Improving Data Locality in Applications using Message Passing

by

Priyaa Varshinee Srinivasan

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, 2014

Author's Declaration

Abstract

This thesis presents a systematic study of two modes of program execution: synchronous and asynchronous. In synchronous mode, program components are tightly coupled. Traditional procedure call represents the synchronous execution mode. In asynchronous mode, program components execute independently of each other. Asynchronous message passing represents the asynchronous execution mode. The asynchronous mode of execution introduces communication overhead in the execution of program components. However it improves the temporal locality of data in a program by facilitating temporal and spatial reorganization of program components. Temporal reorganization refers to the batched execution of program components. Spatial reorganization refers to the scheduling of components on different processors in order to avoid the over-subscription of cache memory. Synchronous execution avoids the communication overhead. The goal of this study is to systematically understand the trade-offs associated with each execution mode and the effect of each mode on the throughput and the resource utilization of applications. The findings of this study help derive application designs for achieving high throughput in current and future multicore hardware.

Acknowledgements

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

In particular, I thank Dr. Martin Karsten, my supervisor who with his consistent support and wisdom helped me to look deeper into issues that I would have otherwise ignored. He made this thesis possible as presented today.

I would like to specially thank Dr. Barry Sanders, the Director of Institute for Quantum Information Science, University of Calgary for his unrelenting support and encouragement which has helped me conquer the tough times and reach where I am today.

I would like to thank Dr. Peter Buhr for his insightful comments and suggestions during this research.

I thank my friends Krishnaveni and Ekta Jain for their support and memorable days we spent together in Waterloo.

Dedication

This thesis is dedicated to my Mother, Shanthi Vathani Srinivasan.

"You are one of the chief architects of this work"

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The architecture of computer hardware has undergone considerable change in the past five decades, which is evident in: 1) the presence of high speed processor structures that hide the gap between the speed of a processor and the latency of main memory and 2) the availability of more than one instruction execution unit embedded on a single processor chip. Since 1970, the CPU frequency has increased 50%-100% every year while the main memory latency has decreased at a rate of just 7% [14]. This growing disparity between the speed of a processor and the speed of main memory is known as the memory wall problem [38]. Cache memory plays an important role in masking the effects of the memory wall problem.

Cache memory was introduced by Wilkes in 1965 as a slave memory between main memory and processor to hold the information that has been most recently used by the processor [37]. Since then there have been numerous advancements in cache memory design. Current computers contain three levels of cache memory with the last level being shared among cores on the same processor chip. In current computers, cache memory is hundreds of times faster than main memory. Although the capacity of cache memory is minuscule compared to the capacity of main memory, cache memory proves to be an effective solution for the memory wall problem because of the temporal and the spatial locality [8] exhibited by the data and the code of a program.

Another innovation in computer hardware is the invention of Chip Multi Processors (CMP). Power requirements of a processor chip increases exponentially with the increase in CPU clock speed [16]. This effect has forced the design of processor chips with more than one CPU embedded to utilize the available transistors. Such chips are known as the Chip Multi Processors (CMPs) and each embedded CPU is called a processor *core*. The advent of CMPs enables concurrent programming where multiple tasks make progress in overlapping time periods utilizing the parallelism provided by the underlying hardware. Figure 1.1 represents the hardware model of current computers versus the hardware model of legacy machines.

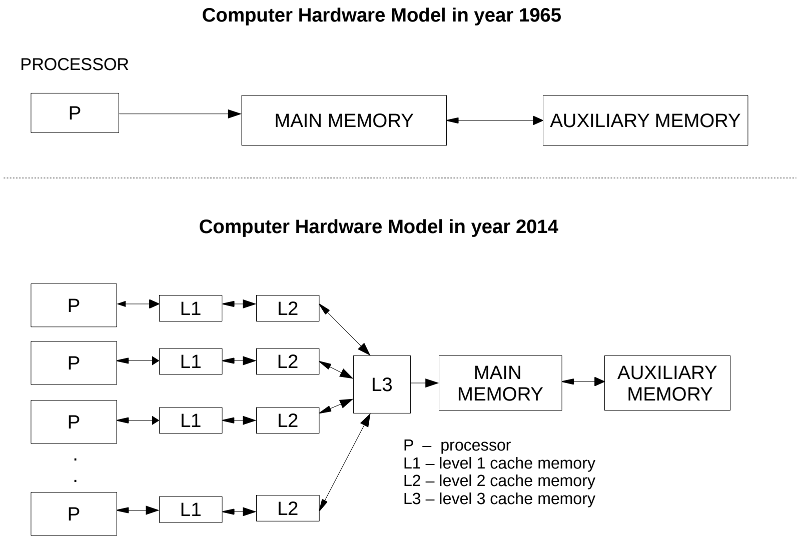

Computer Hardware Model in year 1965
PROCESSOR
P
MAIN MEMORY
AUXILIARY MEMORY
Computer Hardware Model in year 2014
P
L1
L2
P
L1
L2
L3
MAIN MEMORY
AUXILIARY MEMORY
P
L1
L2
P
L1
L2
P – processor
L1 – level 1 cache memory
L2 – level 2 cache memory
L3 – level 3 cache memory

Figure 1.1 Evolution in computer hardware.

Current computer hardware resources can be categorized as:

1. *Supervised resources*, the usage of which is arbitrated by the operating system. The size of these resources available for allocation changes over time. Examples of supervised resources include CPU time and main memory.

2. *Unsupervised resources*, the usage of which is transparent to the operating system and is controlled by hardware mechanisms. The usage of such resources is not supervised by the operating system. Examples of unsupervised resources include translation look-aside buffer(TLB), multilevel cache memory, instruction pipeline, hardware prefetcher [34] and branch predictor [27].

Maximization of the throughput and the utilization of a computer system is realized when the resources provided by the system are used efficiently by multiple tasks running in the system. Current operating systems use the working set model [8] to achieve optimal allocation of main memory space among multiple processes. However the problem of efficient usage of the unsupervised resources is often ignored by the design of both the current operating systems and applications.

The aim of this study is to understand the impact of program design on an application's usage of cache memory, which in turn affects the throughput and CPU utilization of the application. The design of a program is described based on two execution modes namely synchronous mode and asynchronous mode. In the synchronous mode, the execution of program components are tightly coupled. Procedure call represents the synchronous execution mode. In the asynchronous mode, the program components are loosely coupled using message passing, thus allowing the components to be executed independently of each other.

The synchronous execution mode potentially affects the temporal locality of data in a program by executing components sequentially. In order to preserve temporal locality, program components can be reorganized and executed on different cores using the asynchronous execution mode. However, the asynchronous mode introduces communication overhead in a program. This research investigates the trade-offs of each execution mode under various scenarios. A microbenchmark and a real-world application called Memcached [2] has been used for this purpose. The microbenchmark results are applied to improve the throughput of the Memcached server. The modified server design achieves up to 32% improvement in its throughput, conforming to the results of the microbenchmark.

The rest of the thesis is organized as follows. Chapter 2 elaborates on the modes of execution: their characteristics and the factors associated with each mode that can potentially affect the cache usage of a program. It also introduces the program model used to build the microbenchmark and defines necessary new terms used later in the thesis. The design and the implementation of the microbenchmark along with the experiment results are presented in Chapter 3. Memcached experiments and results are presented in Chapter 4. Chapter 5 presents the previous research relevant to this work. The thesis is concluded in Chapter 6.

Chapter 2

Execution Component Model

2.1 Program Model

Program design is an important factor determining the cache usage behaviour of a program. For the general treatment of the modes of execution and their impact on cache memory usage, *a program P is modelled as a collection of execution components (ECs) where each execution component represents a synchronous execution path of the program.*

$$P = \{S_1, S_2, S_3, ...S_n\} \text{ where } S_i \text{ is an execution component}$$

A loop is an example of an execution component. Similarly, a procedure call is an example of an execution component. This program model is referred to as the *Execution Component Model.* The terms component and execution component are used interchangeably for the rest of the thesis. Depending on the program design, a component can be executed either in the synchronous or in the asynchronous execution mode.

To understand the effect of the execution modes on the cache usage behaviour of a program, the metric *reuse distance* (also known as LRU stack distance) [10] is used. The reuse distance of a data element in a sequential program is a measure of the volume of data accessed between two successive references to the element. Reuse distance is approximated as the number of distinct cachelines accessed between two successive references of the same data element. Reuse distance is one of the widely used metric in analyzing the locality behaviour of a program.

2.2 Synchronous mode

Using the Execution Component Model, the modes of execution can be described as follows. A program's design is said to adopt the *synchronous mode of execution* when the design of the program results in tight temporal coupling of program components. An example is shown in Figure 2.1.

```
//thread
while(...){
     component A execution
     component B execution
}
```

Figure 2.1 Synchronous mode of execution.

