EVALUATING SHARED-CACHE PERFORMANCE WITH MICROBENCHMARKS AND REUSE DISTANCE ANALYSIS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Suman Vara

San Marcos, Texas
May 2011

EVALUATING SHARED-CACHE PERFORMANCE WITH MICROBENCHMARKS

AND REUSE DISTANCE ANALYSIS

Committee Members Approved:

____
Apan Qasem, Chair

____
Carol Hazlewood

____
Mark A McKenney

Approved:

____
J. Michael Willoughby
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor, Dr. Apan Qasem for his never ending encouragement and enthusiasm during this research. I am also grateful to Dr. Carol Hazelwood and Dr. Mark McKenney, for their support as my committee members and efforts in furthering my education.

Finally, I would like to dedicate this thesis to my loving parents for their support throughout my educational years.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The Multi-core Shift

As personal computers have become more prevalent and more applications have been designed for them, the end-user has seen the need for a faster, more capable system [19]. Speedup has been achieved by increasing clock speeds and more recently, adding multiple processing cores to the same chip [19]. Having multiple-cores on a single chip opens up opportunities for thread-level parallelism and dramatically increases the performance potential of applications running on these systems [18]. This is of enormous importance in the current power and heat limited environment because lower frequencies imply lower power consumption and less heat [18].

Sadly, however, the state-of-the-art in performance enhancing software is far from adequate in exploiting the hardware features of multicore architectures [18]. As a result, much of the performance capabilities of multi-core systems are yet to be explored [18]. In order to realize the full potential of CMP systems much of the responsibility to find and exploit opportunities for parallelism is now placed on software and programmers [18]. Programmers have to write applications with subroutines able to be run on different cores, meaning that data dependencies will have to be resolved or accounted for (e.g. latency in communication or using a shared cache)[19]. Also, Applications developed should be load-balanced, if one core is being used much more than another, the programmer is not taking full advantage of the multicore system [19]. Thus, the problem of finding and exploiting opportunities for parallelism in software is a difficult one and will require a great deal of effort if CMPs are to deliver at their performance and power capacities [18].

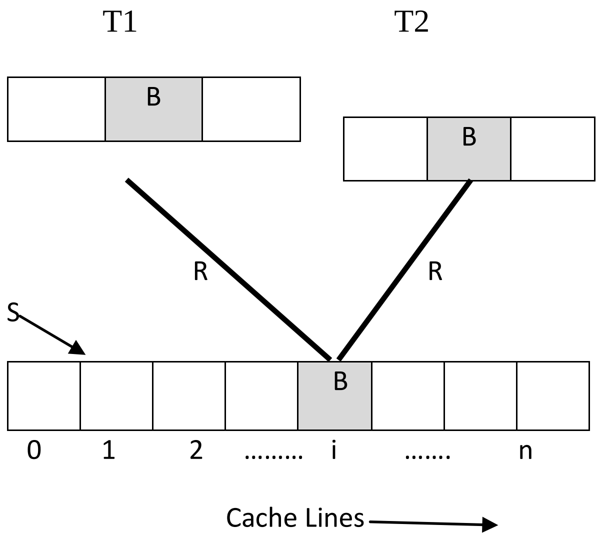


Figure 1(a): Read-Read

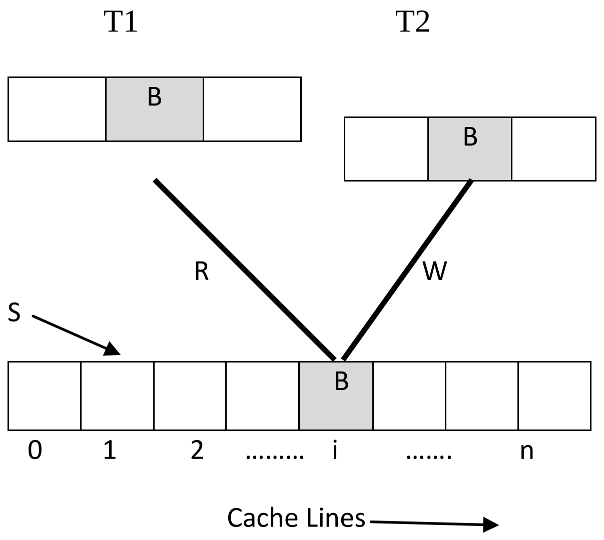


Figure 1(b):Read-Write

1.2 Shared Cache Problem

Further complicating the potential payoffs from the shift toward CMPs is the fact that at some level, memory resources are shared among different processing cores [18]. Sharing of cache resources can have both favorable and unfavorable impact on performance. Consider the scenario shown in Figure 1(a), where the cache is shared among threads T1 and T2. Let us suppose, T1 makes a read request for data element *B* , then the element will be retrieved from memory and brought into shared cache *S* and placed in some cache line *i* . After a while, if T2 makes a read request for same element *B*, then it can be retrieved from the same cache line *i*. Thus, this leads to reducing latency and thereby boosting performance. Contrary to above case, Let us consider read-write scenario among thread T1 and T2 as shown in Figure 1(b) .The problem that may occur with this scenario is when T1 is reading a value and at same time if T2 is writing a value to same location *i* in shared cache *S*, then T1 may not have the updated value. This leads to corruption of data, producing incorrect output. To avoid this data corruption, access by T1 needs to be serialized, leading to degraded performance. Lastly, let us consider a scenario where two threads T1 and T2 access different data elements A and B, both of which map to the cache location *i* in the shared cache line *S* as shown in Figure 2. This scenario may cause accesses by T1 to evict T2's data, or *vice–versa*. These conflict misses can lead to serious system performance loss because the system is spending a disproportionate amount of time if *thrashing* on the shared cache line. Because of this multitude of effects on performance, it is important to understand the behavior of shared cache on multicore architectures.

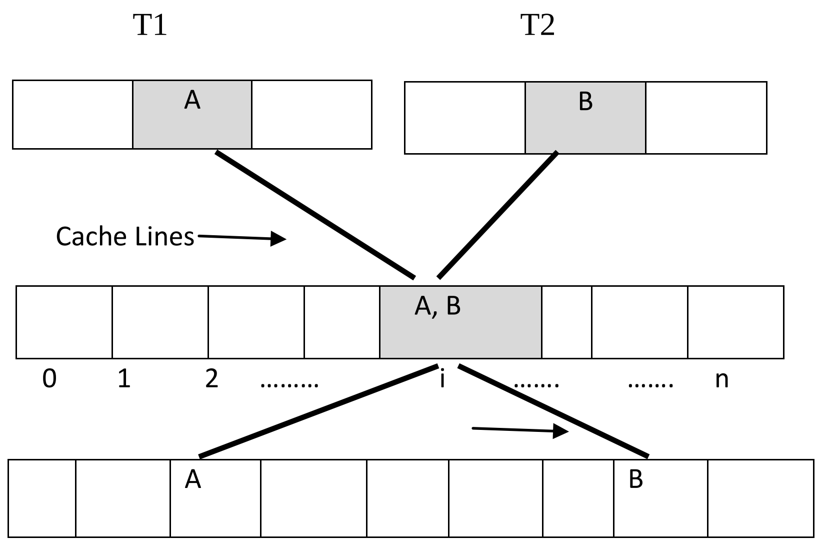


Figure 2: Thrashing Effect

1.3 Understanding Shared Cache Behaviour

This research aims to understand the locality effects and communication costs among threads in relation to a shared cache. To facilitate this, we implemented two strategies. First, we have developed a tool for generating comprehensive reuse distance for multithreaded applications. Although the use of reuse distance analysis is quite prevalent for memory optimizations for sequential programs, their use in analyzing parallel applications has been limited. This is mainly due to difficulties in measuring reuse distance for multithreaded code. Our research addresses this problem and provides a solution for measuring reuse distance for parallel applications using Pin tool frame work. Second, we develop two micro benchmarks to estimate the amount of cache sharing in multithreaded applications and also to determine how code transformation affects performance of parallel code.

CHAPTER 2

BACKGROUND

The background is divided into three sections. In the first section, description of all the tools used to capture data for experiments is presented. In second section, all the benchmarks used for experiments are described. In last section, some of the terminology used in this documentation is defined.

2.1 Tools Used

For this study, we use a set of performance measurement tools like Performance Application Programming Interface (PAPI). It is a portable interface to hardware performance counters on modern microprocessors and is widely used to collect low level performance metrics (e.g. instruction counts, clock cycles, cache misses) [2]. We also use HPCToolkit, a performance toolkit for accurately measuring and pinpointing performance bottlenecks. It uses novel techniques for measurement and analysis of parallel programs [23]. In particular, it uses statistical sampling of hardware performance counters and attributes metrics to both the calling context in which they occur and program structure, including loops and inlined procedures [23]. Lastly, we make use of Pin tool for getting memory trace for all our benchmarks. Pin tool is designed for dynamic instrumentation of program by inserting arbitrary code (written in C or C++) in arbitrary Places in the executable [7]. Thus, we could get trace of each individual thread and its specific memory reference.

