ownership. However, on all systems tested, no corresponding assembly instruction exists. As a consequence, memory write operations are simulated using a read of the corresponding element followed by a dummy write to that element (see the `dummyWrite` function in Figure 5.3). The results presented at the end of this section show that dummy writes affect performance and, therefore, are not elided by the hardware.

```
struct Element {
    volatile unsigned int value;

    /* ... define constants ... */

    inline size_t getIndex() {
        return (value & INDEX_BITMAP) >> INDEX_BITSHIFT;
    }
    inline size_t isWrite() {
        return (value & WRITE_BITMAP);
    }
    inline bool isPhaseEnd() {
        return (value & PHASE_END_BITMASK);
    }
    inline bool isFinished() {
        return (FINISHED_SENTINEL == value);
    }
    inline Element dummyWrite() {
        Element tmp = *this;
        *this = tmp;
        return tmp;
    }
};

void loop(Element* buffer, size_t index, bool isWrite) {
    Element* elementPtr = &buffer[index];
    Element element = (isWrite ? elementPtr->dummyWrite() :
                                 *elementPtr);
    while (true) {
        if (element.isFinished()) break;

        index = element.getIndex();
        isWrite = element.isWrite();
        if (element.isPhaseEnd()) {
            /* ... possibly yield or migrate ... */
        }

        elementPtr = &buffer[index];
        element = (isWrite ? elementPtr->dummyWrite() :
                             *elementPtr);
    }
}
```

Figure 5.3: Simplified code for traversing pointer chasing path.

If Accesstest is configured so that each session synchronously executes the phases of the session loop, then each user thread yields execution after each iteration of the session loop. This is done because Libfibre is cooperative. This means that yielding is required to ensure that each user thread gets a chance to execute. Yielding at the end of an iteration of the session loop is also done to better mimic applications satisfying the session-loop pattern. For example, a webserver halts a session while waiting for each new request.

Otherwise, if Accesstest is configured so that sessions delegate execution of a phase, then each user thread migrates back and forth between two cores before and after the execution of the phase. To perform migration using Libfibre, each user thread explicitly indicates (i.e., in code) to the runtime to where it should be migrated. The last element of each phase in the pointer chasing path is marked. User threads learn of the end of each phase through this marker. Yielding is a consequence of migration so, unlike the synchronous execution of the phases, explicit yielding is not needed.

After a user specified testing duration has passed, all sessions are notified as quickly as possible that the test is complete. The sessions are notified by overwriting elements in the pointer chasing path with a sentinel value. Once encountered, a session immediately breaks out of the session loop and exits. Signaling the sessions in this way requires a constant amount of work per session and requires the sessions to perform no additional memory access to check for microbenchmark completion.

5.3 Evaluation

The performance of the Accesstest microbenchmark is evaluated when the execution of a phase of the session loop is delegated. The working set size and the update ratio of that working set size is varied to understand how performance changes across different levels of the cache hierarchy and with different cache coherence overheads.

5.3.1 Configuration

The machine on which the microbenchmark is run contains an Intel Xeon Processor D-1540. Each of its eight cores has a private L1d cache of size 32 KiB and private L2 cache of size 256 KiB. Additionally, each core has access to a shared L3 cache of size 12 MiB. Hyperthreading and CPU frequency scaling are disabled.

In each run of the microbenchmark, 100 sessions are spawned that concurrently execute a session loop split into two phases. During each phase, each session accesses a set of global

data lines where the set is disjoint from the set of data lines accessed in the other phase. The number of data lines accessed in each phase is equal and the working set size is the total number of data lines accessed during the session loop. Put another way, half of the working set is accessed by all sessions in one phase and the other half is accessed by all sessions in the other phase.

Libfibre, the user threading library, is configured to spawn two kernel threads on which user threads execute. In effect, two cores of the system are used. Thread pinning is employed to prevent kernel thread migration.

The performance of Accesstest is measured using throughput, which is defined as the number of memory accesses performed per second. An access can be performed by any session in any phase of the session loop. In the following plots, each point represents the average performance of Accesstest across 20 runs where each run executes for a 10 second interval. Across all points, the maximum coefficient of variation is less than 0.09.

The throughput is shown for Accesstest run in synchronous and delegated mode. *Synchronous mode* refers to Accesstest configured so that each user thread executes the session loop on a single core. While in synchronous mode, each of the two cores is specialized to execute one half of all user threads. *Delegated mode* refers to Accesstest configured so that each user thread executes the first phase of the session loop on one core and the second phase on the other core. While in delegated mode, each of the two cores are specialized to execute a single phase of the session loop.

5.3.2 Results

L1d Cache

Figure 5.4 shows the performance of Accesstest for working set sizes at and around the size of the L1d cache. When the update ratio is 0% and the working set size is smaller than that of the L1d cache, all memory accesses hit in the L1d cache since the working set can fit in the cache. Further, there is no cache coherence overheads from accessing the pointer chasing path as no cache lines are modified. Delegated mode's performance is far below that of synchronous mode–lower by as much as 41%. Additionally, for each configuration, performance appears to increase as the working set size increases within this range of working set sizes.

The difference in performance of these two configurations can likely be attributed to the difference in time performing user thread scheduling (i.e., yielding and migration). During synchronous mode, user threads yield execution to another user thread at the end of each

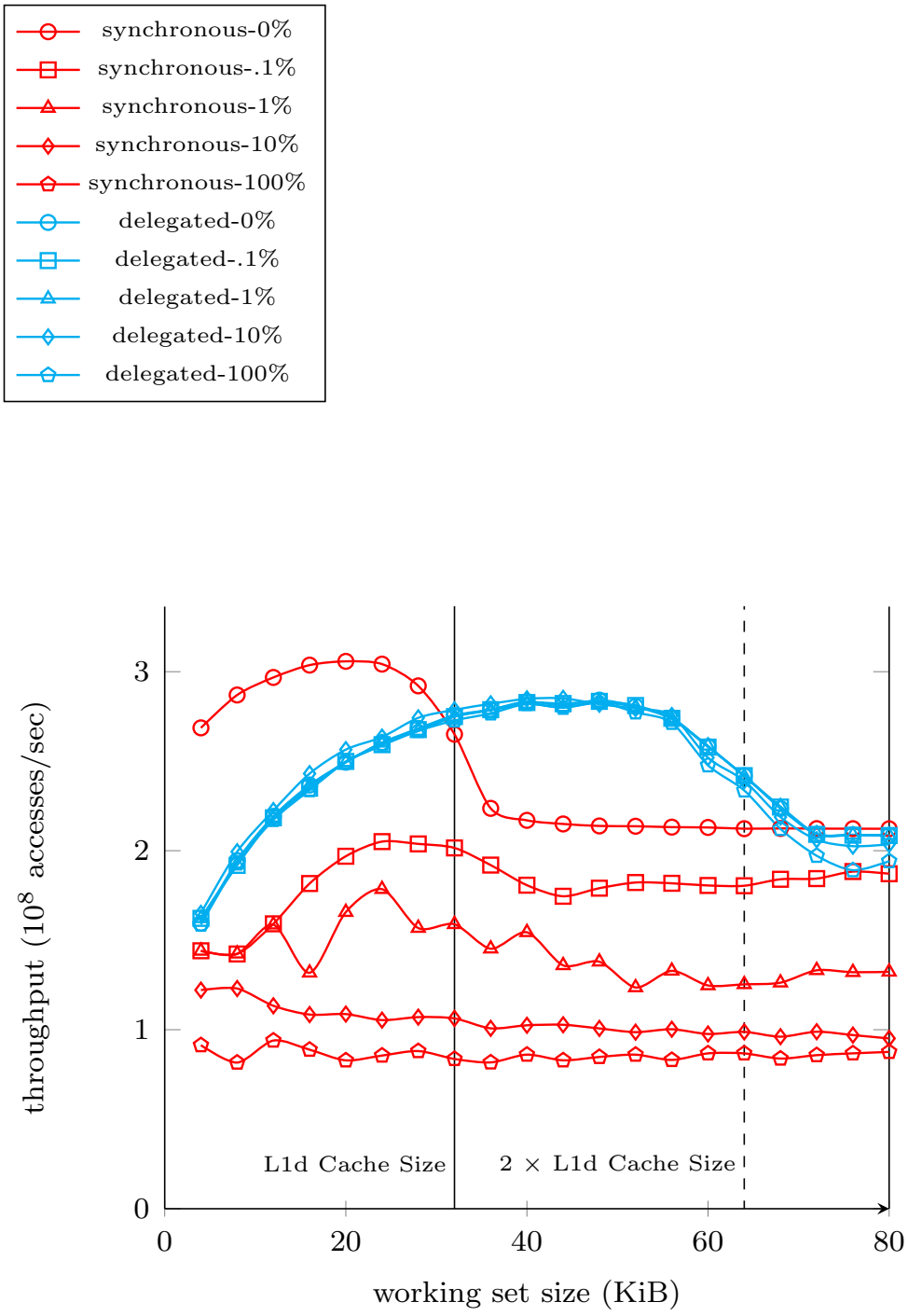


Figure 5.4: Performance of Accesstest for working set sizes around that of L1d cache and for varying update ratios.