The synchronous execution mode increases the reuse distance of a data element in a program by sequential execution of components. The increase in the reuse distance affects the temporal locality of data. Consider the pseudocode given in Figure 2.1. Let component A access data element d. The reuse distance of d depends on the size of the data accessed by both the components A and B. This distance affects the temporal locality of the data element d. The synchronous execution mode potentially leads to two cache effects namely the data displacement and the data invalidation effects, which bring down the throughput and increase the consumption of CPU resource by a program.

2.2.1 Data displacement effect

Consider a single thread executing the while loop given in Figure 2.1. When the combined data size of components A and B exceed the capacity of cache memory, the data of component A is replaced by the data of component B in the cache memory and vice versa during each iteration. This eviction leads to many cache misses in the program. The behaviour is known as the data displacement effect and it occurs when the reuse distance of a data element in a program exceeds the total number of cache lines available at a cache level. The data displacement effect refers to the over-subscription of cache memory.

2.2.2 Data invalidation effect

This effect is related to the current design of cache memory which maintains a coherent view of data across all the caches in the system. Consider a parallel application in which multiple threads execute the while loop given in Figure 2.1. When the data accessed by component A and component B are private to the threads, only the data displacement effect occurs. Now consider that the component B contains data shared among multiple threads. The modification of the shared data by threads running on multiple cores leads to the data invalidation effect. When a thread running on a core p_i modifies a shared data element, the value of the data stored in the cache memory of all the other cores $p_{(j \neq i)}$ is invalidated. This leads to costly cache misses in the application and the behaviour is known as the data invalidation effect.

The data invalidation effect potentially leads to *internal thrashing* of a program. *Internal thrashing of a program is the fall in the throughput of a program when the degree of parallelism in a program increases beyond a threshold.* Internal thrashing occurs whenever the increase in the number of threads increases the probability of cache misses in a program. With the data invalidation effect, an increase in the number of threads might increase the probability of shared data being modified leading to more invalidations. Internal thrashing due to the data invalidation effect is solved by restricting the execution of the component that modifies the shared data to a single core. One method of achieving this is to decouple the execution of a component that modifies shared data and execute that component on a single core in the asynchronous mode.

2.3 Asynchronous Mode

A program's design is said to follow the asynchronous mode of execution when the design of the program decouples the execution of two components temporally. For example, in Figure 2.2, component A and component B are decoupled by means of message passing. To preserve the sequential execution order of component A and component B as shown in Figure 2.1, message passing uses the send-receive-reply semantics [13], where the sender blocks until it receives a reply from the receiver. The reply is non-blocking.

In this study, in the asynchronous mode, the message passing semantics used between two communicating components are blocking buffered send, blocking receive and non-blocking reply. These semantics force sequential execution of communicating components explicitly. The message size is fixed. The communicating components work in the same address space. The message is always guaranteed to reach the receiver once sent.

```
//thread: loop A
while(...){
    component A execution
    send request to thread 2
    wait for result
}

//thread: loop B
while(...){
     receive request from thread 1
     component B execution
     return result
 }
```

Figure 2.2 Asynchronous mode of execution.

In the asynchronous mode of execution, program components run independently of each other and communicate via asynchronous message passing. Such a model of execution introduces communication overhead in the execution of a program. However, the asynchronous execution mode allows program components to be reorganized in time and space to improve the temporal locality of data.

2.3.1 Temporal reorganization of program components

Temporal reorganization means reordering the execution of components on a single core so that the operations with similar memory footprints are performed together. Consider an event-driven program that reads requests received from clients and performs a tree look-up operation for each request. The temporal locality of data in such a program can be improved by batching the requests that access the same subtrees and performing look-up operations for those requests together. Batching is achieved by means of queueing the requests. Batching of requests leads to the deferred execution of the look-up operations. However, it improves the temporal locality of data in the program.

Temporal reorganization is relevant to event-driven applications such as webservers and graphical user interfaces where the critical execution path is a loop that is also a pipeline. In such applications, the execution of the critical path is divided into multiple stages where the data flows from one stage to another. The asynchronous mode based on message passing facilitates batched and concurrent execution execution of stages.

2.3.2 Spatial reorganization of program components

Spatial reorganization refers to reorganizing the execution of program components across different cores to improve cache utilization. Consider the pseudocode given in Figure 2.2. In the spatial reorganization, component A and component B are scheduled to run on different cores to reduce the reuse distance of the data elements in both the components by using separate caches, which also avoids the data invalidation effect described in Section 2.2.2. Consider the scenario described in Section 2.2.2 which leads to the data invalidation effect. To avoid the effect, the execution of the components A and B is reorganized as shown in Figure 2.2. In Figure 2.2, multiple threads execute component A in parallel. However, component B which modifies the shared data is always executed by a single thread bound to a single core. Messages are passed between the threads executing component A and the thread executing component B modifies the shared data. Such reorganization avoids the data invalidation effect by restricting the execution of component B to a single core. Spatial reorganization is enabled by multicore hardware. This study focuses only on the effects of the spatial reorganization of program components.

It can be conjectured that the components in a loop are ideal candidates for spatial and temporal organization. In general, the components which share minimal or no data with other components, and those that are reused extensively in a program are the best candidates for reorganization.

2.3.3 Communication overhead

The asynchronous execution method introduces communication overhead in the execution of a program. This study examines the communication overhead of the asynchronous execution method. Message size is considered as one of the factors that can affect the communication overhead and hence, the throughput of a program.

2.4 Process Structure

This study uses the problem of mutual exclusion to investigate the characteristics of the synchronous and the asynchronous modes discussed in Sections 2.3 and 2.2. To solve the mutual exclusion problem, the synchronous mode adopts a procedure-oriented approach that uses mutual exclusion primitives such as semaphores and locks. The asynchronous mode adopts the proprietor paradigm based on message passing [13] to solve the problem. In the proprietor paradigm, a proprietor is the sole owner of a shared resource, which in this case is a memory location. Other processes can perform operations on the shared resource only by sending messages to the proprietor requesting it to perform the operations on behalf of them. The proprietor handles requests in the order of their arrival and handles only one request at a time. It has to be noted that the proprietor paradigm allows no parallelism since the proprietor handles only one request at a time and a sender blocks until a reply is received. This is the same with the procedure-oriented approach for the mutual exclusion problem. Another process structure that can be adopted by the asynchronous mode is the administrator paradigm [13] which allows concurrent servicing of multiple requests. This paradigm is adopted by Soares et.al. for parallel execution of system calls in the asynchronous mode [29].

The characteristics of the execution modes are investigated using a microbenchmark. The focus is on the effects of spatial reorganization of components in the asynchronous mode. The design of the microbenchmark and the experiments are explained in detail in Chapter 3.

Chapter 3

Microbenchmark

The asynchronous execution method allows temporal and spatial reorganization of the execution of program components. This study focuses only on the effects of spatial reorganization. While it does not consider the advantages of batching, inherent to the asynchronous method, this model allows examination of the communication overhead of the asynchronous execution mode in a worst-case scenario.

For the purpose of this study, the microbenchmark contains a simple loop. The loop is composed of a critical section component and a non-critical section component. In the microbenchmark, the critical section component is considered as the candidate for spatial reorganization based on the guidelines mentioned in Section 2.3.

3.1 Overview

The microbenchmark is a multi-threaded parallel application. The core logic of the microbenchmark executes two execution components (ECs) namely *the thread-local EC* and *the critical section EC* consecutively in a loop. The rest of the thesis refers to this loop as the *core loop*. The thread-local EC scans a private array of size n_1. The critical section EC scans an array of size n_2 each time modifying m distinct array elements, where $0 \leq m < n_2 \leq n_1$. The size of each array element is equal to that of cacheline (64 bytes on the test machine). The size of the data that is referenced by each EC is called the *data size of the EC*.

The code for array read and array write is shown in Figure 3.1. The arrays are scanned using a pointer chasing method where the value of an element at index i is used as the

Array read:

```
int currIdx=0;
while( buffer[currIdx] != 0) {
    currIdx = buffer[currIdx]; //pointer chasing
}
```

Array write:

```
void walk(int divisor ){
     while( buffer[currIdx] != 0) {
       currIdx = buffer[currIdx]; //pointer chasing
       counter++;
       if( (counter & divisor) == 0 ) { // divisor is power of 2
         buffer[currIdx+1]++; // cacheline modified
       }
    }
}
```

Figure 3.1 Microbenchmark code: array read and array write.

index of the next element that has to be read. At the end of the link, the last element refers to the first element thereby creating a circular chain of elements from random positions. Random positions are generated by shuffling a sequential array using the modern version of the FisherYates shuffle [11]. The buffer is a physically contiguous chunk of memory, the setup of which is explained in Section 3.3.3.