2.2 Benchmarks

The proposed research focuses on evaluating reuse distance and cache sharing effects on shared cache performance using several programs from four different benchmarks suites. First, Matrix multiplication benchmark was used with blocking optimization. We used optimized Matrix multiplication to validate the relation among data locality and parallelism in relation to shared cache. Second, *mgrid* and *swim* benchmarks were taken from SPEC CPU2000 suit. mgrid demonstrates the capabilities of a very simple multigrid solver in computing a three dimensional potential field where as *swim* consists of weather prediction program for comparing the performance of current supercomputers [8]. Third, the *stream* benchmark was taken from the HPCC benchmark suit. Stream is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel [14]. Lastly, *blackscholes* and *freqmine* benchmarks were taken from PARSEC benchmark suit. Blackscholes application calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE) whereas freqmine application employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Itemset Mining (FIMI)[1].

2.3 Terminology

- **Reuse Distance:** In multicore architecture instructions are executed in parallel and hence reuse distance will consist of memory access from different threads.

Figure 3 shows an example for reuse distance. The reuse distance of D is two because two distinct data elements, A and C, are accessed in between [3].

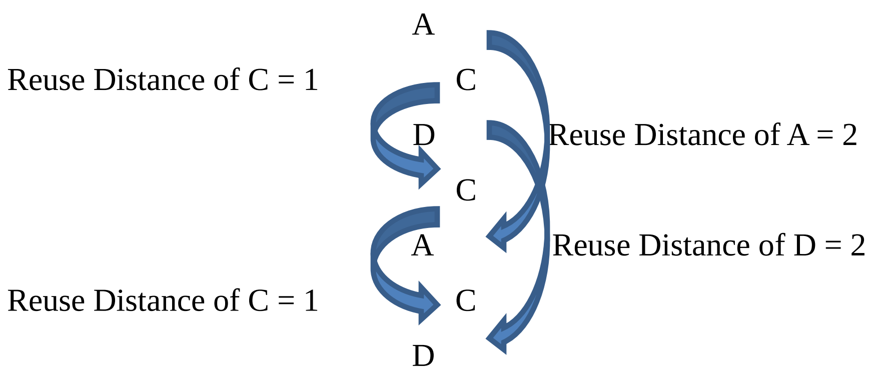


Figure 3: Reuse Distance Example

- **Temoral Locality:** Temporal locality is the tendency of an application to reference the same memory addresses that it referenced recently [26][9].
- **Spatial Locality:** Spatial locality is the tendency of applications to access memory addresses near other recently accessed addresses [26][9].

CHAPTER 3

RELATED WORK

The related work is divided into two sections. First, work related to optimizations of shared cache is presented. Second, we explore the research that pertains to reuse distance based analysis of shared cache.

3.1 Optimization of Shared Cache Performance

There has been some worked done in the past to study the behavior of shared cache on various platforms and also suitable transformations have been applied at software level.

Nikolopoulos talks about optimizing performance of Simultaneous multithreading (SMT) processors which uses shared cache at all levels. Sharing a cache between simultaneously executing threads causes excessive conflict misses. This paper proposes software solutions for partitioning shared caches on SMT processors, based on standard program transformations and additional support from the OS, or information collected from the processor hardware counters. Experimental results show that when cache is shared between threads from the same address space, performance is improved by 16-29% on average [16]. In another paper Nikolopoulos proposes a methodology for dynamically partitioning a shared cache among threads in simultaneous multithreaded architectures. The proposed methodology involves using two tile sizes, one that occupies the entire cache and another that occupies a fraction of the cache inversely proportional to the number of threads sharing the cache. By switching between the two tile sizes dynamically at run-time, Nikolopoulos' methodology reduces unnecessary conflict misses that would otherwise occur when two or more threads of execution attempt to utilize loop tile sizes that would occupy the entire shared cache. Nikolopoulos notes that this dynamic tiling implementation would benefit all processors that make use of a shared data cache, including CMPs [17].

Jeremiassen and Eggers analyze the effectiveness of compile time analysis and shared data transformation in reducing false sharing in explicit parallel programs. According to them changing the way shared data is laid out in memory to better conform to the memory reference patterns, false sharing can be eliminated. In particular, their methodology consists of grouping all data that is accessed by the same processor together and separating individual data from multiple processors. Thus, leading to overall effect of reducing cache misses [10].

Kandemir et al. presented a mathematical framework for studying the interaction between false sharing and locality for programs on shared-memory multiprocessors. They found that in those cases where the compiler can obtain outermost loop parallelism, it might be possible to simultaneously enhance spatial locality and reduce false sharing using memory layout transformations, which do not affect parallelism decisions already made by the compiler. They suggest detailed profile information can be useful in making their decisions [11]. In later work, Kandemir et al. present a compiler directed code restructuring scheme for enhancing locality of shared data in CMP's. The unique characteristic of this scheme is that, using two complementary steps (*allocation,* which determines the set of loop iterations assigned to each core, and *scheduling*, which determines the order in which the iterations assigned to a core are executed), it restructures the application code such that different cores operate on shared data blocks at the same time, to the extent allowed by data dependencies. This results in reducing the reuse distances for the shared data, and thus reducing the number of inter-core conflict misses [12].

3.2 Reuse Distance Based Analysis

Fu and Wang introduce the design and implementation of a cache performance tuning tool named CTuning, which employs a source level instrumentation method to gather program data access information, and uses a limited reuse distance model to analyze cache behavior. Experiments show that CTuning is useful in helping programmers transform their code by locating cache performance bottlenecks and perform data reorganization through analysis of cache behavior relation of some variables [4].

Fang et al. have shown that reuse distance can effectively predict locality and miss rates on a per instruction basis. Their experiments have shown that using reuse distance without cache simulation to predict miss rates of instruction is superior to using cache simulations on a single representative data set to predict miss rates on various data sizes. In addition, their analysis identified the critical memory operations that are likely to produce a significant number of cache misses for a given data size. With this information, compilers can target cache optimization, specifically to the instructions that can benefit from such optimizations most [5]. In later work, Fang et al. investigates the relationship between locality patterns and execution paths by analyzing reuse distance distribution along each dynamic path to an instruction. They relate branch history to particular locality patterns in order to determine exactly when a particular reuse distance will be exhibited by a memory operation. They suggest that being able to determine when a particular locality pattern will occur for a memory instruction allows the compiler and architecture to cooperate in targeting when to apply memory optimizations [6].

Marin and Mellor-Crummey describe an approach that uses memory reuse distance to identify an application's most significant memory access patterns causing cache misses and provide insight into ways of improving data reuse. They demonstrate the effectiveness of their approach in two scientific codes: one for simulating neutron transport and a second for simulating turbulent transport in burning plasmas. Their tools pinpoint opportunities for enhancing data reuse. Using this feedback as a guide, they transform the codes, reducing their misses at various levels of the memory hierarchy by integer factors and reducing their execution time by as much as 60% and 33%, respectively [15].

CHAPTER 4

MICROBENCHMARKS FOR ESTIMATING CACHE-SHARING EFFECTS

In this chapter, we first describe a synthetic microbenchmark we developed that enables us to estimate the amount of cache-sharing in parallel applications. Next, we described a parallel version of matrix-matrix multiplication. Finally, we present experimental Results.

4.1 Synthetic Micro-benchmark

On CMP's shared cache can have significant impact on performance because of a variety of reasons. First, since cache is shared among processors, it reduced resources underutilization, i.e. if one core idles, the other core takes all the shared cache [22]. Secondly, it allows data sharing opportunity for threads running on separate cores [22]. Finally, it reduces data storage redundancy as same data only need to be stored once [22]. Therefore several techniques are introduced to optimize shared cache performance. In one case application code is restructured such that the different cores operate on shared data block at same time, this helps in reducing reuse distance for the shared data and improves on- chip cache performance [13]. In another case, a dynamic cache partition scheme was introduced that explicitly allocates cache space among simultaneously executing process and minimizes over cache misses [20]. However, these techniques are not so successful because optimizations are not able to accurately estimate cache sharing. Therefore, it is important to get a deeper understanding of sharing to optimize programs.

Here, we present a micro benchmark that estimates cache sharing behavior. The basic idea in constructing this cache sharing benchmark was to have two data- structures accessing shared cache iteratively as shown in Figure 4. Micro-benchmarks have been used often in performance analysis of applications and architectures. For example, Cade and Qasem use a syntactic benchmark to encapsulate memory reuse patterns and parallelism characteristics of producer-consumer workloads [18]. Micro-benchmarks have also been used in the X-ray toolset for measuring hardware parameters values [24].