iteration of the session loop. Meanwhile, during delegated mode, user threads migrate between the cores at the midpoint and end of each iteration of the loop. When compared with synchronous mode, which yields once per iteration of the session loop, delegated mode's poor performance can likely be attributed to the larger time spent migrating twice per iteration. Additionally, migration introduces some degree of cache pollution and cache coherence overhead. The cache of the migrated core is unlikely to contain all the data that a thread accesses. While the cache may contain the memory locations accessed through traversing the pointer chasing path, it is unlikely to contain values stored on the thread's stack. These values are likely to have been modified and stored in the previous core's cache.

Still, when the update ratio is 0% and the working set size is smaller than that of the L1d cache, the increase in performance as the working set size increases for both configurations can also be attributed to user thread scheduling. As the working set size increases, the time needed to traverse one iteration of the pointer chasing path increases and hence, thread scheduling occurs less frequently. The result is that, as the working set size increases, each session (and Accesstest as a whole) spends more time traversing the pointer chasing path than thread scheduling on average. Further, the side effects of thread scheduling that hurt performance (i.e., cache pollution and cache coherence overheads) are experienced less frequently because the experiment is timed. Throughput is therefore higher.

When the update ratio is 0% and the working set size is larger than that of the L1d cache but smaller than twice that size, roles are reversed: delegated mode outperforms synchronous mode by as much as 33%. In synchronous mode, accesses frequently (if not always) miss the core's L1d cache but hit at the L2 cache. This is caused by the fact that the working set size is larger than the L1d cache. Additionally, each data line is likely to have been evicted from the L1d cache between two successive accesses since data line access counts follow a Zipf distribution with shape parameter 1. As a result of this distribution, most data lines are accessed only once per iteration of the pointer chasing path.

Meanwhile, during delegated mode, thread migration results in each core's L1d cache containing one disjoint half of the working set. As a result, delegated mode does not experience L1d cache misses until the working set size is roughly twice the size of the L1d cache. As the working set size approaches twice that of the L1d cache, the performance of delegated mode approaches that of synchronous mode.

It is noteworthy that performance begins to drop at smaller working set sizes than expected: the size of the L1d cache for synchronous mode and twice the size of the L1d cache for delegated mode. This can likely be attributed to cache pollution introduced

by user thread scheduling. For example, each thread scheduling event requires memory locations, which are not associated with the pointer chasing path, to be accessed. These locations are, in turn, stored in the L1d cache and possibly evict a portion of the working set. In effect, performance drops earlier than expected because the actual working set includes both the pointer chasing path and the data accessed for user thread scheduling. This effect impacts all levels of the cache hierarchy but is not necessarily the only cause for premature performance drop.

As the update ratio increases from 0% to 0.1% and then to 1%, 10%, and finally 100%, the performance of synchronous mode falls dramatically while that of delegated mode remains relatively constant. The drop in performance of synchronous mode can be explained by cache coherence overheads. During synchronous mode, each thread performs the same set of memory accesses on the same set of memory locations. Additionally, threads execute in parallel on the two cores. As a result, when one core modifies a cache line, it must first acquire exclusive access over that cache line. This involves invalidating all copies of that cache line in any other core's private cache. As a result, when the other core accesses the same cache line at some later point in time, it must reacquire the updated cache line. A pointer chasing path is traversed one memory location at a time since each memory location stores a pointer to the next memory location in the path. As a result, a cache line must be reacquired because it is invalidated and hence, progress in traversing the pointer chasing path is stalled. The more updates that are performed, the more this stalling occurs and throughput drops.

Meanwhile, as the update ratio increases, the performance of delegated mode remains relatively constant. This is because each of the two cores is specialized to execute one of the two phases of the session loop, and each phase accesses a disjoint set of data lines. Since the set of data lines accessed by each core is disjoint, updating a cache line does not result in a cache line invalidation. Therefore, no cache line must be reacquired because it is invalidated, which means that no stalling occurs while traversing the pointer chasing path. When the update ratio is 100%, the performance of delegated mode is higher than that of synchronous mode by as much as 237%.

When the working set size is around twice the size of the L1d cache, there is a noticeable difference in the performance of delegated mode as the update ratio increases. This drop in performance can likely be attributed to the increased cost of simulating write operations (a load followed by a store) as compared to read operations (just a load).

For the L1d cache and all cache levels, the impact of the prefetcher is expected to be minimal as accesses made to locations in the buffer are pseudorandom. This is enforced by the approach used to distribute elements of the looping pointer chasing path amongst

the data lines of the buffer storing the path.

L2 Cache

Figure 5.5 shows the performance of Accesstest for working set sizes at and around the size of the L2 cache. Note that, in this figure, results are shown for working set sizes just larger than the largest shown in Figure 5.4. When the update ratio is 0% and the working set size is smaller than that of the L2 cache, all memory accesses hit in the L2 cache. Additionally, there are no cache coherence overheads from accessing the pointer chasing path as no cache lines are modified.

Unlike the results observed in Figure 5.4, synchronous mode and delegated mode have comparable performance. For both modes, working set sizes are large and times spent traversing an iteration of the session loop are long. As a result, the difference in time spent performing thread scheduling per iteration of the session loop (a yield at the end of an iteration in synchronous mode and a migration at the midpoint and end in delegated mode) is negligible. Hence, performance remains constant (instead of increases) as the working set size increases. This is dissimilar to the results in Figure 5.4 where performance increases as the working set size increases.

When the update ratio is 0% and the working set size is larger than the L2 cache, delegated mode outperforms synchronous mode by as much as 154%. The reason for this behavior is identical to that discussed for Figure 5.4 when the working set size is larger than that of the L1d cache. In synchronous mode, accesses frequently if not always miss the core's L2 cache but hit at the L3 cache. Meanwhile, in delegated mode, thread migration results in each core's private L2 cache containing one disjoint half of the working set. Therefore, delegated mode does not experience L2 cache misses until the working set size is roughly twice the size of the L2 cache.

As the update ratio increases from 0% to 100%, the performance of synchronous mode falls dramatically while that of delegated mode remains relatively the same. When the update ratio is 100%, the performance of delegated mode is higher than that of synchronous mode by as much as 384%. The same general behavior is observed for working set sizes around that of the L1d cache and has the same cause: cache coherence overheads.

There are several unexpected behaviors not previously exhibited that occur around cache line boundaries. For example, the performance of synchronous mode with update ratio 1% increases as working set sizes approach that of the L2 cache, the performance of synchronous mode with update ratio 10% and 100% decrease and then increase for working sets just larger than that of the L2 cache, and the performance of delegated mode