It has to be noted that in the microbenchmark, no data element is referenced more than once during a single execution of a component. All the referenced elements are distinct. With such a model, in the synchronous mode, if the combined data size of the thread-local EC and the critical section EC exceeds the capacity of cache memory, then a data element of one component might be replaced by a data element of another component in the cache memory before the element is reused. Hence, this data access model represents a worst-case access pattern for the synchronous mode. This model is also adopted by Pingali et al. [25] for their study on temporal reorganization of program components to improve the data locality in memory intensive programs.

3.2 Design

The core logic of the microbenchmark is executed in the synchronous as well as the asynchronous mode.

3.2.1 Synchronous Execution Mode

Figure 3.2 shows the microbenchmark design for the synchronous execution mode of ECs. The core logic is implemented as a single while loop composed of the thread-local EC and the critical section EC. Instances of the while loop are created and are executed in parallel in the microbenchmark. A lock is used to protect the critical section.

3.2.2 Asynchronous Execution Mode

The asynchronous execution mode is realized using message passing. In the asynchronous execution mode, the critical section EC is instantiated as a separate thread known as the *asynchronous server thread* or just the server thread. Inside the core loop, after the execution of the thread-local EC, a request is sent to the server thread requesting the critical section execution. The *asynchronous client thread*, which sent the request blocks until its request is serviced and a reply is received from the server. Figure 3.3 shows the microbenchmark design for the asynchronous execution mode. The figure shows 3 asynchronous client threads and a single server thread executing in parallel.

3.2.3 Asynchronous Communication

In the asynchronous execution mode, communication from clients to server is facilitated using a single *message queue*. To send a request to the server, a client adds a message at the end of the queue and blocks for results. The server dequeues the message asynchronously, services the request and unblocks the client. The server writes the results to a shared variable before unblocking the client. Hence, the communication uses blocking buffered send, blocking receive and non-blocking reply primitives for synchronization.

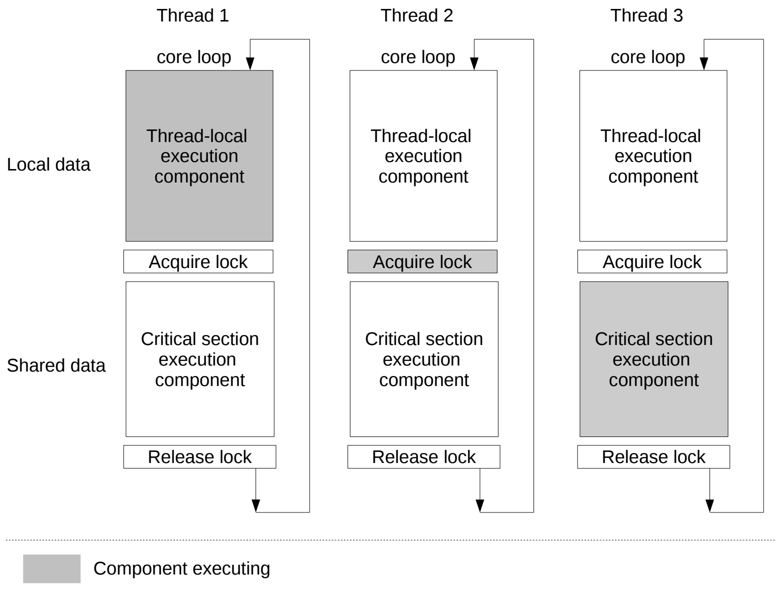

Thread 1
Thread 2
Thread 3
core loop
core loop
core loop
Local data
Thread-local execution component
Thread-local execution component
Thread-local execution component
Acquire lock
Acquire lock
Acquire lock
Shared data
Critical section execution component
Critical section execution component
Critical section execution component
Release lock
Release lock
Release lock
Component executing

Figure 3.2 Synchronous execution mode.

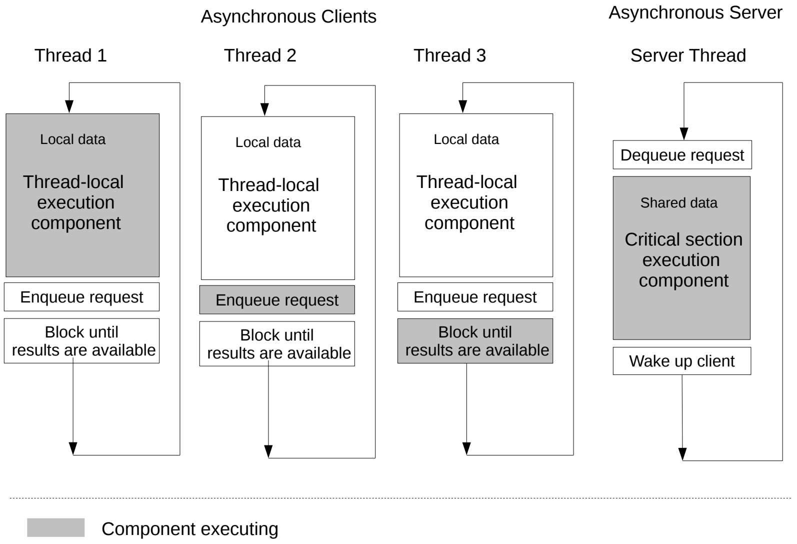

Asynchronous Clients
Asynchronous Server
Thread 1
Thread 2
Thread 3
Server Thread
Local data
Thread-local execution component
Local data
Thread-local execution component
Local data
Thread-local execution component
Dequeue request
Shared data
Critical section execution component
Enqueue request
Enqueue request
Enqueue request
Block until results are available
Block until results are available
Block until results are available
Wake up client
Component executing

Figure 3.3 Asynchronous execution mode.

3.3 Implementation

3.3.1 Message Queue

The message queue used in this study has been implemented as a circular ring buffer using an array. The enqueue and the dequeue operations are mutually exclusive and are protected by a lock. In all the microbenchmark experiments, the size of the queue is fixed to be greater than the number of threads. Hence, the message queue is never full since any client has only one outstanding request in the queue at any time t. The message queue is a multi-producer single-consumer queue. The threads are blocked when a queue-empty condition occurs.

The message queue is based on a *single-copy* design. In single-copy queues, a copy of the message is passed to the server. Other design choices are zero-copy and double-copy queue. In the zero-copy queue, a pointer to the message is passed from a client to the server. In the double-copy queue, for each send-receive operation, a message is copied twice. The message is copied from client to the queue during the send operation and from the queue to the server during the receive operation. Message queues are used for a wide range of applications like multiplayer online games, internet chat room, instant messaging services, etc. The design adopted by a message queue varies according to its purpose. In this study, a message queue is used for asynchronous communication between two components of a program. Here, the single-copy design has been adopted where a pointer to the message is stored in the queue during the send operation and the message is copied to the server during the receive operation.

3.3.2 Tests

An overview of the tests included in the microbenchmark are shown in Figure 3.4. The microbenchmark allows a user to study the data displacement and the data invalidation effects of the synchronous mode, and the communication overhead of asynchronous mode. The parameters of the tests are set using a configuration file.

Data Displacement Effect

Inside the core loop, the thread-local EC walks an array of n_1 elements and the critical section EC walks an array of n_2 elements. Data displacement occurs when the combined

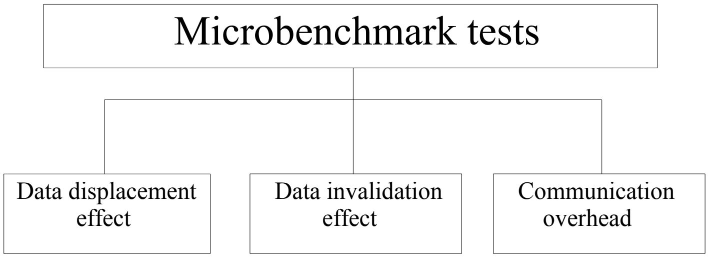


Figure 3.4 Microbenchmark experiments.

data size of the ECs exceeds the capacity of cache memory in the synchronous execution mode. The microbenchmark allows a user to set values for n_1 and n_2 at runtime.

Data Invalidation Effect

The data invalidation effect is simulated by modifying m out of the n_2 cachelines referenced by the critical section EC in the synchronous execution mode causing increasing amounts of cache coherence traffic. The thread-local EC walks an array of n_1 elements as before. The microbenchmark allows a user to set the value of m at runtime.

Communication Overhead

In the single-copy design, the message size affects the communication overhead because of the copy operation. To investigate the effect of the message size on the communication overhead, the size s of the message passed between an asynchronous client and the server is increased progressively in the experiments. Similar to the data displacement test, the ECs scan different arrays without modifying any data. The microbenchmark allows a user to set the value of s at runtime.

3.3.3 Elimination of the System Effects

This section explains the hardware and system software mechanisms that might affect the results of the experiments. It also explains how these effects are mitigated.