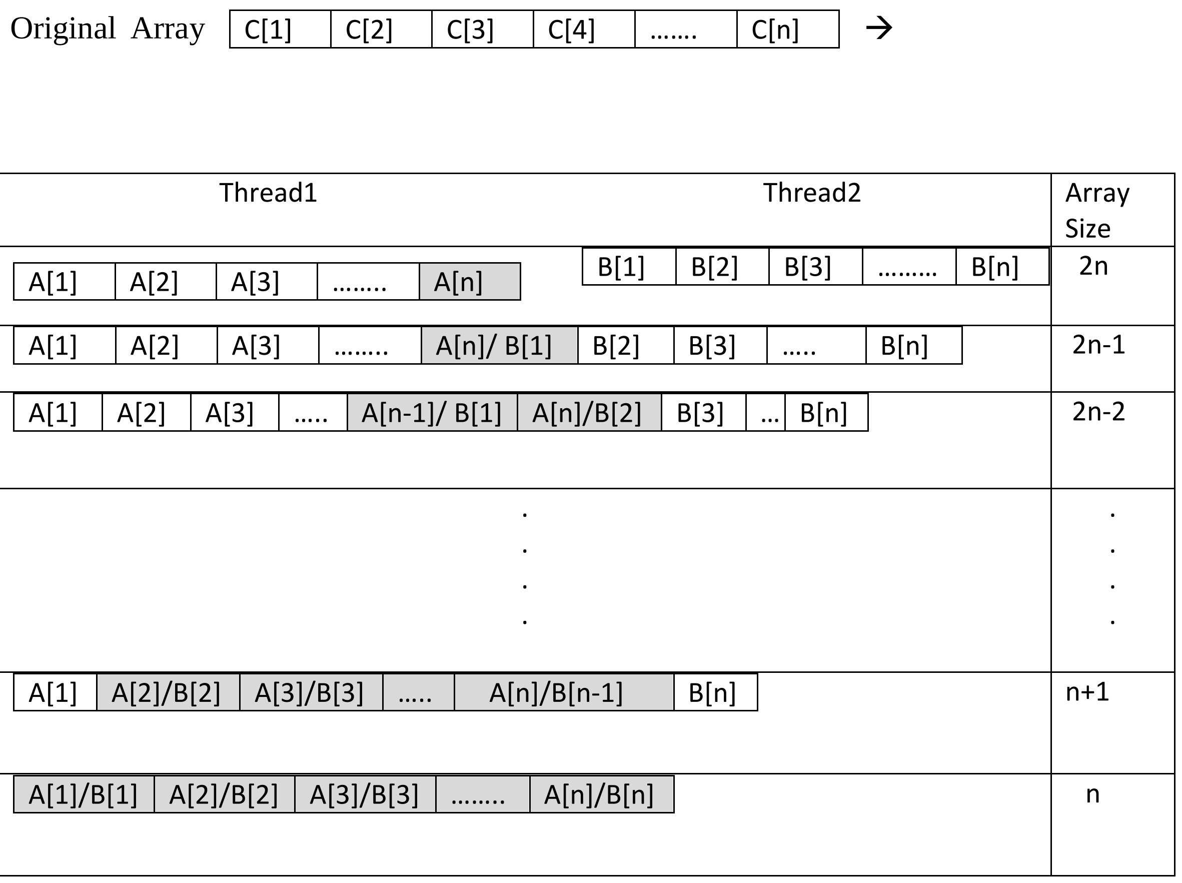

Original Array
C[1] C[2] C[3] C[4] ……. C[n]
Iterations
Thread1
Thread2
Array Size
I=1
A[1] A[2] A[3] …….. A[n]
B[1] B[2] B[3] ……… B[n]
2n
I=2
A[1] A[2] A[3] …….. A[n]/ B[1] B[2] B[3] ….. B[n]
2n-1
I=3
A[1] A[2] A[3] ….. A[n-1]/ B[1] A[n]/B[2] B[3] … B[n]
2n-2
I=n-1
A[1] A[2]/B[2] A[3]/B[3] ….. A[n]/B[n-1] B[n]
n+1
I=n
A[1]/B[1] A[2]/B[2] A[3]/B[3] …….. A[n]/B[n]
n

Figure 4: Data access patterns in Micro Benchmark

The major computational component of the synthetic benchmark involves a large array C of size n as shown in Figure 5. During execution the array of elements to be accessed are divided into two threads, such that one thread access elements from C[i] ...C [n/2] and other from C [n/2]...C[n].This leads to running two threads separately on two cores. For simplicity we consider, Thread1 to access the first half of array as A[i]...A[n] and other half accessed by Thread2 as B [i] …B[n]. During each iteration, each member of Thread1 is made to overlap with Thread 2 depending upon the number of elements to be shared among cores.

The synthetic benchmark is made to run on two different cores using two threads ID’s as shown in Figure 4. To predict the amount of sharing among Thread1 and Thread2, we consider two factors Threshold and Decrement. The Threshold is taken as the ending point for Thread1 accessing arrays from A[i]...A[n], where n is Threshold. Decrement is taken as the starting point for Thread2 accessing arrays from B[i]...B [i+n], where i is Decrement. The Overlap region is identified by Thread1 running by same amount for every iteration, i.e 0-50000 , whereas Thread2 running from 500001 to 100000 in first iteration and then decrementing by 1000 for every iteration until it completely overlaps with Thread1.

```
// Threshold= 49
 //Decrement = 50
//n = 100
for (i=0; i<n; i++)
{
  c[i]=i;
}
omp_set_num_threads(2);
#pragma omp parallel private(j) shared(Threshold)
{
    int ID = omp_get_thread_num();

      if(ID == 0)
      {   // thread 0
         #pragma omp for shared
            for(j=0;j<Threshold;j++)
               a[j]= c[j]+ 17;
      }
      else if(ID ==1)
      {
        // thread 1
          #pragma omp for shared
             for(j=Decrement;j<Threshold+Decrement;j++)
             b[j]=c[j]+17;
      }
}
```

Figure 5: Pseudo code for Cache Sharing Algorithm

Thus, during execution of the above algorithm the overlap region increases as the decrement factor is reduced. To calculate how the amount of sharing among threads has an effect on cache parameters, we concentrate on three important points during the execution of program. First, when there is no overlap among both threads, i.e Threshold = 50,000 and Decrement =50001. Second, when there is 50 % overlap, i.e Threshold = 50,000 and Decrement = 2500. Lastly, when there is complete sharing, i.e Threshold = 50,000 and Decrement = 50,000. Ideally, the miss rates can be considered to be inversely proportional to overlap region. For example, when there is no sharing involved then two threads will access elements independently and there will be at least once access to memory and thus contributing to compulsory misses, so this scenario may contribute to 100% misses, However with overlap region increasing and finally completely overlapping the other thread, the misses will be reduced to 50 %, as elements requiring access by one thread may have already been brought in by other thread. The above scenario can be justified by substituting the overlap region and size of array values in the below formula:

Let’s consider, O = Overlap

T1= Size of Thread1- Overlap

T2= Size of Thread2- Overlap

S= Total Size of Array

M= Missrate

Therefore, $M = ((T1+T2+O)/S)*100$

For following cases, let us consider S= 1000, Size of Thread1= 500, Size of Thread2 = 500.

Case 1: For no overlap, S= 1000; T1= 500; T2 = 500; O=0

$M= (1000/1000)*100 = 100\ \%$

Case 2: For 50 per cent overlap, S= 1000; T1= 250; T2 = 250; O=250

$M= (750/1000)*100 = 75\ \%$

Case 3: For 100 per cent overlap, S= 1000; T1=0; T2 = 0; O=500

$M= (500/1000)*100 = 50\ \%$

Ideally, the performance should increase by two fold when no sharing scenario is compared with complete sharing. Recall the read–read scenario discussed in Chapter-1. When Thread1 reads the data into cache, then it can be used by Thread2 which overlaps with Thread1 for access to same elements, this leads reducing number of access to memory and thus reducing latency and improving performance. However, this may not be the case when considering read-write situation. As, Thread1 may read data at the same time when Thread2 will be writing to same location, then Thread1 may get an invalid value next time when it makes a request for access to same location. Thus the performance will be degraded with incorrect data. Also, when threads are accessing the same data within shared cache then there will be lot of contention for shared resources and hence will lead to lot of conflict misses.

4.2 Block-Parallel Matrix Multiplication

One more scenario that makes shared cache optimization more critical is during use of pipelined parallel application. In case of pipelined parallelism, the problem is decomposed into chain of independent stages as shown in Figure 6[25]. Each stage is then related to its temporal neighbor in a producer-consumer fashion where *P* is the producer and *C* is the consumer [25]. In other words, each stage consumes output from a previous stage and provides input to future stages as shown in Figure 7[25]. This strategy of pipelined parallelism have positive impact in reducing the missrates and also utilizing bandwidth effectively [25]. But, the negative side of this strategy comes into picture when data locality and parallelism becomes related, when considering a cache is being shared among multiple processors [18].