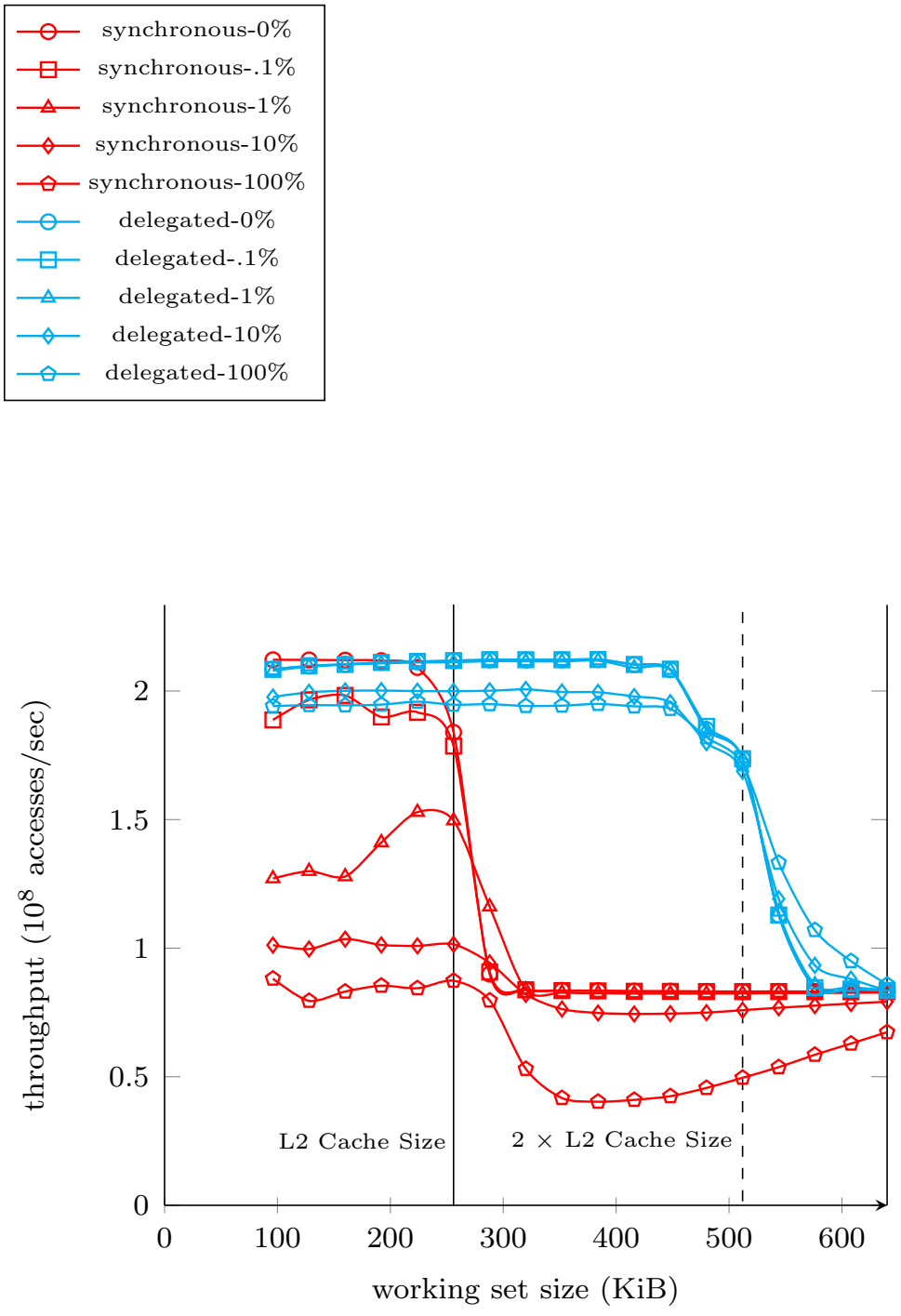


Figure 5.5: Performance of Accesstest for working set sizes around that of L2 cache and for varying update ratios.

with update ratio 10% and 100% increase for working set sizes just larger than twice the size of the L2 cache. A cursory analysis suggests that they are caused by unexpected cooperation/competition for shared cache resources. A thorough analysis is required to identify the cause behind each of these behaviors with certainty and is left to future work.

L3 Cache

Figure 5.6 shows the performance of Accesstest for working set sizes at and around the size of the L3 cache. Note that, in this figure, results are shown for working set sizes just larger than the largest shown in Figure 5.5. It is interesting to note that both configurations have roughly the same performance no matter the update ratio.

When the update ratio is 0% and the working set size is smaller than that of the L3 cache, all memory accesses hit in the L3 cache and there is no cache coherence overheads from accessing the pointer chasing path as no cache lines are modified. This behavior is observed and discussed with respect to the L2 cache. However, when the update ratio is 0%, observed performance for working set sizes larger than the L3 cache differs greatly from that for working set sizes just larger than the L2 cache: the performance of both modes is the same. In synchronous mode, the working set is too large to fit into the L3 cache. Accesses to the working set typically miss the L3 cache because most data lines are evicted from L3 to make space for a more recently accessed data line. In delegated mode, thread migration results in each core accessing one disjoint half of the working set and each half is smaller than that of the L3 cache. However, the L3 cache is shared between the cores. As a result, delegated mode experiences the same L3 cache misses as synchronous mode and, therefore, has the same behavior and performance.

As the update ratio increases, the performances of corresponding synchronous modes and delegated modes are roughly the same. Moreover, all executions have roughly the same performance. This can be understood by realizing that each data line is accessed rather infrequently (only once for most data lines as access counts follow a Zipf distribution with shape parameter 1) during each iteration of the session loop. As a result, for a given data line, a core is likely to access the entire working set before accessing that data line again. Therefore, it is likely to have been evicted to the level of the memory hierarchy that is large enough to contain the entire working set: either L3 or main memory depending on the working set size. Put another way, modifying a cache line and subsequently invalidating copies in all other cores' caches stabilizes performance at a low level of performance (i.e., roughly four times slower than the other experiments as all data is drawn from main memory). It is likely that the cache line would have been evicted to make room for more recently accessed data before being accessed again.

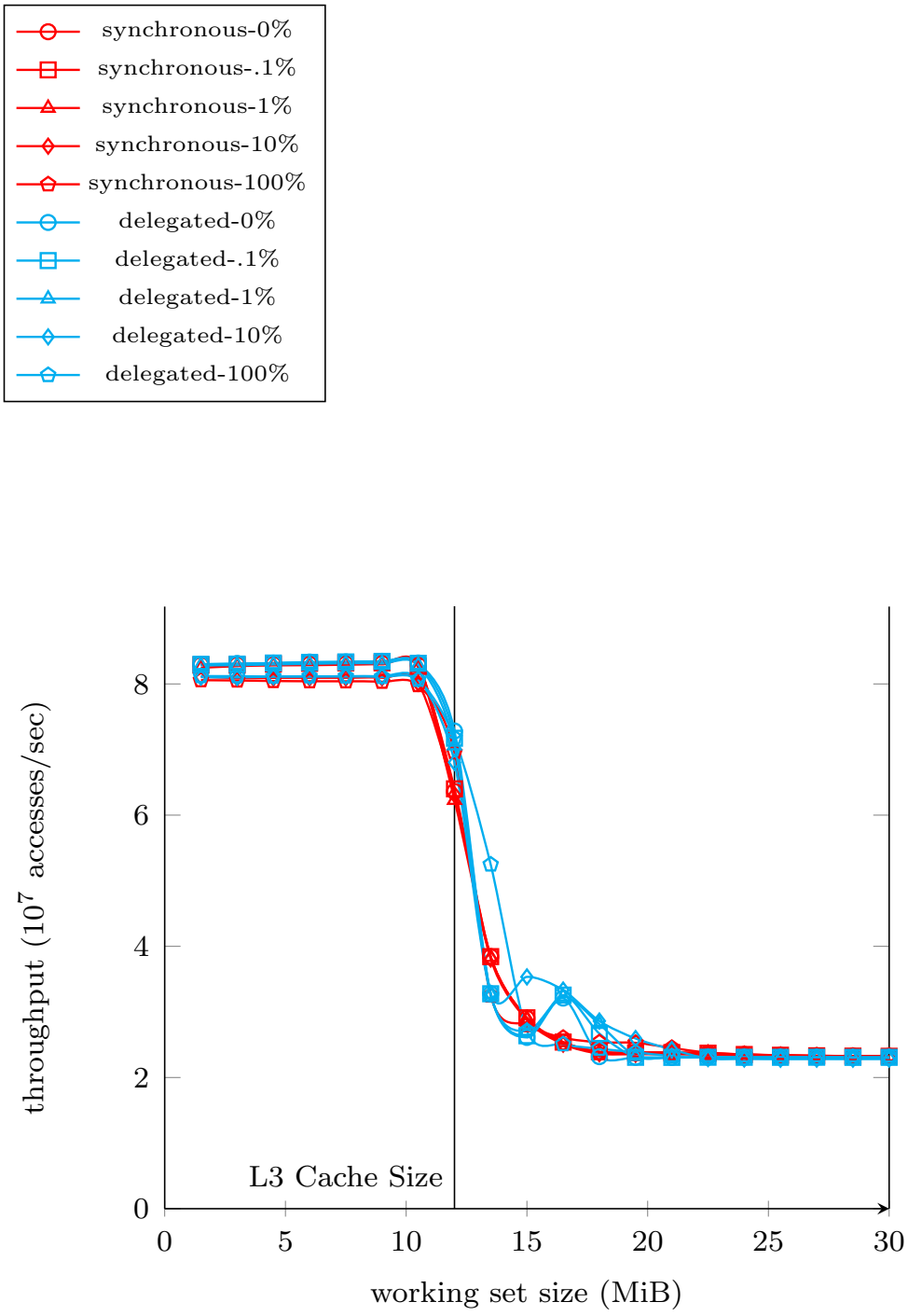


Figure 5.6: Performance of Accesstest for working set sizes around that of L3 cache and for varying update ratios.