The microbenchmark is implemented as a kernel application in a simple, modular kernel called KOS [1]. It is run as a kernel application to avoid the direct and the indirect costs of mode switching [29] during system calls and interrupts. A simple modular kernel has been chosen for better understanding and fine-grained control over the test environment.

Eliminating operating system scheduler effects

In general, a CPU core is halted when an operating system scheduler does not find any task to run. The CPU halt and wake up operations have latencies associated with them. The execution time of a program is dominated by the overhead caused by the CPU halt and wake up operations when the latency of work done between each halt-wake up operation is minuscule compared to the latency of a single halt-wake up operation. The microbenchmark results are found to be affected by the overhead of halting and waking up the CPU for some of the experiments owing to the same reason. In KOS, allowing an idle thread to spin for three preemption intervals before halting the CPU solves the issue.

Eliminating data prefetching effect

Each EC of the core loop scans an array. The effects of data displacement and data invalidation is masked if the hardware prefetcher prefetches the array elements thereby avoiding all the cache misses that might occur otherwise. To avoid the effect of the prefetcher, a pointer chasing method is used to scan the array. The code in Figure 3.1 shows the array-walk based on pointer chasing.

Eliminating the effect of conflict misses

During the boot up stage of KOS, 32 MB of contiguous physical memory is requested by the microbenchmark from the kernel. The physically contiguous chunk is used for buffer allocation for array walks since a physically contiguous chunk achieves perfect cache coverage [33] and avoids conflict misses that might occur otherwise.

3.4 Results

3.4.1 Parameters, Metrics and Configuration

The parameters chosen for the microbenchmark experiments are listed in Table 3.1. The number of threads represents the degree of parallelism of the application. The data size of the ECs represents the memory demand of the application. The size of the message sent from an asynchronous client to the server represents the communication overhead of the asynchronous execution mode.

For each experiment, the throughput and the CPU utilization of the microbenchmark are measured. The throughput is measured using a local counter inside the core loop that is incremented during every iteration of the loop. The CPU utilization is measured using Time Stamp Counters (TSC). The counter is used to count the CPU cycles during each time interval a thread is active. The KOS scheduler notes the TSC counter value when a thread is scheduled to run. The scheduler notes the TSC counter value again when the thread is preempted or blocked. The difference between the noted values is calculated. Summation of all the differences gives the CPU cycles consumed by the thread during the experiment. Since the TSC counter is used to measure only the number of CPU cycles, variation in tick rate due to dynamic frequency scaling does not affect the results. During the experiments, each thread is bound to one of the cores. Hence, the TSC counter used for a thread to measure the CPU cycles is always the same. The metrics used are listed below.

1. *Throughput:* number of iterations of the core while loop
2. *CPU utilization:* CPU cycles consumed by a single iteration of the core while loop

Along with the above data, each experiment outputs the *server-busy percentage* as well. It is the percentage of requests for which the asynchronous server thread found the queue non-empty. This metric is measured by using a counter in the message queue that is incremented every time the queue is found empty during a dequeue operation. The metric is used to understand the behaviour of the microbenchmark in the asynchronous mode.

The configuration of the machine on which the experiments are run is shown in Table 3.2. Each experiment is run for 30 seconds and is repeated 20 times. The results represent the average of 20 runs. Each core is used by only one of the threads running in the application. For all the experiments, the number of threads in the asynchronous mode refers to the total number of threads including the asynchronous server thread.

Parameter
Number of threads
Data size of the critical section EC (in cachelines)
Data size of the thread-local EC (in cachelines)
Number of cachelines modified inside the critical section
Size of the message sent from an asynchronous client to the server

Table 3.1 Microbenchmark parameters.

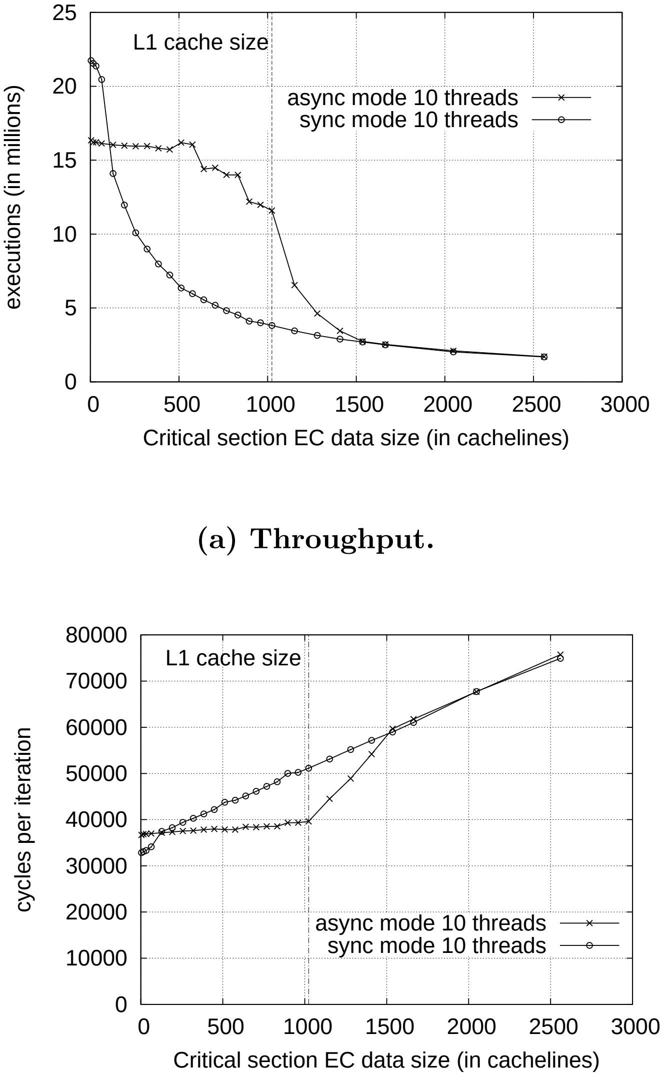
Model	HP Proliant DL585 G6
Processor	AMD Opteron 8431
Processor speed	2.4G
Number of sockets	4
Number of cores	24
L1 instruction cache	64K (per core)
L1 data cache	64K (per core)
L2 cache	512K (per core)
L3 cache	6144K
Cacheline size	64 bytes

Table 3.2 Microbenchmark experiment: Hardware configuration.

3.4.2 Data Displacement Experiment

In this experiment, the data size of the thread-local component is fixed to be 2048 cache lines and the data size of the critical section component is gradually increased. The number of threads in this experiment is fixed at 10 because the throughput of the microbenchmark under the synchronous execution mode achieves its peak around this point for different data sizes of the thread-local and the critical section EC. The results of the experiment are shown in Figures 3.5a and 3.5b.

At the beginning of Figure 3.5a, when the data size of the critical section EC is very low (< 64 cache lines), the throughput of the microbenchmark in the asynchronous execution mode is lower due to the communication overhead. However, as the data size of the critical section EC increases, the asynchronous mode achieves up to 3 times higher throughput by avoiding the data displacement effect. In the asynchronous mode, the data of the critical section EC is accessed from Level 1 cache whereas it is accessed from Level 2 memory in the synchronous mode. When the data size of the critical section EC eventually



(a) Throughput.

(b) CPU utilization.

Figure 3.5 Data displacement experiment: thread-local EC data size: 2048 cachelines.

exceeds Level 1 cache size, the throughput of the microbenchmark in the asynchronous mode falls drastically and becomes equal with that of the synchronous mode. This is clearly visible in the plot shown in Figure 3.5a, where the throughput falls drastically in the asynchronous mode after the number of cachelines accessed by the critical section EC exceeds 1024. The capacity of L1 cache in this experiment is 1024 cachelines. The sharp drop in throughput of the synchronous mode due to the data displacement effect is attributed to the worst case data access model adopted in the microbenchmark, where there is no data reuse during a single iteration of the ECs.