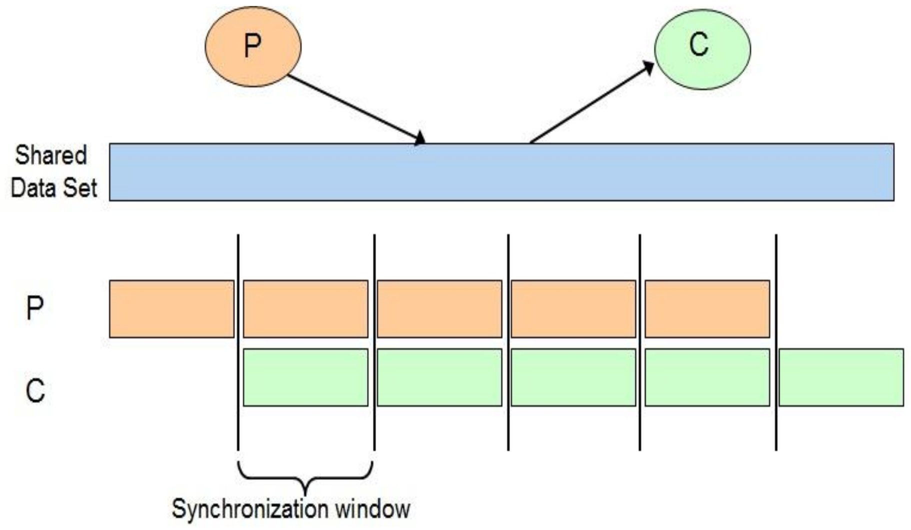


Figure 6: Producer-Consumer Model

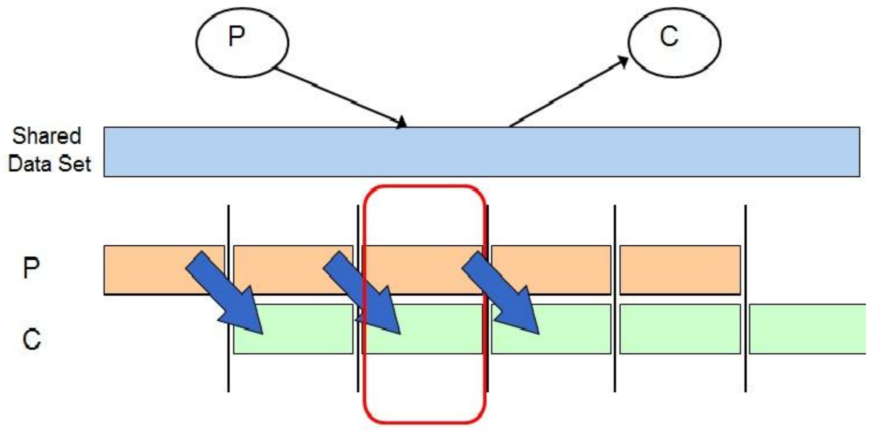


Figure 7: Producer-Consumer Working Model

Consider a pipeline – parallel application model as shown in Figure 8. In first case, if we have lot of locality of data to be processed among different threads, then data remains in cache for longer period and thus decreasing capacity misses [18]. But, on the other hand to achieve this positive impact, we need to delay threads for longer periods, this leads to unexploited parallelism [18]. In second case, if we divide the data to be processed among multiple cores, then there will be a significant impact on performance because of parallelism [18]. But, on the down side as parallelism is increased by addition of more cores or threads then there will be more contention for shared memory resources [18]. Thus the benefit of increased parallelism is then gained with an associated cost of reduced data locality [18].

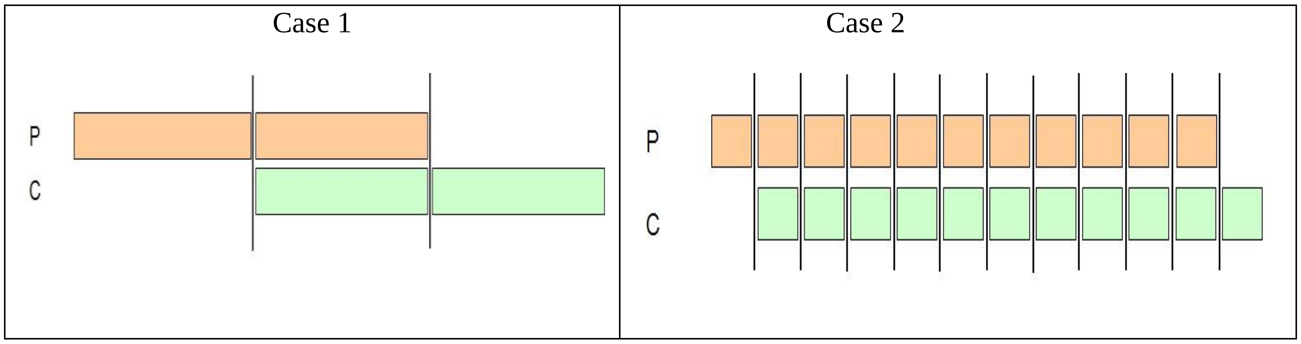


Figure 8: Locality and Parallelism related Model

To validate the above scenario, we optimize Matrix- Matrix multiplication for both parallel and sequential versions. First a Sequential matrix multiplication is implemented using three two dimensional arrays of fixed size. In sequential version, the three for loops to initialize the arrays are executed sequentially as show in Figure 9. After initialization, matrix multiplication is performed using blocking optimization. With blocking, each matrix is divided into blocks of smaller matrices i.e. 16, 32...etc., and the algorithm multiplies two submatrices, storing their product before moving on to the next two submatrices. This better exploits cache locality so that data in the cache can be reused before being replaced.

```
 //n = size of array
//bs= Block Size

for (i=0; i<n; i++)
{for (j=0; j<n; j++)
    a[i][j]= i+j;}

for (i=0; i<n; i++)
 {  for (j=0; j<n; j++)
    b[i][j]= i*j;}

for (i=0; i<n; i++)
{  for (j=0; j<n; j++)
     c[i][j]= 0;}

for(j=0; j<n; j+=bs)
 for(k=0; k<n; k+=bs)
    for (i=0; i<n; i++)
       for(kk=k;kk<MIN(k+bs,n);kk++)
           for(jj=j;jj<MIN(j+bs,n);jj++)
             c[i][jj] += a[i][kk] * b[kk][jj];

  }   /*** End of sequential region ***/
```

Figure 9: Sequential Matrix Multiplication

```
#pragma    omp    parallel    shared(a,b,c,nthreads,chunk)
private(tid,i,j,k,ii,jj,kk)
{  tid = omp_get_thread_num();
 /*** Initialize matrices ***/

#pragma omp for schedule (static, chunk)
for (i=0; i<n; i++)
{for (j=0; j<n; j++)    a[i][j]= i+j;}

 #pragma omp for schedule (static, chunk)
 for (i=0; i<n; i++)
 {  for (j=0; j<n; j++)  b[i][j]= i*j;}

#pragma omp for schedule (static, chunk)
 for (i=0; i<n; i++)
{  for (j=0; j<n; j++) c[i][j]= 0;}

 #pragma omp barrier   /*** Barrier***/

#pragma omp for schedule (static, chunk)
for(j=0; j<n; j+=bs)
 for(k=0; k<n; k+=bs)
     for (i=0; i<n; i++)
        for(kk=k;kk<MIN(k+bs,n);kk++)
            for(jj=j;jj<MIN(j+bs,n);jj++)
              c[i][jj] += a[i][kk] * b[kk][jj];

  }   /*** End of parallel region ***/
```

Figure 10: Parallel Matrix Multiplication

The parallel version of matrix multiplication is implemented as shown in Figure 10.The for loops that were serialized before are parallelized using *pragma* API's provided by OpenMp. With for loops being parallelized, initialization of arrays is done in parallel among different threads. Also, a *barrier* API is used, so that all threads are done with initialization before staring matrix multiplication. With blocking optimization, blocks of different sizes are applied to threads, which are computing different submatrices in parallel. Thus, the parallel version of matrix multiplication with blocking is ideal for finding the relation between locality and parallelism in context of shared caches.

4.3 Evaluation

In this section, we first present the results from synthetic microbenchmark that was developed to measure the amount of cache-sharing among threads. Next, the results of Optimized Matrix-matrix multiplication for both parallel and sequential versions are presented.