Chapter 6

Prediction

6.1 Overview

This work develops an approach for predicting whether or not an application fitting the session-loop pattern would benefit from delegating the execution of a given phase of its session loop. At the core of this approach is the reframing of execution delegation as memory access delegation. Put another way, the impact of delegation on an application can be understood by studying accessed memory locations. This approach is applied to a real world application to form qualitative predictions, which are verified using actual performance results.

6.2 Approach

As stated previously, execution delegation works in situations where the cost of moving computation is cheaper than the cost of moving data. By assuming that user thread migration is used to perform delegation, these cost can be studied in detail. In particular, the aim is to determine if a phase is amenable to delegated execution via user thread migration. Given that an unmodified application pays some cost in performing data movement, this is equivalent to determining if the reimbursement from eliminating data movement is more than the cost of computation movement.

There are two data movement costs targeted by this work: cache coherence overheads and capacity misses. Cache coherence overheads stem from a set of multiple cores repeatedly reading and updating the same set of cache lines. By delegating the execution

that performs these accesses to a subset of cores, which share a higher level of the cache hierarchy, cache coherence overheads are reduced. If delegating to multiple cores, cache coherence overheads are reduced through a reduction in propagation delay. If one core is used, they are eliminated entirely since only one core will have the data in its cache. Capacity misses are caused by the working set being too large to be contain within the cache or one of its levels. By delegating execution, the data accessed by that execution is effectively removed from the working set of the cores from whom execution is delegated. Inversely, the data not accessed by that execution is effectively removed from the working set of the cores to whom execution is delegated. As is shown using the Accesstest microbenchmark in Section 5.3.2, splitting the application's working set in this way can better utilize the cores' private caches by reducing each core's working set.

Alternatively, a major cost of moving computation is the loss in data locality from forcing a thread to migrate between two cores with private caches. The data stored in the private cache of one core is abandoned when the thread migrates to the other core. Subsequent accesses to this data on the other core are likely to miss the cache and must be retrieved again.

A working set can be thought of as being composed of some number of unique data lines. As stated previously, a data line refers to the data that can be stored in a cache line such that one cache line can store any one of a multitude of data lines. For each phase, its working set is characterized along three dimensions: the degree to which the working set is accessed exclusively within the phase, the degree to which it is accessed by all sessions, and the rate at which different groups of data lines get modified.

In order to characterize a working set, each of its data lines is characterized as being either phase-local or phase-global and either session-local or session-global. Figure 6.1 depicts these characterizations visually. If a data line is accessed exclusively within a single phase of the session loop, the data line is called *phase-local*. Otherwise, the data line is accessed in multiple phases and is called *phase-global*. Returning to the webserver example, one phase of the session loop consists of receiving a request and another phase consists of processing the request and sending a reply. The data lines holding the request are likely phase-global, while the data lines containing the response are likely phase-local. Similarly, if a data line is accessed by a single session, the data line is called *session-local*. Otherwise, the data line is assumed to be accessed by all sessions and is called *session-global*. For example, in the webserver, the data lines holding the data structure representing a connection are likely session-local. Meanwhile, the data lines storing a database are likely session-global. The degree of a working set is then measured as the percentage of its data lines with respect to the number of all data lines accessed by the application.

Given the session loop split into two phases and a phase to delegate, a prediction is formulated using the following features. To estimate the reimbursement from delegation, phase-local/session-global and phase-local statistics are vital. The degree of the working set of the delegated phase that is phase-local/session-global and its update ratio is directly related to the reimbursement from reduced cache coherence overheads. The degree of the working set of the delegated phase that is phase-local is directly related to the reimbursement from reduced cache capacity misses. To estimate the costs of delegation, phase-global/session-local statistics are vital. The degree of the working set that is phase-global/session-local is directly related to the cost from the loss in data locality from migration.

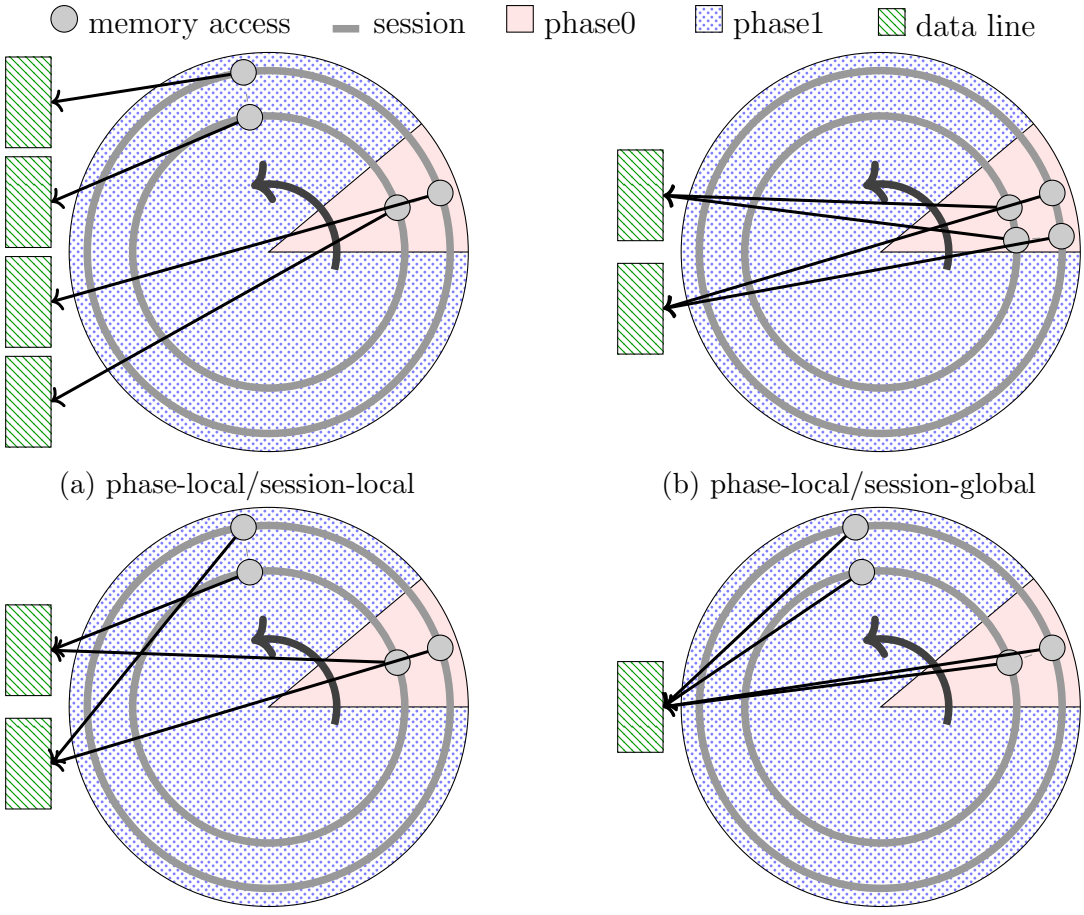
There are other costs to moving computation that are not accounted for with the proposed prediction approach. The work required to migrate threads and the cache pollution caused by migration are two examples that can be observed in the Accesstest microbenchmark results in Section 5.3.2. In the following experiments, these are assumed to be small and constant. Another cost is the loss in temporal locality from suspending a thread's execution for some extended duration of time. In applications to which the proposed prediction approach is applied, this cost is assumed to be negligible. Nevertheless, this assumption is verified while validating predictions produced by this approach in Section 6.5.