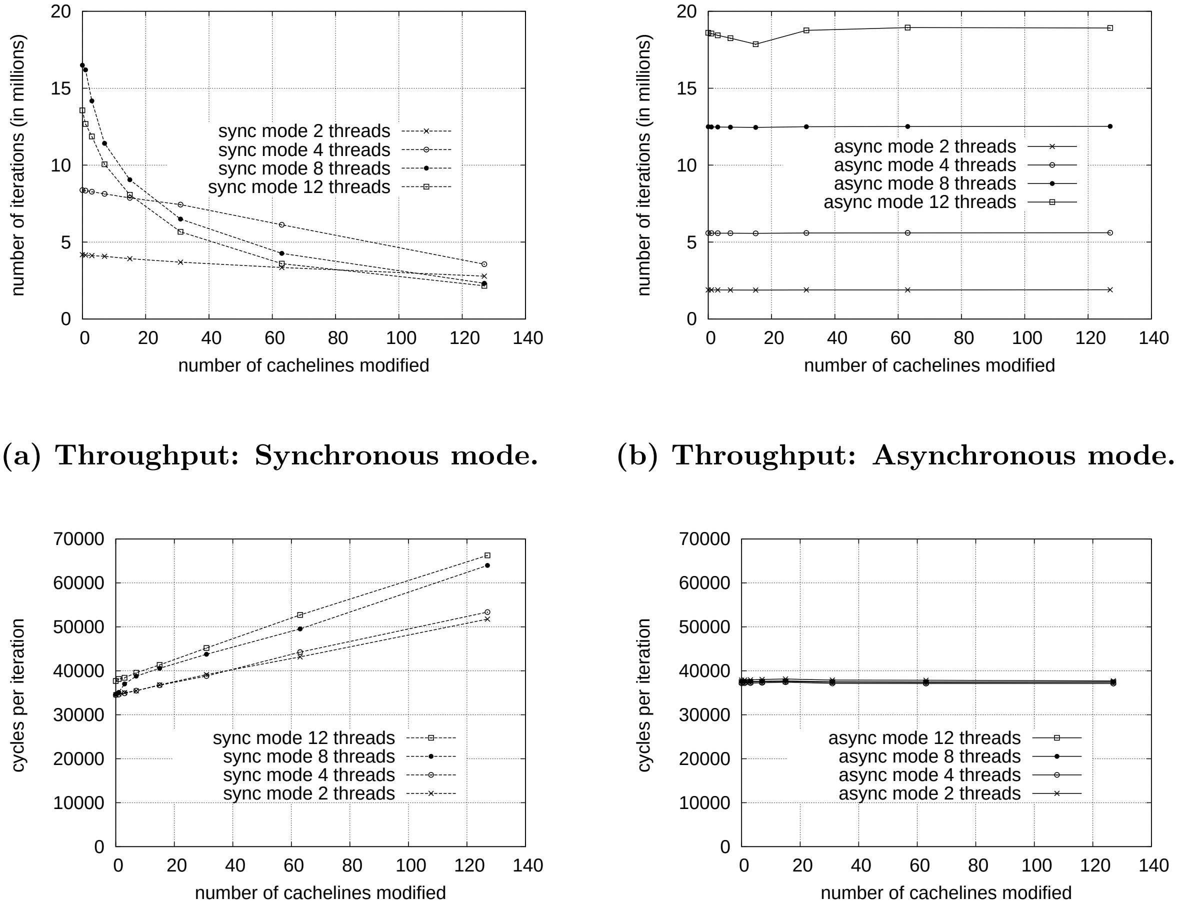
Figure 3.5b shows that the asynchronous method uses up to 23% less CPU cycles compared to the synchronous method when the size of the critical section data fits with the Level 1 cache. In the synchronous mode, the microbenchmark experiences cache misses inside the critical section due to the data displacement effect and accesses data from the Level 2 cache. This contributes to the increased number of cycles consumed in the synchronous mode. After the data size of the critical section EC exceeds Level 1 cache size, both the synchronous and the asynchronous modes consume almost equal number of CPU cycles.

3.4.3 Data Invalidation Experiment

For the data invalidation experiment, the number of cache lines modified inside the critical section component is gradually increased. The experiment is repeated for an increasing number of threads starting from 2 up to 12. The different number of threads represents different levels of contention of the critical section in the synchronous mode. The critical section contention is low when the number of threads is small and it increases as the number of threads increases. In the asynchronous mode, the server-busy percentage increases as the number of threads increases and reaches 100% for 12 threads. The results of the experiment do not change significantly beyond 12 threads. Figures 3.6a and 3.6b shows the throughput of the microbenchmark in the synchronous mode and in the asynchronous mode respectively.

The throughput of the microbenchmark as well as the CPU utilization remains almost constant for all the number of threads in the asynchronous mode. This is because the critical section data is modified by only one thread on a single core p_i during the experiment. This avoids the cacheline invalidations that occur by modifying the shared data on multiple cores. The throughput increases as the server-busy percentage increases and reaches the peak value when the server-busy percentage is close to 100 at 12 threads.

In the synchronous mode, the throughput decreases as the number of cachelines



(a) Throughput: Synchronous mode. **(b) Throughput: Asynchronous mode.**

(c) CPU utilization: Synchronous mode **(d) CPU utilization: Asynchronous mode**

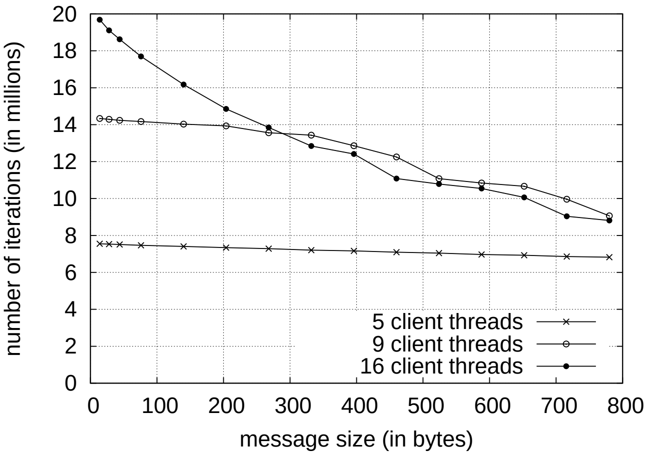
Figure 3.6 Data invalidation experiment: thread-local EC data size: 2048 cachelines; critical-section EC data size:128 cachelines.

modified increases, which is shown in Figure 3.6a. As the number of cachelines modified increases, the cache misses inside the critical section increases which in turn increases the execution time of the critical section execution. The execution time of the critical section is referred to as the critical section execution latency. This increase in the latency reduces the throughput of the microbenchmark.

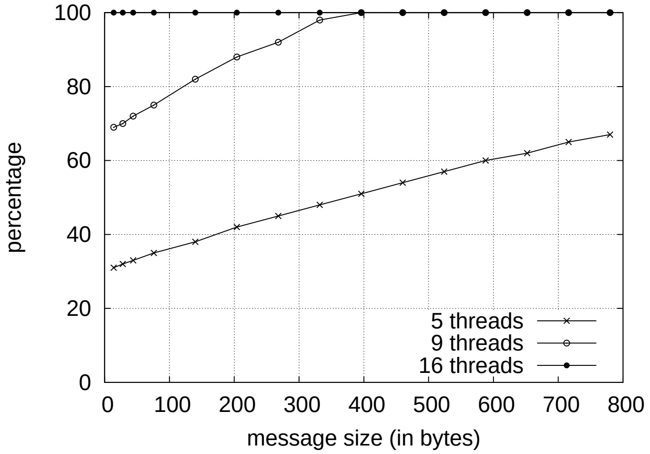
In the synchronous mode, the throughput decreases linearly when the number of threads is small. This is evident in the curve for 2 threads and 4 threads in Figure 3.6a. For smaller number of threads, the critical section contention is low and the threads mostly enter the critical section without having to wait for the entry. In such cases, the increase in the latency of critical section execution results in the linear decrease in the throughput of the microbenchmark.

The throughput of the microbenchmark in the synchronous mode falls exponentially for larger number of threads which is shown in the curve for 8 threads and 12 threads in Figure 3.6a. This is because, the increase in the number of threads results in the increase of the critical section contention as well the number of cache misses inside the critical section. When the critical section is contended, the threads wait to gain entry into the critical section. In such cases, any increase in the critical section execution latency also increases the waiting time of the threads, which in turn increases the contention of the critical section further. Hence, the overhead of executing the critical section increases with the number of cachelines modified. This increase in the overhead compounded by the increase in the critical section execution latency results in the exponential decrease of the throughput. It is also to be observed that, in the synchronous mode, the throughput of the microbenchmark diminishes as the number of threads increases. This decrease is shown in Figure 3.6a where the throughput of the microbenchmark is higher for 4 threads than for 8 threads and the throughput of 8 threads is higher than 12 threads, when the number of cachelines modified increases beyond 32. This indicates the *internal thrashing* effect explained in Section 2.2.

In the synchronous mode, as shown in Figure 3.6c, the decrease in the throughput is accompanied by a linear increase in the number of CPU cycles consumed per iteration of the core while loop. The increase in the CPU cycles is attributed to the increase in the cache misses that increases as the number of cachelines modified increases. The microbenchmark performs better in the asynchronous mode than in the synchronous mode, by achieving higher throughput while consuming lesser number of CPU cycles.



(a) Throughput.



(b) Server-busy percentage.

Figure 3.7 Message size experiment: thread-local EC data size: 2048 cachelines; critical-section EC data size:128 cacheline.