4.3.1 Synthetic Micro-benchmark Results

The code for synthetic microbenchmark is written in C/C++ and then parallelized using OpenMP. Number of threads used for running the benchmark is fixed to 2. For running our benchmark, a 2.33 GHz Intel Core 2 Duo E6550 (Core2, dual core) system is used. The system consists of 2 physical cores and a 4 MB L2 Cache shared among the two cores. The benchmark is compiled executed using GCC 4.3.2 on Linux 2.6.24.To facilitate probing of hardware performance counters through PAPI, the linux kernel is patched with the perfctr module. HPCToolkit profiler and PAPI are used to collect measurement of CPU cycles, L2 cache sharing, and L2 misses of benchmark.

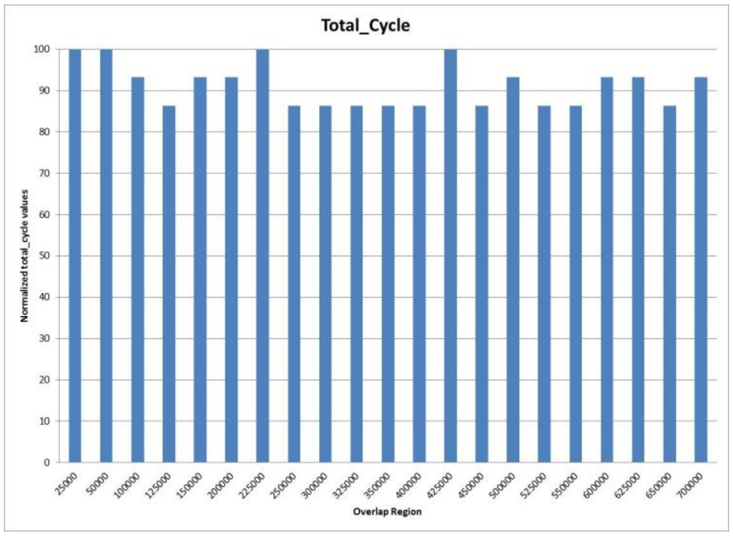


Figure 11: Total execution Cycle

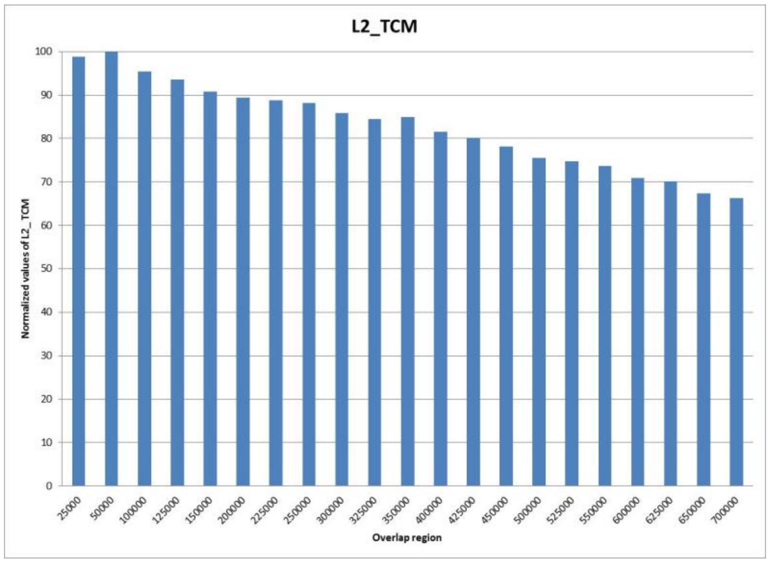


Figure 12: L2 Misses

We present charts for four different performance metrics of hardware performance counters. Each chart’s x- axis consist of an overlap region value (35000, for example is 35000 values of thread2 array which overlaps with thread 1). The y-axis represents the normalized values of different hardware counters. In Figure 11, the total execution cycle for synthetic benchmark is shown. As seen in the graph, change in clock cycle varies with a difference less that 20 % for every overlapping region, thus indicating less significant effect of cache sharing on execution time. However, Figure 12 shows a gradual decrease in the number of misses from L2 cache as the overlapping region is increased, which means there is more reuse of data in cache and thus less no of access is to main memory.

The amount of sharing among threads and number of access to shared cache is validated using counters like L2_CA_SHR and L2_Rqsts_Both_cores_Any _S. Figure 13 shows the amount of sharing two threads will do, when they are made to overlap. The results show that as the overlap region is increased, at first the sharing increases gradually, but after reaching a point it starts to descend. Figure 14, also shows the same random scenario when access is made to shared cache.

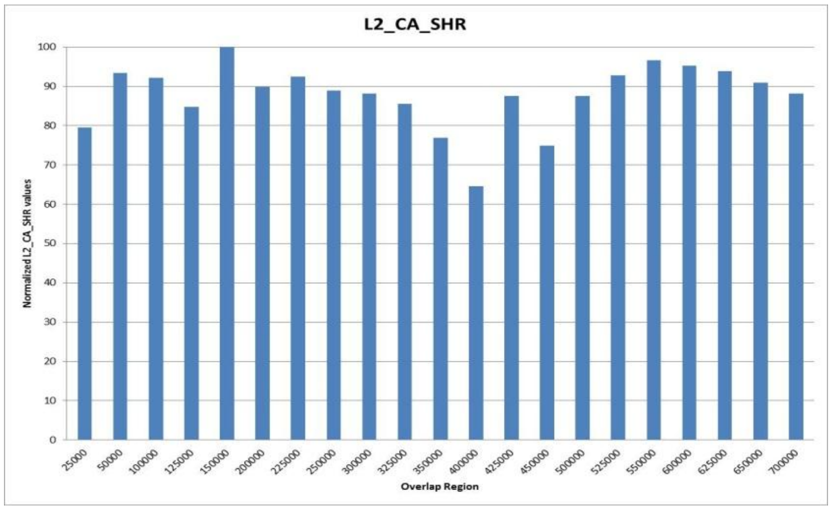


Figure 13: L2 cache shared

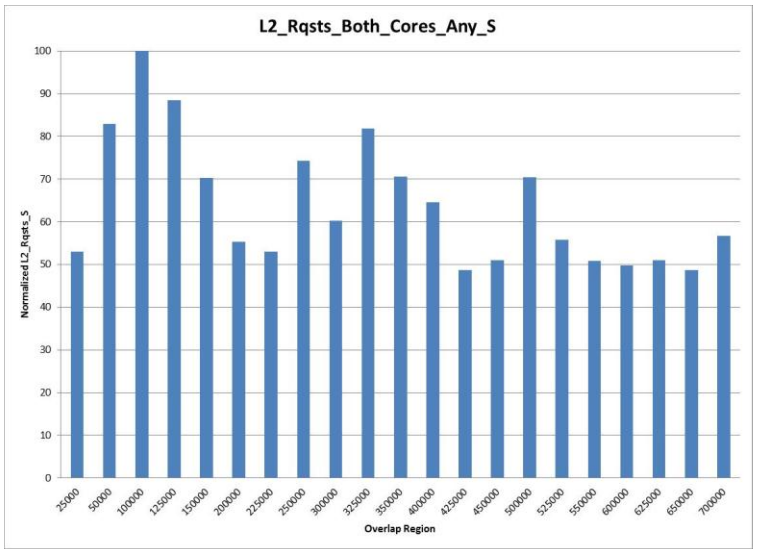


Figure 14: L2 requests to shared cache

4.3.2 Block-Parallel Matrix-Matrix Multiplication Results

A serial version of Matrix - Matrix multiplication is written in C/C++ and then parallelized using OpenMP. Number of threads for parallel version was fixed to 2. Both version of matrix multiplication is compiled and executed using GCC 4.3.2 on Linux 2.6.24.

We present charts for parallel and sequential version of matrix multiplication using four different performance metrics. Each chart's x- axis consists of block size i.e. 16, 32.etc. The y-axis represents the normalized values of different hardware counters. In Figure 15, the total execution cycle of parallel version and sequential version is shown .As seen in the graph, the execution time of sequential version is less for smaller block sizes, but thereafter increases randomly. Whereas for parallel version the case is reverse, as the execution time is more for smaller blocks, but later decreases and stabilizes for larger ones. Thus, blocking can have significant impact on performance.