6.3 Design & Implementation

Prediction takes place in two stages: memory access sampling and working set characterization. From a sampling of memory accesses of the application, and given the session loop split into phases, a characterization of the phases is produced using a tool named *Accessprof*. From this characterization, a qualitative prediction can be made as to delegation's impact on performance. Further documentation can be found in Appendix C and in the code repository.

6.3.1 Memory Access Sampling

Before working sets of the application can be characterized, information about the data lines constituting those working sets must be collected. To collect this information, the Linux dynamic profiling tool Perf [5] is used. Perf is a powerful performance analysis tool for collecting information about both software and hardware events of an executing program or an entire system. In particular, it can be used to sample the memory accesses



(a) phase-local/session-local

(b) phase-local/session-global

(c) phase-global/session-local

(d) phase-global/session-global

Figure 6.1: Visualization of data line characterizations.

performed by an executing program and record contextual information about each sampled memory access.

The key reason why Perf is used in this work is because it is able to profile memory accesses performed when executing both user code and kernel code. Many applications fitting the session-loop pattern make heavy use of kernel services like IO. As a result, including accesses performed while executing kernel code in working set characterization is of vital importance for accurately predicting the qualitative impact of delegation. This is why Perf is used over other dynamic profiling tools such as Valgrind [33] or static analysis tools.

In order to characterize working sets from memory accesses, several pieces of contextual information about each access is required: the type of memory access performed (load or store), the memory location accessed, the session performing the access, and the phase in which the access is performed. On systems with a modern Intel processor and recent Linux kernel version, Perf is able to collect all of this information. (Perf may still be able to collect this information on systems with processors from other vendors. However, only Intel processors are used in this work.) Further information, including the specific Perf command line flags used, can be found in the code repository.

The memory access type is straightforward to collect. When Perf is invoked, the user must specify the event to be sampled. Events `mem_uops_retired.all_loads` and `mem_uops_retired.all_stores` are specified for sampling load and store micro-ops respectively.

The accessed memory address is captured and stored through the use of Processor Event-Based Sampling (PEBS), which is available on many modern Intel processors. Using PEBS, the address of the memory location access can be collected. While Perf allows for the collection of either the physical or virtual address of the memory location accessed, physical addresses are used for characterization in this study. This is done because a physical address uniquely identifies a data line. Meanwhile, a data line can have multiple virtual addresses. Further, a virtual address can refer to different data lines over the course of a program's execution. A data line can be assigned multiple virtual addresses (or even a virtual address that previously referred to a different data line) through repeated calls to the `mmap` and `munmap` system calls. This is an important problem because Apache, which is studied later in this thesis, makes heavy use of the `mmap` and `munmap` system calls. The above points make virtual addresses difficult to use with Accessprof as the tool assumes that each address uniquely identifies a data line.

The session performing the memory access can be inferred by identifying the user thread performing the access. It is assumed that each session uniquely corresponds to a

user thread. Further, the address range of each user thread's stack is never reused, in whole or part, as the stack of another user thread for the lifetime of the process being profiled. (The latter requirement is not natively satisfied by the user thread runtime. For all experiments presented, it is enforced.) When Perf samples a memory access, the session performing the access is identified by recording the user stack pointer at the time that the access occurs. By comparing the stack pointer with the address ranges of each user thread's stack, the session is identified. (The address ranges of the user thread stacks are recorded during session initialization.)

It is important to note that this association can be made when kernel code is executed even though the kernel has no knowledge of user threads. When a memory access is sampled while user code is being executed, the user stack pointer is retrieved from the stack pointer register. However, when kernel code is being executed, the executing kernel thread has a stack that is distinct from the running process and is protected in kernel space. As a result, the stack pointer register contains a stack pointer associated with this kernel space stack. However, part of the switch from executing user to kernel code involves storing the state of the registers at the time of the switch in the kernel stack. The user stack pointer can therefore be retrieved from the kernel stack. Using the appropriate command line flag, Perf collects this information.

Identifying the phase in which a memory access is performed is challenging. Perf can associate each sampled memory access with an instruction. However, a phase can be any sequence of sequentially executed instructions. As a result, two restrictions are placed on the phases that are examined. First, a phase is not examined if there exists another phase whose corresponding instruction sequence contains that of the prior. Second, the corresponding instruction sequence of at least one phase must be a function. This is referred to as the *function phase*. The other phase is typically referred to as the *implicit phase* as it consists of the part of the session loop that is not in the function phase. Weakening of these restrictions is left to future work.

Figure 6.2 shows an example session loop split into two phases. Phase 1 consists of a call to function `foo`. Phase 2 consists of a call to `doStuff1`, then `foo` again, and finally `doStuff2`. The second restriction is satisfied since phase 1 consists of a function: `foo`. Phase 1 is therefore the function phase. However, this split of the session loop into phases fails to satisfy the first restriction. The first restriction states that a phase cannot be examined if there exists another phase whose instruction sequence contains that of the prior. However, in this example, phase 2 contains the instruction sequence (`foo`) constituting phase 1. With these restrictions on the phases that can be studied, associating each sampled memory access with a phase entails checking whether or not the call chain at the time of the access contains the function constituting the function phase. Using the

```
while (true) {
    foo();          // phase 1

    doStuff1();     // \
    foo();          //  > phase 2
    doStuff2();     // /
}
```

Figure 6.2: Example phase that is not studied in this work.

appropriate command line flag, Perf collects this information.

6.3.2 Working Set Characterization

The sampled memory accesses of the application are fed into Accessprof. Given the function name with which the session loop is split into two phases, a working set characterization is produced for each of these phases. The characterization indicates the degree to which each working set is accessed exclusively within its phase, the degree to which it is accessed by all sessions, and the rate at which different groups of data lines get modified. This characterization is used to predict the qualitative impact of delegating the execution of a phase.

Working set characterization entails collecting a list of data lines accessed during the execution of the program. These data lines are classified as being either phase-local or phase-global and either session-local or session global using the sampled memory accesses.

A data line is phase-local if all of its associated sampled memory accesses originate exclusively from one phase of the session loop. Otherwise, the memory accesses originate from both phases and the data line is classified as phase-global. A data line is classified as session-local if its associated sampled memory accesses originate from only one session. Otherwise, the memory accesses originate from at least two sessions and all sessions are assumed to have accessed it. The data line is then classified as session-global.

Then, the degree to which a phase's working set is accessed, for example, exclusively within the phase and by a single session (i.e., phase-local/session-local) is computed as the number of all data lines accessed by the session that satisfying such requirements over the number of data lines accessed by the application. The update ratio of each group of data lines is computed using the associated sampled memory accesses. The update ratio is the percentage of operations that are store operations.

6.3.3 Characterization Accuracy

It is important to mention that this work does not present an approach for computing the margin of error with a given confidence level of the working set characterizations. In order to compute the margin of error of a characterization, the sample proportion of each category of a categorical distribution (i.e., phase-local/session-local, phase-local/session-global, etc.) must be computed. However, these proportions are determined through indirectly sampling from a different categorical distribution (i.e., memory accesses categorized by requesting session, current phase, and data line accessed). To further complicate things, Perf uses systematic sampling (sampling at a regular interval) to record events instead of simple random sampling. As a result, determining the accuracy of working set characterizations is left to future work.

6.4 Evaluation

The previously described approach for predicting whether or not an application would benefit from delegating the execution of a given phase of its session loop is evaluated on a real-world application. After splitting the session loop of the application into phases, Perf and Accessprof are used to characterize the working sets of the phases. This characterization is used to predict the qualitative impact of delegating the execution of a phase. This prediction is then verified with actual performance results. Several divisions of the session loop into phases are evaluated.

The application studied is Apache [3], which is a popular open-source HTTP web server. A survey found that Apache is used by 31.3% of all the websites whose web server is known at the time of writing [8]. Apache can be configured to employ the thread-per-session paradigm using kernel threads. However, to produce the results in this thesis, Apache is modified to instead use user threads and is made to fit the session-loop pattern. Each client starts a session with the Apache server by opening a connection. Over this connection, the client initiates any number of HTTP request/response interactions. The session loop therefore consists of receiving an HTTP request, processing the request, and sending an HTTP reply.

6.4.1 Configuration

Two machines are used to perform the following experiments: the server machine and client machine. They are connected via a dedicated 40 Gbps link. Each contains an Intel

Xeon Processor D-1540. The machines run Linux kernel version 5.11.0-34.