3.4.4 Message size experiment

In this experiment, the effect of the message size on the communication overhead is studied. This experiment is relevant only to the asynchronous execution mode. The datasize of the thread-local EC and the critical section EC is fixed to be 2048 cachelines and 128 cachelines respectively. There are no data modifications within the critical section. The data size of the critical section EC is fixed to be 1 cacheline so that the message passing latency is greater than the critical section execution latency in the microbenchmark. In this configuration, the throughput of the microbenchmark is influenced more by the message passing latency. The number of threads are fixed at 5, 9 and 16 which represent low, medium and high server-busy percentages respectively, when the message size is low (< 128 bytes).

The plot in Figure 3.7a shows the throughput of the microbenchmark for a set of different message sizes and different number of threads in the microbenchmark. Figure 3.7a shows that the increase in the message size reduces the microbenchmark throughput. In the single-copy design, the increase in the message size increases the message passing latency.

Figure 3.7b shows the server-busy percentage. When the server-busy percentage is less than 100, the increase in the message passing latency is compensated by the decrease in the idle time of the server which is evident in the 5 threads curve in Figure 3.7b. Hence, in such cases the throughput does not decrease significantly.

When the server is saturated, the throughput of the microbenchmark falls significantly with the increase in the message size. This effect is visible in the curve for 8 threads in Figure 3.7a, where the throughput fall is steep after the server-busy percentage is close to 100. The reason for the steep fall is as follows. When the server-busy percentage is close to 100, the increase in the message passing latency of a single request increases the latency of all the other requests that are pending in the queue. This increases the communication overhead in the microbenchmark and leads to a significant fall in throughput. The increase in the number of asynchronous client threads after the server saturates increases the number of pending requests in the queue. This, in turn increases the communication overhead further. Hence, the throughput is lower for 16 threads than for 9 threads. Hence, it can be inferred that the increase in the message size adversely affects the peak throughput of an application.

3.4.5 Trade-offs of Synchronous vs Asynchronous Execution Modes

The microbenchmark experiments show that the synchronous execution of components is beneficial when the reuse distance d is lesser than the capacity of cache memory. For example, the implementation of the asynchronous execution mode for this microbenchmark study utilizes a lock to add and remove messages from the queue. The communication overhead introduced by the asynchronous execution method is tolerable when the data locality in an application is improved by the reorganization of program components. Hence, a good application design balances the communication overhead and the cache utilization to decide the execution mode that has to be adopted for the execution of a component.

Chapter 4

Real-world Application

The results of the microbenchmark experiments are applied in a real world application called the Memcached server. The results of the Memcached experiments are found be in agreement with the microbenchmark results.

For the experiments, the Memcached server is modified to execute a critical section EC in the asynchronous mode. The throughput of the modified version of the server is compared with that of the unmodified version for different values of the parameters chosen. The experiments are run on Linux since KOS does not supported all the needed software libraries yet.

4.1 Memcached Server

Memcached [2] is a real-world, distributed, in-memory, key-value storage server that uses a hash table for quick look up of data. Memcached is an event-driven application based on client-server architecture. The server maintains a key-value store that is populated and queried by clients.

This experiment uses two types of Memcached operations: GET and SET. The GET operation retrieves information from the key-value storage. The SET operation either updates the value of an existing key or adds a new key-value pair to the store.

The total memory available for the key-value store is specified during the start up of the Memcached server. The total memory available is called the software cache as opposed to the hardware cache memory.

4.1.1 Memcached Server Internals

The write operations (SET) to the hash table in a Memcached server application uses a global lock called the `cache_lock`. If the server's cache is full when adding a new key-value pair, a server thread looks for expired items for replacement before replacing the least recently used items in the cache. All these operations are encapsulated in a routine called `do_item_alloc()` which is protected by the `cache_lock`. The read operation (GET) to the hash table is protected by granular locks called `item_lock[]`. The `do_item_alloc()` routine also uses `item_lock[]`.

Memcached has an event-driven component that creates a number of worker threads to handle incoming requests. The number of worker threads can be set by the user. Each worker thread handles only a particular set of client connections.

4.2 Memcslap, a Memcached Benchmarking Tool

Memcslap [3] is a benchmarking tool for the Memcached server. It simulates Memcached clients and collects server statistics. The Memcslap tool provides a rich set of parameters for fine-grained control of an experiment. Relevant to this research are the percentage of GET and SET operations, the number of Memcached client threads, the duration of the experiment, and the number of concurrency levels per client. The concurrency level determines the number of requests a client can send consecutively before waiting to hear back the results from the server. Each concurrency level opens one or more socket connections to a Memcached server. The default number of socket connections is one. At the end of each experiment, the Memcslap tool outputs the throughput of the Memcached server represented by the number of requests serviced per second.

In this study, the Memcached clients and the server are run on the same computer. In such a set up, it is found that the number of Memcached client threads has to be set equal to the number of server threads to keep the server threads from starving for requests. Any concurrency level (per client thread) greater than 20 seems to have no effect on the number of requests sent per second by a client.

4.3 Memcached Modifications

Profiling the Memcached server using AMD CodeXL tool for 100% of SET operations has revealed that the routine `do_item_alloc()` called during the SET operation experiences a large number of cache misses as the number of threads increase. The Memcached server has been modified to execute this EC asynchronously. The `cache_lock` that protects the EC is left intact since the lock protects other critical sections in the application as well.

During the experiments, the message passing latency is found to be high which adversely affects the peak throughput of the Memcached server. The server experiences low throughput in the asynchronous mode inspite of the improved cache utilization. The increase in latency can be attributed to the following reasons.

1. The enqueue and the dequeue operations of the message queue are mutually exclusive and are protected using a lock. The threads, which try to access the queue, might be suspended and rescheduled later by the operating system if the lock is held by another thread.

2. The asynchronous server blocks itself when the queue is empty and is woken up by the client when a new message is added in the empty queue. The block and wake up operations make costly system calls that increase the latency [29] of the mechanism.

Hence few modifications are made to the asynchronous method to reduce the message passing latency. The modifications are as follows.

1. The message queue is changed to the zero-copy design to avoid the overhead of copying the message from a client to the server. In the zero-copy design, a message sent by an asynchronous client is not copied into the local buffer of the asynchronous server. Rather a pointer to the message structure is passed from a client to the server. This design does not represent a general use case, but is useful for the current problem since the client does not modify the message until the server completes its request. After sending a message, the client blocks until it receives results from the server. So, the data remains safe.

2. Asynchronous client threads spins for a short interval of time, polling the availability of results at regular intervals before blocking themselves. The client threads break from the busy loop as soon as result is available. The clients block themselves if the result is not available at the end of the busy loop interval. This technique is to avoid the costly system calls made during the block and wake up operations.

3. The asynchronous server, instead of dequeueing a single item from the message queue, dequeues all the available items in a single remove operation.

4.4 Configuration

The Memcached version used in this study is 1.4.20. The configuration of the computer used for the experiments is given in Table 4.1. The machine runs Linux kernel v3.2.0-35. The version of the gcc compiler is 4.9.0. The experiment configuration is given in Table 4.2. The experiment is repeated thrice for each set of parameter values and the average value of the throughput is calculated. Each time the experiment is run for five minutes. The asynchronous server is bound to a core where as the worker threads have no CPU affinity. The worker threads can run on any core in the machine. The size of a key is 64 bytes and the size of value is 1024 bytes which are the default values used by the Memcslap tool.

Model	Supermicro AS-2042G-6RF
Processor	AMD Interlagos 6274
Processor speed	2.2G
Number of sockets	4
Number of cores	64
L1 instruction cache	16K (per core)
L1 data cache	64K (per core)
L2 cache	2048K (per core)
L3 cache	6144K
Cacheline size	64 bytes
Operating system	Linux
Kernel version	v3.2.0-35

Table 4.1 Memcached experiment: System configuration.

Number of Memcached server process	1
Number of Memcached server threads	1 to 30
Number of Memcached client threads	same as the number of server threads
Memcached client concurrency	20 per client

Table 4.2 Memcached experiment: Parameter values.

4.4.1 Metrics

The metrics used in the experiments are the throughput of the Memcached server and the average critical section cycles.

The throughput of the Memcached server application is displayed in the output of the Memcslap tool. The throughput refers to the average number of SET and GET operations performed by the Memcached server in one second.

The metric, average critical section cycles is the average number of CPU cycles spent by the Memcached server inside the critical section of the `do_item_alloc` routine. It is measured using the TSC counter using the `RDTSC` instruction. It represents the cache misses incurred by the Memcached server inside the critical section. This is because the cycles spent inside the critical section increases as the number of cache misses inside the critical section increase. The average number of CPU cycles does not include the cycles spent to acquire and release the `cache_lock`. The value of the TSC counter is read after the lock is acquired and before the lock is released during each critical section execution. The difference in the counter values denotes the number of cycles spent inside the critical section. The average number CPU cycles for one critical section is calculated by dividing the total number of CPU cycles spent inside the critical section by the number of critical section executions.