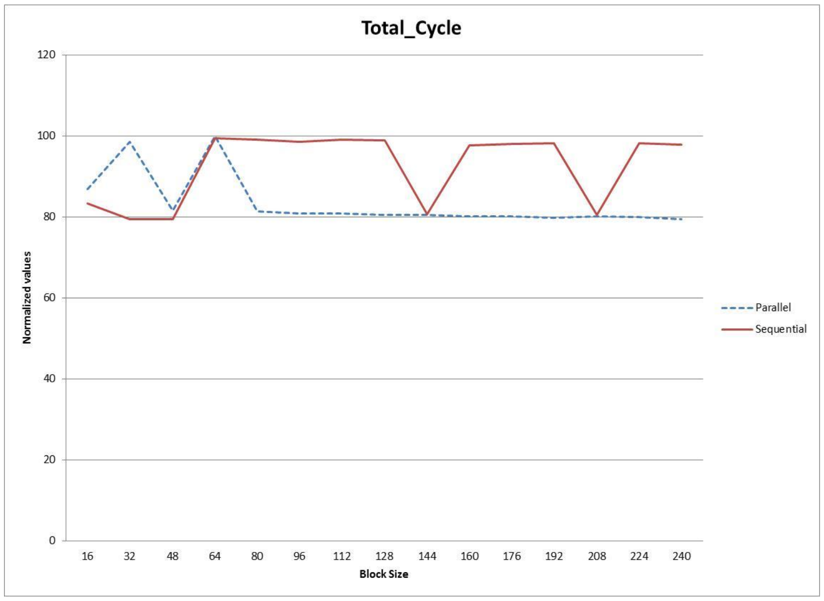


Figure 15: Total execution cycle for parallel and sequential

Figure 16, shows the number of L2 misses for both parallel and sequential version of matrix multiplication. As per the graph shown, the number of L2 misses for parallel version and sequential version decreases significantly with varied block size. However, the L1 misses in Figure 17 shows significant difference in both versions.

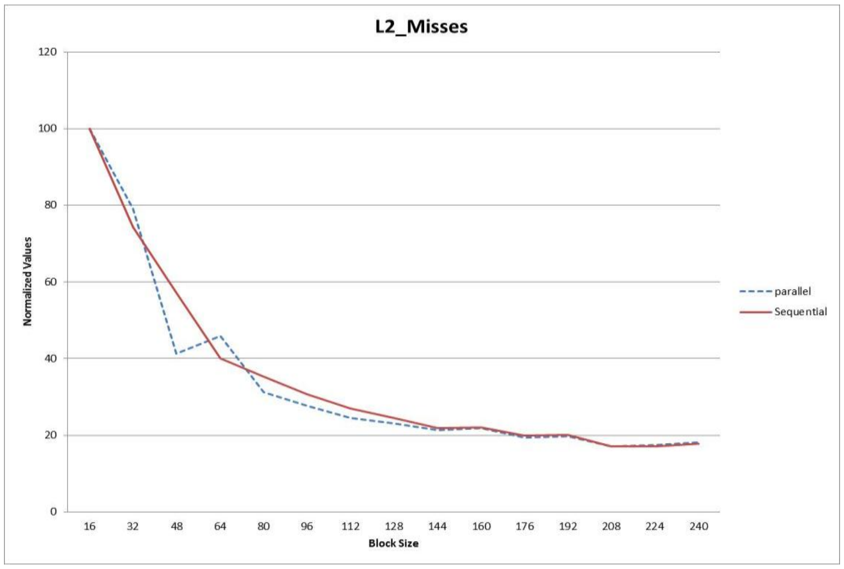


Figure 16: L2 Cache Misses for parallel and sequential

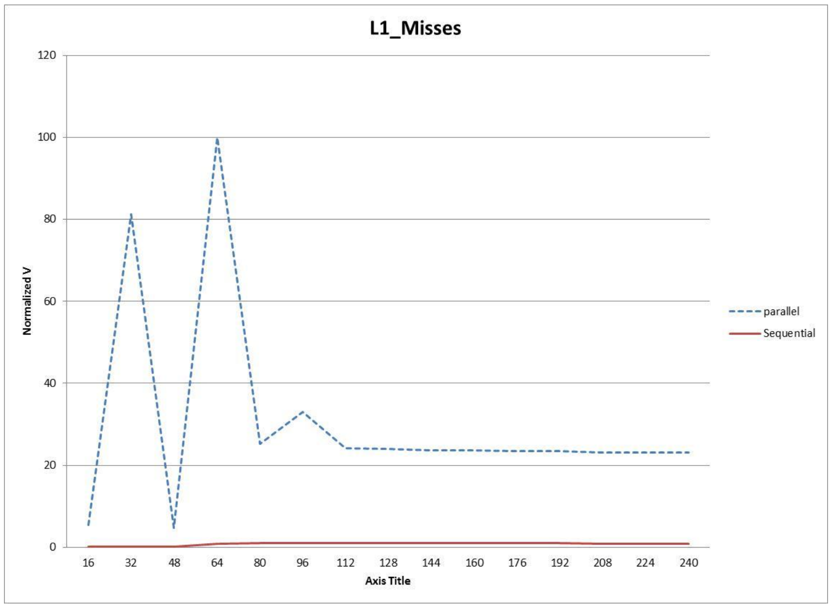


Figure 17: L1 Cache Misses for parallel and sequential

Finally, we obtain the data for both parallel and sequential versions accesses to shared cache as shown in Figure 18. The sharing among threads for parallel version shows a significant change with block size varied, whereas for sequential version the data is negligible as shown in Figure 19.

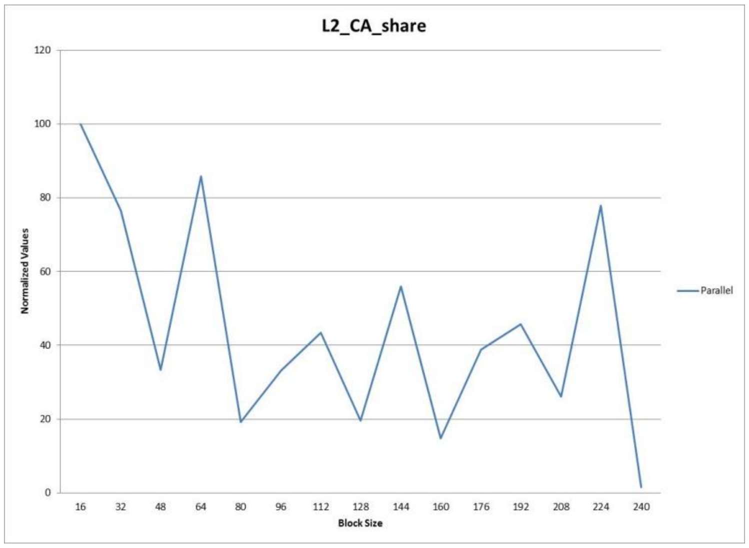


Figure 18: Parallel version shared cache data

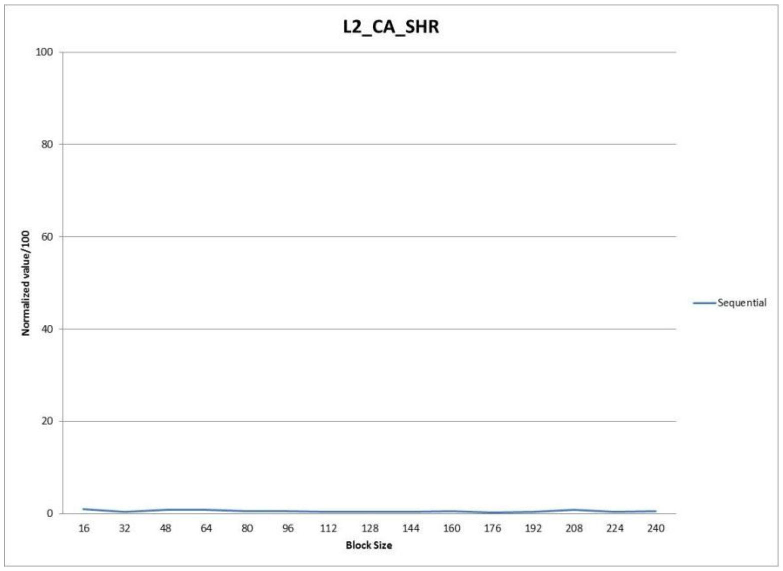


Figure 19: Sequential version shared cache data

CHAPTER 5

REUSE DISTANCE FOR SHARED-CACHES

5.1 Computing Reuse Distance in Multi-Threaded Code

In multicore architecture instructions are executed in parallel and hence reuse distance will consist of memory references from different threads. When threads run on cores that share a cache, the locality of each thread is affected by the behavior of other threads in several ways [21]. First, one thread accessing data will effectively prefetch the data for all other threads that share the same [21]. Second, all threads sharing cache can use just one copy of a given widely-read data element, reducing unnecessary replication and using the cache capacity more efficiently [21]. Third, different threads may have different working set sizes and thus require different partitions of total capacity, possibly freeing up space for other threads, or taking space away from them [21].

Reuse distances represent a measure of locality, with shorter distances being more likely to hit and longer ones less likely [21]. However, if one thread has written to the same address between two reuses of that address by another thread with a different cache, invalidation may take place and the access will miss regardless of how short the reuse distance is [21]. Thus, reuse distance alone does not consistently correspond to locality in multicore systems [21].