CPU frequency scaling is disabled on both machines. On only the server machine, hyperthreading and Linux's irqbalance daemon (enabled by default in most Linux distributions) are disabled. The latter is done as the irqbalance daemon has been found to affect the performance of networked applications running on multiple cores [23]. Each NIC queue is manually bound to a single CPU core (a job typically done dynamically by the irqbalance daemon) and each core is bound to an equal number of NIC queues.

To produce the following results, 100 clients repeatedly send requests for the same web page to an Apache server instance. The clients generate requests using 16 threads on the client machine. The Apache server instance services requests using four kernel threads pinned to four cores on the server machine.

6.4.2 Prediction

Figure 6.3 shows the working set characterization of Apache's session loop split into phases based on one of several functions. For each function, the session loop is split into the function phase with the remainder of the session loop constituting the implicit phase. While the prediction approach can be applied for any function, only system calls are studied in this work. This is done because previous work suggests that large performance gains can be attained through their delegation [41]. While Apache services requests, memory access samples are collected via Perf at a rate of one sample every 1,750 events for 15 seconds after a five second warm-up period. On average, approximately 9,000,000 samples are collected per run. Working set characterizations are generated for ten runs from which averages and 95% confidence intervals are computed.

Over all runs, the average total number of unique data lines accessed by Apache (and sampled by Perf) is 97,313.3 data lines with a 95% confidence interval of 881.06. The figure shows how much of each function's working set is phase-local/session-local, phase-local/session-global, phase-global/session-local, and phase-global/session-global as a percentage of Apache's working set. The figure further presents, for each classification group, the update ratio, which is the percentage of accesses to the corresponding data lines that were store operations. Aggregated phase-local and phase-global statistics are also shown.

From Figure 6.3, the `open` system call appears to be a good candidate for delegation. The phase-local/session-global working set percentage and update ratio are relatively high (with averages of 4.02% and 35.53% respectively). Additionally, the phase-global/session-local working set percentage and update ratio are relatively low (with averages of 0.61% and 0.39% respectively). When taken in aggregate, these results strongly suggest that

function	classification		% of app working set	update ratio
open	phase-local	session-local	3.20% ± 0.52%	63.59% ± 4.59%
		session-global	4.02% ± 0.25%	35.53% ± 1.47%
		both	7.21% ± 0.61%	47.63% ± 1.93%
	phase-global	session-local	0.61% ± 0.03%	0.39% ± 0.12%
		session-global	10.00% ± 0.42%	19.88% ± 0.31%
		both	10.61% ± 0.43%	18.77% ± 0.30%
read	phase-local	session-local	4.56% ± 0.13%	23.06% ± 1.13%
		session-global	1.63% ± 0.08%	17.74% ± 1.59%
		both	6.19% ± 0.20%	21.66% ± 1.16%
	phase-global	session-local	2.46% ± 0.04%	18.47% ± 0.70%
		session-global	6.43% ± 0.38%	23.47% ± 0.86%
		both	8.90% ± 0.36%	22.08% ± 0.71%
write	phase-local	session-local	1.49% ± 0.06%	34.44% ± 1.58%
		session-global	0.44% ± 0.03%	38.05% ± 1.33%
		both	1.93% ± 0.07%	35.24% ± 1.10%
	phase-global	session-local	2.34% ± 0.04%	8.89% ± 0.40%
		session-global	7.37% ± 0.20%	19.79% ± 0.45%
		both	9.71% ± 0.19%	17.16% ± 0.41%
mmap	phase-local	session-local	0.55% ± 0.07%	43.30% ± 6.58%
		session-global	0.23% ± 0.02%	12.28% ± 3.55%
		both	0.77% ± 0.07%	33.88% ± 4.33%
	phase-global	session-local	0.01% ± 0.00%	37.74% ± 12.84%
		session-global	4.95% ± 0.14%	13.63% ± 0.75%
		both	4.96% ± 0.13%	13.69% ± 0.72%

Figure 6.3: Working set characterization of Apache's session loop split into phases based on each of the specified system calls.

delegating the `open` system call could improve Apache's performance through a reduction in cache coherence overheads and possibly an overall reduction in cache coherence misses.

By delegating, the phase-local/session-global portion of the working set is accessed and modified exclusively by the cores to which execution is delegated instead of all cores. If these cores share a cache that is higher in the memory hierarchy than that of all cores, cache coherence overheads can be reduced through a reduction in propagation delay. If all cores delegate to only one core, then cache coherence misses are eliminated entirely for this data. Further, delegation produces only a small increase in cache coherence overheads through coherence misses as a result of phase-global/session-local data. By delegating, phase-global/session-local data is accessed by at least two cores (i.e., a core delegating execution and a core executing delegations) per session. Without delegation, only at least one core access this data.

Additionally, since the phase-local working set percentage is relatively large, it is possible that delegation could also improve performance through reduced capacity misses experienced by all cores. This is possible because, by delegating, cores executing the implicit phase no longer access–and therefore cache–the phase-local portion of the working set of `open`.

From these results, it is not clear if the `read` system call is a good candidate for delegation. This is because the phase-global/session-local working set percentage is larger than that of the phase-local/session-global portion of the working set. Further, their update ratios are approximately the same. This suggests that the reduction in cache coherence overheads from delegating is counteracted by the resulting increase. However, the relatively large phase-local working set percentage suggests a potential reduction in capacity misses as a result of delegation.

It is similarly unclear whether the `write` system call is a good candidate for delegation. The phase-global/session-local working set percentage is far larger than that of the phase-local/session-global portion of the working set. However, the phase-local/session-global update ratio is far larger than that of the phase-global/session-local portion of the working set. From these results, delegating `write` may introduce as much cache coherence overheads as it eliminates. Additionally, its phase-local working set percentage is relatively low, suggesting that a reduction in cache capacity misses should not be expected.

The `mmap` system call appears to be the least likely candidate for delegation. Its phase-local working set percentage is extremely low (with an average of 0.77%). As a result, a very limited improvement to cache coherence overheads or reduction to cache capacity misses should be expected. Further, its phase-global/session-local working set percentage is small enough to be negligible.

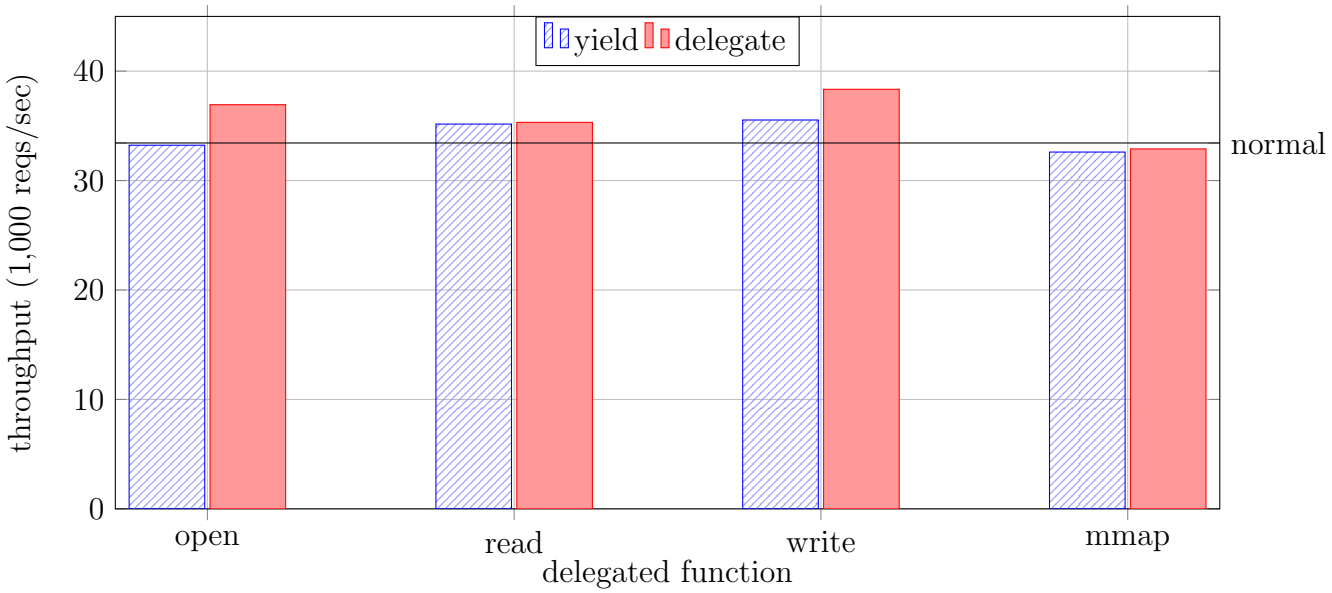


Figure 6.4: Performance of Apache when system call is delegated.