In the asynchronous mode, the asynchronous server thread which executes the `do_item_alloc` routine is bound to a single core. Hence, the TSC counter used is always the same. However, in the synchronous mode, the worker threads that execute the critical section are not bound to any core. Hence, they use the TSC counter of the core on which they currently execute. The TSC counters in the machine used for the experiments are not always synchronized and there exists a difference between the values of two different TSC counters in the system. However, this difference does not affect the results of the experiments, because it is assumed that each worker thread executes the entire critical section on the same core. The assumption is validated by running the experiments with each worker thread bound to a core as well as with the worker threads not bound to any core. In both the cases, the curves obtained by plotting the average number of CPU cycles against the number of threads, shows the same pattern.

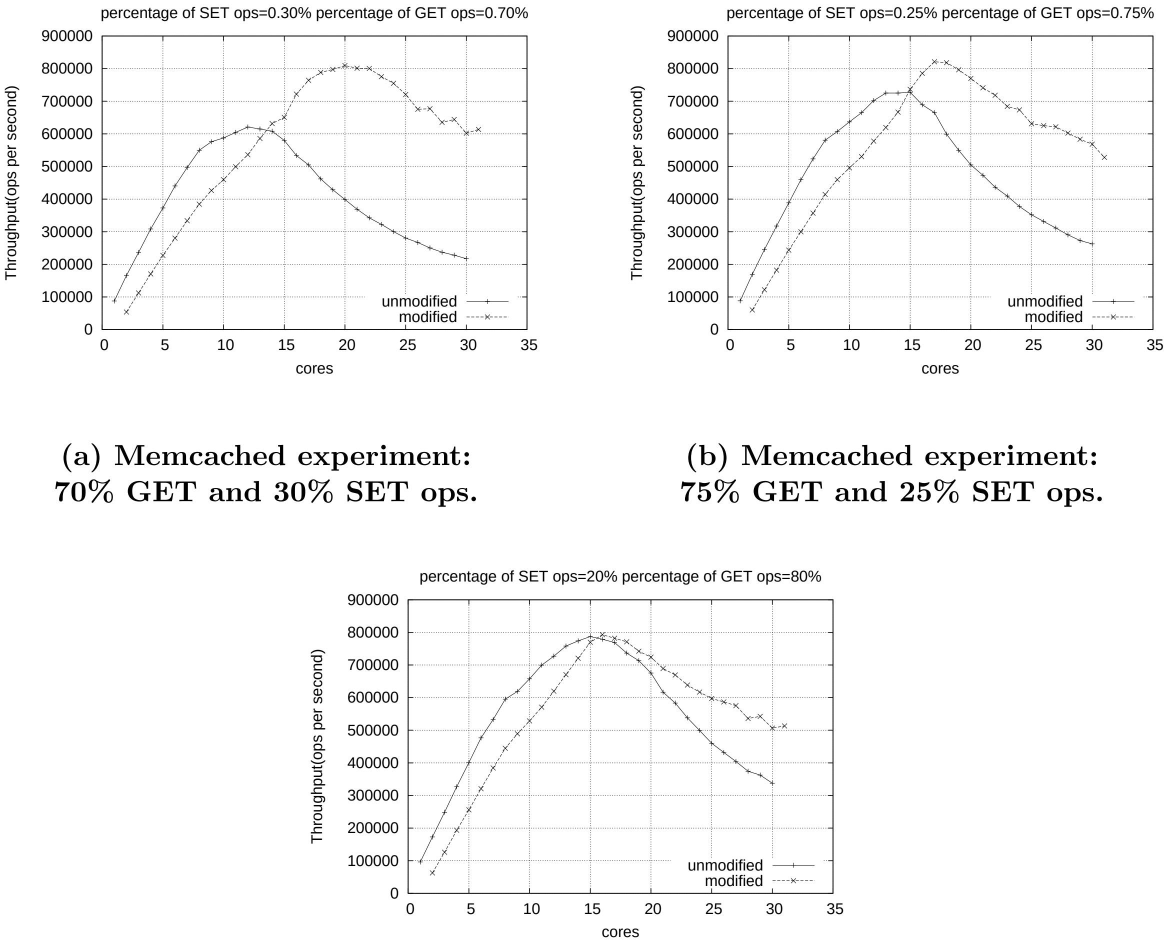
4.5 Results

Figures 4.1a, 4.1b and 4.1c shows the throughput of the Memcached server application for different percentages of GET and SET operations. The throughput of the modified version increases by 32%, 13% and 1% for 30%, 25% and 20% SET operations respectively. The throughput increase is only marginal for 20% of SET operations because the contention of the `cache_lock` reduces as the percentage of SET operations reduce.

The increase in the throughput of the modified version is attributed to the reduced number of cache misses inside the `do_item_alloc` routine. This is because in all the experiments, the average critical section CPU cycles remains almost constant in the asynchronous mode whereas it increases with the increase in the number of worker threads in the synchronous mode. The same effect is also observed in the microbenchmark during the data invalidation experiment explained in Section 3.4.3.

The effect of the internal thrashing is seen in the curves of both the modified and the unmodified version. The fall in the throughput of the unmodified version is steeper due to the data invalidation effect in the SET operations. Notice that the throughput of the unmodified version increases as the percentage of GET operations increase. However for the modified version, the throughput remains almost the same. This behaviour implies the latency of SET and GET operations are almost the same in the modified version.

Thus, the results of the Memcached experiment reinforces the microbenchmark results showing that preserving locality in a program by adopting both synchronous and asynchronous modes of execution improves its throughput.



(a) Memcached experiment: 70% GET and 30% SET ops.

(b) Memcached experiment: 75% GET and 25% SET ops.

(c) Memcached experiment: 80% GET and 20% SET ops.

Figure 4.1 Memcached experiment results.

Chapter 5

Related Work

5.1 On Locality of Data

To analyze the cache usage behaviour of applications, the concept of *LRU stack distance* [19] has been introduced by Mattson in 1970. Since then it has been used as a basic tool for locality behaviour and memory systems research. Ding et al. [10] redefine the LRU stack distance as *reuse distance* to analyze the behaviour of a program and to find reuse patterns consistent across all inputs. This research uses the concept of reuse distance to describe the effect of the execution modes on the temporal locality of data in an application.

Early methods of restructuring programs [12, 17, 20] have focused on reordering computations and data accesses in a program to promote data reuse. Compile time strategies for reordering include cache blocking techniques [17] which group computations onto data tiles to promote data reuse, loop transformation techniques [12, 20] and data layout transformations [5]. Runtime transformations for irregular applications, where the data reuse pattern of which is unknown at compile time, have also been studied [9, 21]. Ding et al. [9] use the inspector-executor strategy [7] to learn the data access patterns of a process, and use the knowledge for dynamic reorganization of computations and data. In [21], data and computations are reordered using various techniques such as space-filling curves [28].

With the advent of multicore processors, distributing computations and data across cores to improve locality has been considered. In affinity-based scheduling [30], data locality is preserved by scheduling a task on a core where the task's data has already been cached. Scheduling decisions are made based on the information on processor-cache affinity. Affinity scheduling adopts thread migrations to spread computations and avoid capacity misses.

Splitting a sequential program into a group of computations and reordering the computations with the aid of fine-grained threads has been considered by Philbin et al. [24] and Pingali et al. [25]. Fine-grained threads mean that the overhead of the thread primitives is low both in time and space. Philbin et al. split a sequential program into separate units of computations with no data dependency between them. Each unit of computation represents an independent fine-grained thread and the threads are scheduled to result in fewer cache misses in a program. To achieve this, the scheduling algorithm uses information about the address ranges that would be accessed by the threads. Pingali et al. consider a program as a series of logical operations, which are short streams of computations and the operations are reordered to improve the temporal locality of data in the program. Two transformations, namely early execution and deferred execution are used. These transformations batch the execution of one or more logical operations. The concepts presented in [24, 25] about splitting a program into multiple logical operations is similar to the Execution Component Model used in this research. The focus of these studies is temporal reorganization, where as the current study focuses on spatial reorganization. Also, in contrast to the current study, the methods used by Philbin and Pingali do not allow communication between the reordered components.

With the increase in the number of cores per CMP, specializing cores to offload certain tasks to improve locality has been considered [18, 29]. This approach involves communication between off-loaded tasks and other tasks in the system. For example, Remote Core Locking [18] adopts message passing to execute the critical sections in an application, on separate cores. In FlexSC [29], the execution of system calls are moved to separate cores using similar approach. This research has a different focus and uses the critical section problem to systematically study the effects of the execution modes on the temporal locality of data in programs by offloading critical section execution on to a separate core.

5.2 Accelerating Critical Sections

The microbenchmark used in this study represents applications that have one or more critical sections in their performance-critical execution paths. In the microbenchmark experiments, the temporal locality of the data in the critical section is improved using the asynchronous execution method which improves the throughput of the microbenchmark.