Our strategy provides an implementation–independent measure of locality, by computing reuse distance based on individual words accessed by different threads. Our tool is built on top of the Pin tool framework for performance analysis. The memory trace of all references was generated using Pin tool. Since the memory trace file provided by Pin tool only corresponds to single stream of references, we modify the trace file, so that it can capture read and write instructions of each thread and give their corresponding memory reference. The sample code for getting memory trace of each thread using Pin tool is shown in Figure 20.

```
VOID ThreadStart(THREADID threadid,
CONTEXT *ctxt, INT32 flags, VOID *v)
{
   GetLock(&lock, threadid+1);
   fprintf(out, "thread begin
%d\n",threadid);
   fflush(out);
   ReleaseLock(&lock);
}

VOID ThreadFini(THREADID threadid,
const CONTEXT *ctxt, INT32 code,
VOID *v)
{
   GetLock(&lock, threadid+1);
   fprintf(out, "thread end %d code
%d\n",threadid, code);
   fflush(out);
   ReleaseLock(&lock);
}
```

```
thread begin 0
thread 0 address : R 0x7fff032c6f68
thread 0 address : W 0x7fff032c6f70
thread 0 address : W 0x7fff032c6f78
thread 0 address : R 0x7fff032c6f80
thread begin 1
thread 1 address : R 0x40800200
thread 1 address : W 0x40800208
thread 0 address : W 0x7fff032c7068
thread 0 address: R 0x2b58bb1da3e8
thread 1 address : R 0x40800208
thread 0 address : R 0x7fff032c6fd8
thread begin 2
thread 2 address : R 0x40800200
thread 2 address : W 0x408001f8
thread 1 address : R 0x408000f8
thread end 1
thread 1 address : W 0x40800d30
thread 0 address : R 0x7fff032c7080
thread begin 3
```

Figure 20: Pseudo code for Memory Trace

Figure 21: Memory trace with threads

After successfully compiling all the benchmarks, the executable of every benchmark is made to run with the memory trace file for getting corresponding memory trace as shown in Figure 21. The trace file generated consists of individual thread's memory reference in hexadecimal format and their corresponding read and write instructions. The trace file also describes when the thread is created during execution. For example, In Figure 21, beginning of thread1 in the trace file is shown as "thread1 begin" and corresponding trace of thread1 is "thread 1 address: R 0x408000f8". The generated output of the memory trace is fed as input to our reuse distance algorithm, for calculating all the distinct memory references between two identical references. Since the trace generated for each benchmark is very large, to calculate the corresponding reuse distances the file is broken down in 2MB size each.

The reuse distance algorithm is implemented as shown in Figure 22. First all memory references are retrieved from the trace file and stored, After that all the unique memory references are retrieved from the file and stored. Then for each unique memory reference the first occurrence of that reference is found. If first occurrence found then all the distinct memory elements within that reference is stored in a list. After the end of the first occurrence the value of the corresponding count is retrieved and the list is deleted. This loop repeats, until all the reuse distance corresponding to particular unique reference is calculated. Finally, after getting all reuse distance counts for a particular reference and storing that in a reuse distance profile file, next unique reference is taken and compared for first occurrence and all the above tasks are repeated until all the reuse distance for all unique references are calculated and stored.

```
//c= range of unique references;
//n= range of all references

for (i=0;i<c;i++)  // unique element range
   {
        for (j=0;j<=n;j++)  //n is range of elements in
list
          {
               if (a[i]==b[j])
               {
                    if (flag == 0)
                    {//cout<<"List is empty " <<endl;
                    c1.deletefirst();
                    }
                    else
                    {
                    //cout<<"Element  :" <<  b[j] <<"
                    //"<<"count :"<<c1.count<<endl;
                    c1.deletefirst();
                    }
                          flag++;
               }
                else
                  { mem2 = b[j];
                                  cout<<mem2<<endl;
                     c1.insertlast(mem2);
                     c1.display_list();
                  }
          if(j==n)
                c1.deletefirst();
              }
     flag=0;
     }
}
```

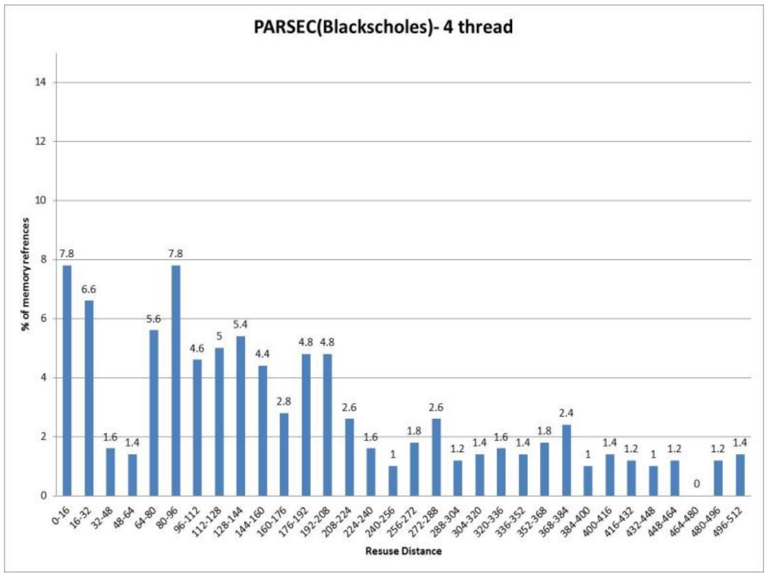
Figure 22: Reuse Distance Algorithm

The reuse distance file consisting of all reuse distance counts of all the unique memory references is used to get reuse distance histogram. For calculating the number of unique memory references with same reuse distance count an array reuse[i] is used, such that the index i correspond to the count value .For example, reuse [2] corresponds to unique reference with count 2. To find all the unique references with same count the array reuse[] is increment by 1 for every occurrence of same count, this leads to capturing all the reuse distance for a particular count and thus generating a graph that will map reuse distance with memory references.

5.2 Evaluation

We present results of applying our reuse distance based strategy to some of the high performance computing benchmarks. The benchmarks used for these experiments consisted of a streaming application taken from the HPCC suite, swim and mgrid applications taken from SPEC2000 suite, and lastly blackscholes and frequmine applications taken from PARSEC suite. The description of all the above mentioned benchmarks is provided in Chapter 2. The benchmarks are available in sequential form, but for gaining insight into performance bottlenecks of multicore environment, these applications were parallelized using OpenMp. Number of threads used in each parallel variant is 4 and 8. For running our experiments, a 2.40 GHz Intel Core 2 Quad Q6600 (Quad, 4-core) system is used. The system consists of 4 physical cores and two 4MB L2 caches, with each L2 cache shared between two cores in each socket. The benchmark tests were run with GCC 4.3.2 compiler on Linux 2.6.24 with the perfctr module installed. The memory trace of all the benchmarks was generated by dynamically instrumenting the executable of the application with Pin tool. The HPCToolkit profiler with Hardware PAPI counters was used to measure the performance metrics as discussed in Chapter 4.

We present two sets of charts for each suite of benchmarks and applications. The first chart gives a visual representation of raw histogram output of the reuse distance tool formed by 4 threads and the second represents one formed by 8 threads. Each histogram bucket along the x-axis represents a range of distances (16, for example, is shortened for less than 16 and greater than 8 and represents distances 8 - 15) and the value of a bucket along the y-axis is the percentage of total references whose reuse distance fell within that range.