6.5 Validation

In order to validate the previous predictions, the studied functions are delegated and Apache's throughput is measured. The same configuration of Apache server and clients is used as in the previous section. However, one of Apache's four kernel threads (implying one of its four cores) is assigned to execute the delegated function. Further, Apache is modified to delegate the execution of the function being studied. From 40 runs of each configuration, the maximum coefficient of variation is less than 0.025. During each run, a five second warm-up interval is used during which no measurements are collected. After this interval, measurements are collected for 20 seconds. In Figure 6.4, the bars labeled **delegate** show Apache's throughput when the corresponding function is delegated. The horizontal line labeled **normal** indicates throughput when no delegation is performed.

When delegation is performed, the executing user thread is paused, migrated to another core, and resumed on this core at some later point. When the operation has finished executing, the same process happens in reverse. As a result, pausing and resuming execution of a user thread at some later point is likely to have some negative impact on a core's temporal locality. Further, this pause changes the flow of execution of the system as a whole. To better understand the performance impact of delegation, the impact of this pausing is isolated. Figure 6.4 also contains bars labeled **yield**. These bars show the

throughput of Apache when user threads yield execution (but do not migrate) before and after the corresponding function is executed.

As predicted from working set characterization, delegating the `open` system call yields a reasonable improvement in performance with an average increase of 10.48%. Additionally, the decrease in throughput caused by yielding execution before and after `open` is not statistically significant. Nevertheless, for all functions, any loss in performance caused by yielding can likely be attributed to a loss of temporal locality.

From the working set characterization, it was unclear how delegating the `read` system call would affect Apache's throughput. These validation results are in line with this prediction. Even though throughput increased by 5.66% on average, that increase can be attributed to the yielding of execution performed before and after delegation. The actual impact on performance of delegating `read` is statistically insignificant.

Yielding (but not migrating) before and after the `read` system call provides a surprisingly large improvement to Apache's throughput. Further investigation reveals that, after Apache sends a response back to a client, it soon after attempts a non-blocking `read` from the socket to see if another request has been sent from the client. If the client has sent another request, then the Apache server immediately handles it. However, if the client has not sent another request, then Apache has the thread `blocked` until there is data to be read from the socket (via `poll`). Since the clients are running in a closed loop (i.e., a client sends a request only after receiving a response for its previous request), Apache never finds a new request immediately after sending a response and the non-blocking `read` always returns nothing. Calling a non-blocking `read` that consistently fails every iteration of the session loop not only wastes cycles of the core but also pollutes the cache. By yielding before the `read`, the client has more time to receive the response and generate a new request. This increases the likelihood that the non-blocking `read` succeeds. Eliminating this failed non-blocking `read` from the session loop through yielding causes the observed improvement in performance. Yielding both before and after the `read` system call provides the same benefits as yielding exclusively before `read`.

Yielding before and after the `write` system call also provides a surprisingly large improvement to Apache's throughput (with an average increase of 6.30%). Further investigation reveals that the yield after the `write` has the same effect as the yield before the `read` system call. Yielding both before and after the `write` system call provides the same benefits as yielding exclusively after `write`.

Even while accounting for the increase in throughput from yielding after `write`, delegation of this operation results in a surprising increase in throughput (with an average increase of 14.70%). This was not predicted through working set characterization. Look-

ing more closely at the process of working set characterization helps to explain this result. When sampling memory accesses, several values are collected: physical address accessed, access type (read or write), stack pointer, and call chain. If any of these values are unspecified, the sample gets dropped. The predominant cause of samples getting dropped is an unspecified physical address accessed. Up to Linux kernel version 5.19, the physical address of memory allocated in the kernel via a call to `vmalloc` cannot be collected by Perf. On average, approximately 300,000 memory access samples are dropped per run because of this reason. Considering that the working set characterization for `write` is generated using approximately 400,000 samples on average, this suggests that the characterization for `write` is inaccurate. Nevertheless, from these validation results, the characterization for `write` can be expected to look like that of `open`: possibly a higher phase-local/session-global working set percentage and update ratio and a higher phase-local working set percentage. Enabling Perf to collect the physical address of memory allocated in the kernel via a call to `vmalloc` is left to future work as a result of time restrictions.

Lastly, delegating the `mmap` system call results in a negligible change in throughput as was predicted by the working set characterization. Further, the negative impact to temporal locality caused by yielding execution before and after `mmap` eliminates any potential performance improvement produced from delegation.

Chapter 7

Conclusion

This thesis presents a systematic study of execution delegation via user thread migration in applications fitting the session-loop pattern. The goal of this work is to develop tools and methods for predicting situations in which execution delegation via thread migration is beneficial to reduce or eliminate cache coherence overheads and cache capacity misses. Through these reductions, application performance can be improved. To this end, a microbenchmarking tool named Accesstest, a profiling tool named Accessprof, and a qualitative prediction approach have been developed.

Using Accesstest, execution delegation via user thread migration is studied where memory accesses are made to a variable sized working set and with a variable ratio of load and store operations. For all but the smallest working set sizes (smaller than the L1d cache), execution delegation resulted in either no change or an increase in performance of the microbenchmarking tool. Through reducing capacity misses, performance is increased by as much as 154%. Through reducing coherence misses, performance is increased by as much as 203%. Through reducing both, performance is increased by as much as 384%.

Using Linux's Perf and Accessprof, characterizations of the working sets of the phases of an application's session loop are produced. Through a detailed analysis of these characterizations, a qualitative prediction can be produced that indicates whether or not execution delegation would positively affect the application's performance. These characterization tools and the prediction approach are applied to the Apache HTTP server for four system calls. Of the four system calls, the prediction method accurately predicts execution delegation's impact on Apache's performance for three of the system calls. For the fourth, performance is not accurately predicted because of a limitation in Perf's profiling capabilities. Through delegation via user thread migration, Apache's performance is improved by

as much as 11% on average.

Beyond resolving this limitation, future work includes studying a wider range of functions (i.e., application functions, not just system calls). Further, larger systems with cores located on separate processor dies or even cores connected by a network must also be studied. This setting is vital as core counts continue to scale and systems become more distributed. Additionally, future work includes the study of another type of execution reorganization: temporal execution reorganization. This thesis studies specializing the cores of the system to execute certain regions of code in an ad hoc manner. A core is then left to idly wait or steal work during periods when it has been assigned no new work. Through execution batching, temporal locality and utilization of each core can be improved through intelligently scheduling when to perform assigned work and when to steal.

References

[1] Linux Scheduler - CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html. Accessed: 2022-08-26.

[2] JDK 1.1 for Solaris Developer's Guide - Chapter 2 multithreading. https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/index.html, 2010. Accessed: 2022-08-26.

[3] Apache HTTP Server Project. https://httpd.apache.org/, 2022. Accessed: 2022-08-28.

[4] *C∀ (Cforall) User Manual*, Aug 2022.

[5] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, Jul 2022. Accessed: 2022-08-26.

[6] runtime - Documentation. https://pkg.go.dev/runtime, Aug 2022. Accessed: 2022-08-26.

[7] TIOBE Index for September 2022. https://www.tiobe.com/tiobe-index/, Sept 2022. Accessed: 2022-09-05.

[8] Usage statistics of Apache. https://w3techs.com/technologies/details/ws-apache, Aug 2022. Accessed: 2022-08-28.

[9] Inas Abuqaddom, Sami Serhan, and Basel A. Mahafzah. Cache complexity of cache-oblivious approaches: A review and extension. volume 13, 2022. Copyright - © 2022. This work is licensed under https://creativecommons.org/licenses/by/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2022-07-01.

[10] AMD. *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*, Nov 2021.

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[12] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, Oct 2019. Accessed: 2022-08-26.

[13] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[14] Lubomir F Bic. Distributed computing using autonomous objects. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 160–168. IEEE, 1995.

[15] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[16] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the object-oriented language C++. *Software: Practice and Experience*, 22(2):137–172, 1992.

[17] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, Mar 1972.

[18] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. Highly scalable near memory processing with migrating threads on the emu system architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pages 2–9, 2016.