The problem of accelerating critical sections to improve the throughput of a program has been addressed in [18, 31, 32]. Suleman et al. [32] recommend asymmetric multicore computers to accelerate critical sections. In this method, the execution of a critical section is accelerated by executing it on a core with higher clock speed. Sridharan et al. [31] propose thread migrations to execute a critical section on the same core where it has been executed previously. This transfer is to improve the locality of the data in the critical section. In the Remote Core Locking (RCL) method [18], critical sections are executed on a dedicated core using a server thread. The threads in the application send messages to the server requesting the execution of critical section. Each thread uses a dedicated cache line to send messages to the server. The RCL method includes special optimizations to handle multiple critical sections. The methods used in RCL and in the current study are somewhat similar. However, this research differs from RCL and the other works by providing a systematic study of the execution modes.

5.3 Programming Paradigms

A programming paradigm [22] is a model of programming, the concepts of which drive the structure of a program written using that paradigm. A programming language provides constructs to implement a program using one or more programming paradigms.

Traditional computers with a single CPU execute programs in the synchronous mode inherently. The Von Neumann Architecture [35], which is the fundamental model of computers, has a single CPU that executes instructions sequentially. Hence, early programming paradigms such as procedural programming and functional programming have assumed a synchronous model of hardware and sequential execution of instructions. The applications developed using the languages based on such paradigms naturally adopt the synchronous execution mode.

The advent of multicore processors has led to the design of concurrent programming paradigms. In concurrent models of computation, tasks are allowed to make progress in overlapping time periods. Such models adopt asynchronous execution as their fundamental design characteristic to exploit the parallelism of the underlying hardware. This research shows that a good application design adopts both modes of execution to improve the temporal locality of data.

The compilers of the existing programming languages incorporate optimization techniques [12, 17, 20] and reorder computations to improve locality. This research tends to alter the control flow of an application by changing application design to improve locality. A flexible solution for this problem is to design programming languages that would enable compilers to optimize the control flow of a program by adopting appropriate execution modes.

To enable such optimization of the control flow, the programming languages need to express the data flow of a program along with the control flow. They have to support the features of both imperative programming, which expresses the control flow, and declarative programming, which expresses the logic without the control flow. The Jade Programming Language [26], which is an extension of C language, provides constructs to decompose computations into tasks and declare the data dependency between the tasks. The Jade compiler adopts sequential execution if there exists data dependency between tasks; otherwise it executes the tasks in parallel.

5.4 Software Systems based on Message passing

Since the advent of CMPs, the number of cores in desktop and server machines is increasing every year. Intel has introduced its 72 core x86 Knights Landing Chip at the end of 2013. Processor chips with thousands of cores are expected to be built in the future [4, 6]. Researchers consider message passing as the key to build software systems that exploit the parallelism that would be provided by future hardware [15, 23, 36]. In [15], Holland et al. propose lightweight channels and messages based programming model for future multicore computers. They observe that when the software systems running on super computers ran into scalability issues in 1990s, the solution has been to move from a shared-memory to a share-nothing model based on message passing.

Chapter 6

Conclusion

This thesis presents a systematic study of two modes of program execution namely synchronous and asynchronous modes. The goal of the study is to understand the trade-offs associated with each execution mode and the effect of the execution modes on a program's throughput and resource usage. The asynchronous mode allows spatial and temporal reorganization of program components. This study focuses only on the effects of spatial reorganization.

A microbenchmark has been designed for the purpose of this study. The microbenchmark results show that the communication overhead associated with the asynchronous execution mode is tolerable when the reorganization of components leads to improved temporal locality in a program. In such cases, a program achieves high throughput under the asynchronous mode. The synchronous mode is advantageous when the combined data size of program components that are executed synchronously fits within a cache level. The synchronous mode creates longer execution paths in a program by sequential execution of components. Hence, it increases the reuse distance of the data elements in a program. Data displacement and data invalidation are the two effects associated with the synchronous execution mode that affect the cache usage of a program adversely. It is concluded that an application design that balances the communication overhead and the cache utilization achieves high throughput in current multicore hardware. The findings of the microbenchmark experiments are applied to improve the throughput of a real-world distributed in-memory key-value storage server. The server application is modified to execute one of its program components in the asynchronous mode. The modified design achieves 32%, 13% and 1% higher throughput for 30%, 25% and 20% of SET operations respectively. The improvement in throughput is attributed to the improved temporal locality of data in the application.

This study focuses only on the effects of the spatial reorganization on the temporal locality of data in applications. A more general model would accommodate both spatial and temporal reorganization of components. Also, the general properties of the ECs that are potential candidates for reorganization is yet to be understood. The message queue used in this work is a circular ring buffer which is implemented using a fixed-size array. In real-world applications such as web servers, the number of requests is large at times and varies from time to time. For such applications, queues that grow and shrink dynamically are preferred over those that are statically sized. Investigation of the asynchronous mode with such queues is relevant for applications with highly varying workloads. Improving the usage of cache memory by means of application design is one approach. A more flexible solution would be to design programming languages, the constructs of which allow automatic identification and reorganization of program components to improve the data locality. All these issues are left for future work.

References

[1] KOS. https://cs.uwaterloo.ca/~mkarsten/kos.html. Accessed: 2014-18-11. 16

[2] Memcached. http://memcached.org. Accessed: 2014-18-11. 3, 26

[3] Memcslap. http://dev.man-online.org/man1/memcslap/. Accessed: 2014-18-11. 27

[4] Andreas Bechtolsheim. Memory Technologies for Data Intensive Computing. In *Proc. HTPS 2009*, October 2009. 36

[5] MichałCierniak and Wei Li. Unifying Data and Control Transformations for Distributed Shared-memory Machines. *SIGPLAN Not.*, 30(6):205–217, June 1995. 33

[6] Jack Clark. Intel: Why a 1,000-core chip is feasible. http://www.cnet.com/news/intel-why-a-1000-core-chip-is-feasible/. Accessed: 2014-18-11. 36

[7] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22:462–479, 1993. 33

[8] Peter J. Denning. The Working Set Model for Program Behavior. *Commun. ACM*, 11(5):323–333, May 1968. 1, 3

[9] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM. 33

[10] Chen Ding and Yutao Zhong. Predicting Whole-program Locality Through Reuse Distance Analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003. 4, 33

[11] Richard Durstenfeld. Algorithm 235: Random Permutation. *Commun. ACM*, 7(7):420–, July 1964. 11

[12] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587 – 616, 1988. 33, 35

[13] W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. 11(5):435–466, May 1981. 6, 9

[14] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. 1

[15] David A Holland and Margo I Seltzer. Multicore OSes: Looking Forward from 1991, er, 2011. In *Proc. HotOS*, volume 11, pages 33–33, 2011. 36

[16] Laszlo B Kish. ”End of Moore’s law: thermal (noise) death of integration in micro and nano electronics ”. *Physics Letters A*, 305(34):144 – 149, 2002. 1

[17] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM. 33, 35

[18] 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 *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. 34, 35

[19] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970. 33

[20] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996. 33, 35

[21] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the 13th International Conference on Supercomputing*, ICS ’99, pages 425–433, New York, NY, USA, 1999. ACM. 33

[22] John C. Mitchell. *Foundations of Programming Languages.* MIT Press, Cambridge, MA, USA, 1996. 35

[23] Simon Peter and Thomas Anderson. Arrakis: A Case for the End of the Empire. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 26–26, Berkeley, CA, USA, 2013. USENIX Association. 36

[24] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread Scheduling for Cache Locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 60–71, New York, NY, USA, 1996. ACM. 34

[25] Venkata K. Pingali, Sally A. Mckee, Wilson C. Hsieh, and John B. Carter. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31:2003, 2003. 11, 34

[26] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26:28–38, 1993. 36

[27] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design : Fundamentals of Superscalar Processors.* McGraw-Hill Higher Education, Boston, 2005. Index. 2

[28] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal Of Parallel and Distributed Computing*, 27:118–141, 1995. 33

[29] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. 9, 16, 28, 34

[30] M. S. Squiillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):131–143, February 1993. 33

[31] Srinivas Sridharan, Brett Keck, Richard Murphy, Surendar Ch, and Peter Kogge. Thread Migration to Improve Synchronization Performance. In *Workshop on Operating System Interference in High Performance Applications*, 2006. 34, 35

[32] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009. 34

[33] Alexander Szlavik. Cache-Aware Virtual Page Management. Master's thesis, David R Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, 2013. 16

[34] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000. 2

[35] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993. 35

[36] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009. 36

[37] M. V. Wilkes. Readings in Computer Architecture. chapter Slave Memories and Dynamic Storage Allocation, pages 371–372. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. 1

[38] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. 1