[19] Moritz Gmelin, Jochen Kreuzinger, Matthias Pfeffer, and Theo Ungerer. Agent-based distributed computing with jmessengers. In *International Workshop on Innovative Internet Community Systems*, pages 134–145. Springer, 2001.

[20] Adele Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984.

[21] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers, an imprint of Elsevier, 6 edition, 2019.

[23] Hoang, Huy. Building a framework for high-performance in-memory message-oriented middleware. Master's thesis, University of Waterloo, 2019.

[24] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Feb 2022.

[25] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, Apr 2022.

[26] Martin Karsten and Saman Barghi. User-level threading: Have your cake and eat it too. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–30, 2020.

[27] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, nov 1992.

[28] Peter M. Kogge and Brian A. Page. Locality: The 3rd wall and the need for innovation in parallel architectures. In *Architecture of Computing Systems: 34th International Conference, ARCS 2021, Virtual Event, June 7–8, 2021, Proceedings*, page 3–18, Berlin, Heidelberg, 2021. Springer-Verlag.

[29] P.M. Kogge. Of piglets and threadlets: Architectures for self-contained, mobile, memory programming. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 130–138, 2004.

[30] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating {Critical-Section} execution to improve the performance of multithreaded applications. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 65–76, 2012.

[31] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, page 162, New York, NY, USA, 2004. Association for Computing Machinery.

[32] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, Jun 2007.

[34] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[35] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), Nov 2015.

[36] Venkata K. Pingali, Sally A. McKee, Wilson C. Hsieh, and John B. Carter. Restructuring computations for temporal data cache locality. *Int. J. Parallel Program.*, 31(4):305–338, 2003.

[37] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[38] Stefan Reif, Phillip Raffeck, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Work in progress: Control-flow migration for data-locality optimisation in multi-core real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 371–374, 2020.

[39] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 342–358, 2017.

[40] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, Sep 1982.

[41] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[42] Paul L. Springer, Thomas Schibler, Géraud Krawezik, Jack Lightholder, and Peter M. Kogge. Machine learning algorithm performance on the lucata computer. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[43] Srinivasan, Priyaa Varshinee. Improving data locality in applications using message passing. Master's thesis, University of Waterloo, 2014.

[44] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *ACM SIGARCH Computer Architecture News*, 20(2):256–266, 1992.

APPENDICES

Appendix A

Quantitative Performance Prediction of Execution Delegation

This thesis presents an approach for predicting the qualitative (i.e., good or bad) impact of execution delegation on an application's performance. A logical next step would be to examine whether this approach can be used for predicting the *quantitative* impact (i.e., percentage speedup or slowdown) of execution delegation on an application's performance. In parallel with the experiments presented in this thesis, experiments were run in an attempt to answer this exact question.

The Accesstest microbenchmark can actually do more than that presented in this thesis. Given a characterization produced by Accessprof of an application whose session loop has been split into phases, Accesstest can take the characterization as input in order to simulate the application. (Specifying the flag `--log-phase-chars` to Accessprof results in a string containing the characterization being printed, which is directly consumable by Accesstest.) Additionally, Accessprof produces and Accesstest consumes a value called the access percentage for each group of data lines. This value indicates the percentage of accesses that are performed during the session loop to each group of data lines.

For example, Apache's performance with and without delegation of several operations is measured. Then, Accesstest's performance with and without delegation and using the characterizations produced by Accessprof is measured. However, Accesstest's performance results could not be used to predict the performance results of Apache. There are several possible reasons for this result that require further study.

A likely culprit is the limited number of memory access patterns that Accesstest can simulate. First, Accesstest assumes by default that data lines are accessed according to a

Zipf distribution. Second, assuming that cache lines are 64 bytes in size, each data line can only be accessed at most 16 times during a single iteration of the session loop. This restriction is enforced by the fact that each element of the pointer chasing path requires 4 bytes of space. (These four bytes store indicator bits, the next index of the element in the path, and other information.) Additionally, each element of the pointer chasing path can represent at most one memory access in the session loop. As a result, Accesstest is unable to represent memory access patterns where data lines are accessed tens, hundreds, or thousands of times per iteration of the session loop. For example, consider modeling two data lines accessed at a ratio of 17 to 1. Since a data line can only store up to sixteen elements, a data line can only be accessed up to sixteen times per iteration of the session loop. As a result, Accesstest can't model this access pattern or more extreme access patterns (e.g., 100 to 1). In reality, some session loops may have a data line that is accessed tens or hundreds of times per iteration. For example, consider how many times a thread might access a given location in its stack. Since the performance impact of execution delegation is so strongly influenced by cache usage behavior and memory access pattern dictates cache usage behavior, Accesstest's prediction power is low when used in such a way.

Another possible cause is the use of dynamically allocated memory. This work implicitly assumes that a session accesses roughly the same set of memory locations during each iteration of the session loop. Dynamic memory allocated each iteration of the session loop breaks this assumption. It is possible that, during each iteration, a session dynamically allocates memory whose corresponding data lines were never previously accessed. As a result, the working set appears larger to Accessprof than it actually is. Moreover, one session reusing memory that was previously dynamically allocated by another session can give Accessprof the false impression that there are more phase-global data lines than there actually are. This lowers the accuracy of Accessprof's characterization.

Appendix B

Accesstest Usage

```
Usage: ./accesstest [test_option...] phase_local_chars...
                    [phase_global_chars...] session_type ...
  test_option: one of the following
    -s --sessions=UINT             Number of sessions to traverse the pointer
                                     chasing path. (default=1)
    -d --duration=UINT             Duration to run in seconds (default=1)
    -r --seed=UINT                 Value used to seed random number generators
                                     (default=5489).
    --sess-distr={UINT|u}          Distribute session starting positions across
                                     phases. Specify phase index to have all
                                     sessions start on that phase or "u" to have
                                     them spread evenly across phases (default=0).
    --signal                       Start test only after SIGUSR1 is received.
                                     (disabled by default).
    --log-sessions=FILEPATH        Filepath to which session information gets
                                     saved. This information is consumed by
                                     profiler (disabled by default).
    -v --verbose                   Increase logging verbosity.
    -h --help                      Print help message and exit.

  phase_local_chars: UINT[,FLOAT[,FLOAT]]/UINT[,FLOAT[,FLOAT]]
    Specifies the local characteristics of the corresponding phase of execution.
    Consists of two parts separated by a slash (/). The first part consists of:
      - the number of session local data locations;
      - optionally, the update ratio of accesses made to the data locations
        (default=0.0); and
```

```
    - optionally, a positive scalar indicating the relative number of accesses
      made to these data locations (default=max accesses).
  The second part consists of the same fields but in reference to session
  global data locations.
  Note: the number of phase_local_chars specified indicates the number of
  phases.

phase_global_chars: UINT/UINT:UINT[,FLOAT,FLOAT[,FLOAT,FLOAT]]/
                    UINT[,FLOAT,FLOAT[,FLOAT,FLOAT]]/...
  Specifies the global characteristics of some set of phases of execution
  (>= 2). Consists of three parts with the first two separated by a slash (/)
  and the latter two a colon (:). The first two parts are the number of
  session local and session global data locations shared by the set of phases.
  The third part is a list of two or more tuples (separated by a slash (/)) of:
    - phase index;
    - optionally, update ratios of accesses made by this phase to the session
      local and session global data locations respectively (default=0.0); and
    - optionally, positive scalars indicating the relative number of accesses
      made by this phase to the session local and session global data
      locations respectively (default=max accesses).

session_type: one of the following
            thread              Represent each session with a Pthread.
             fibre              Represent each session with a Fibre.
```

Appendix C

Accessprof Usage

```
Usage: ./accessprof DATA_OPTIONS... [OPTIONS ...]
DATA_OPTIONS:
  -m --samples PATH              Specify path to samples.data file.
  -s --sessions PATH             Specify path to sessions.data file.
OPTIONS:
  -c --cacheline UINT            Specify system's cacheline size. (default=64)
  -p --samples-per-dataloc UINT  Minimum samples a data location needs to avoid
                                   being filtered out (default=1).
  --log-phase-chars              Log phase characteristics to pass to accesstest.
  --extended                     Log extended statistics.
  --filter REGEX                 Filter samples whose call chain contains
                                   function name matching (C++) regex
                                   (default=none).
  -v --verbose                   Increase logging verbosity.
  -h --help                      Print help message and exit.
```