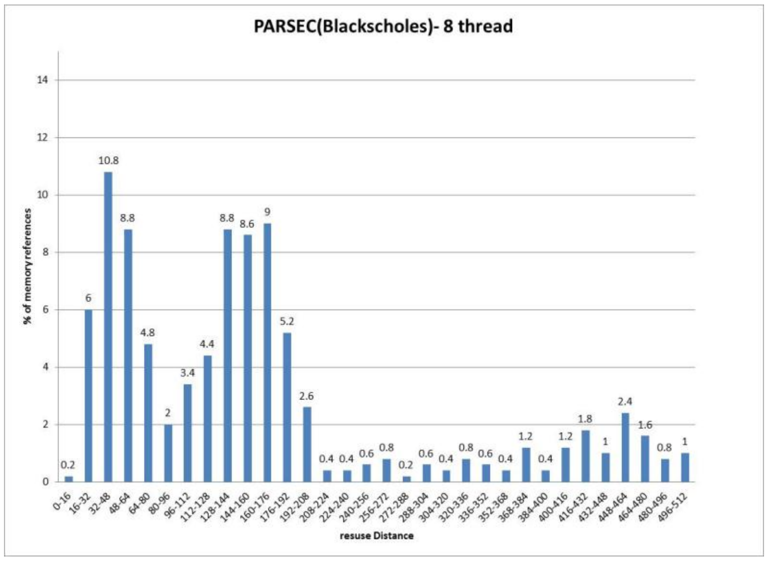
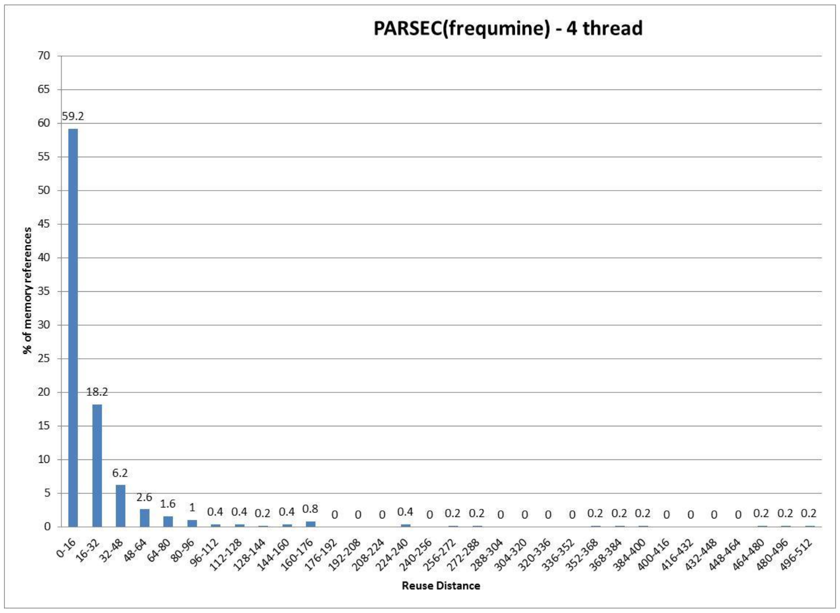


Figure 23: Reuse Distance Histogram of Blackscholes(PARSEC)

Figure 23 shows the reuse distance histogram for blackscholes. As shown in the graphs, change in locality for both charts is non-uniform and significant variations are observed. 70 % of all references for 8 threads are concentrated in first half of chart, while for 4 threads the references are distributed throughout.



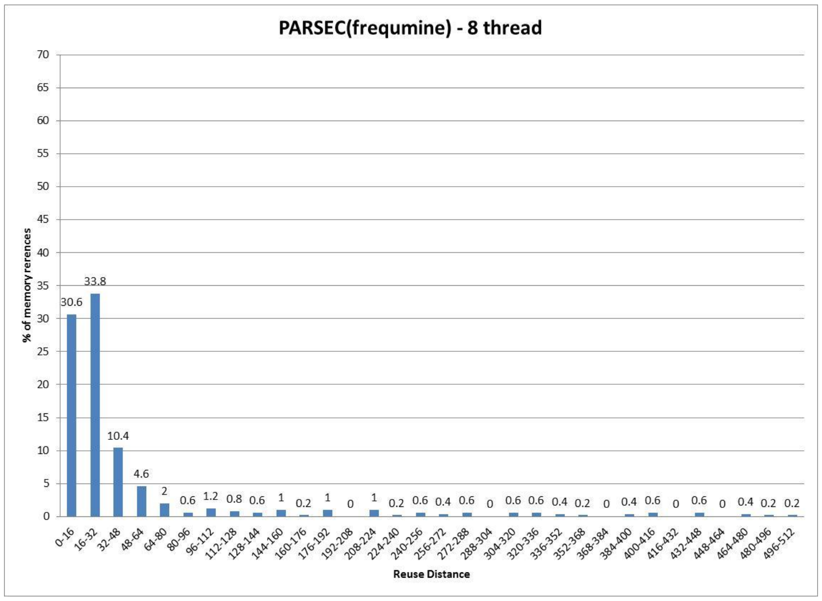
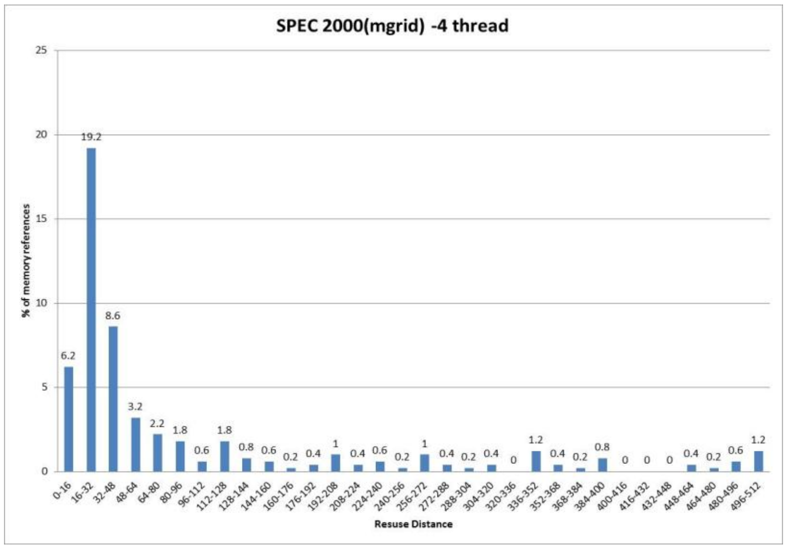


Figure 24: Reuse Distance Histogram of Frequmine(PARSEC)

Figure 24 shows the reuse distance histogram for frequmine. The reuse distance profile for frequmine is very different from the profile for blacksholes. For both the 4 thread and 8 thread versions, over 90% of locality is concentrated within RD< 80. For 4 threads we observe some variations going from RD 16 to RD 32. For the 8 thread versions this difference is much smaller.



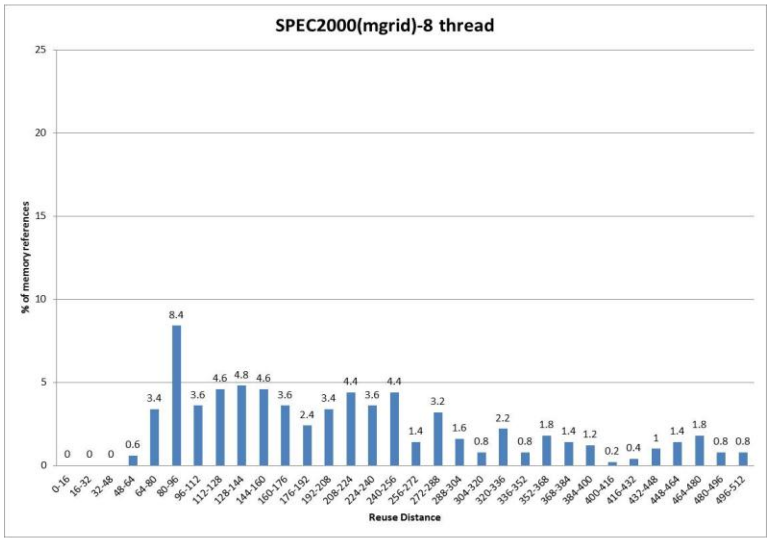
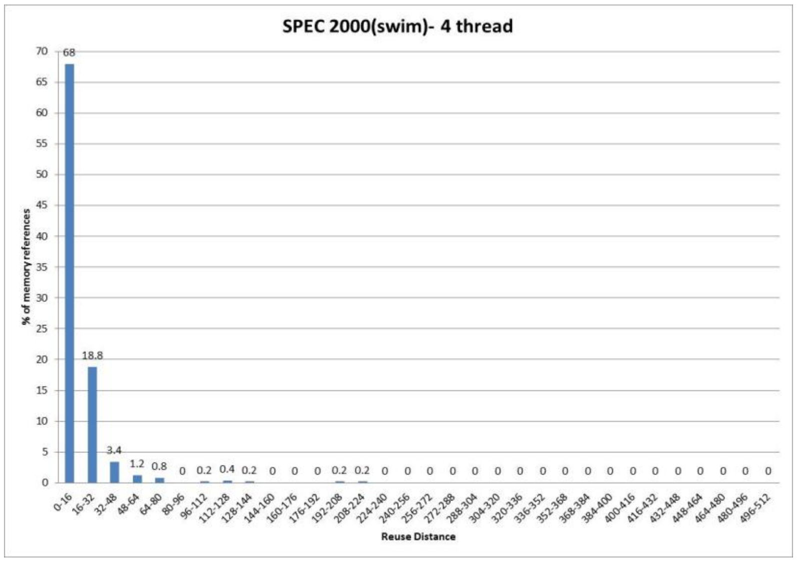


Figure 25: Reuse Distance Histogram of Mgrid(SPEC 2000)

Figure 25 shows the reuse distance histogram for mgrid. As shown in graphs, locality changes by a huge margin from 0-16 to 16–32 for 4 threads, whereas for 8 threads the locality increase is relative. 50 % of all localities for 4 threads are concentrated in first quarter, whereas for 8 threads the locality is distributed throughout the chart, with very few references falling in 0-48 buckets.



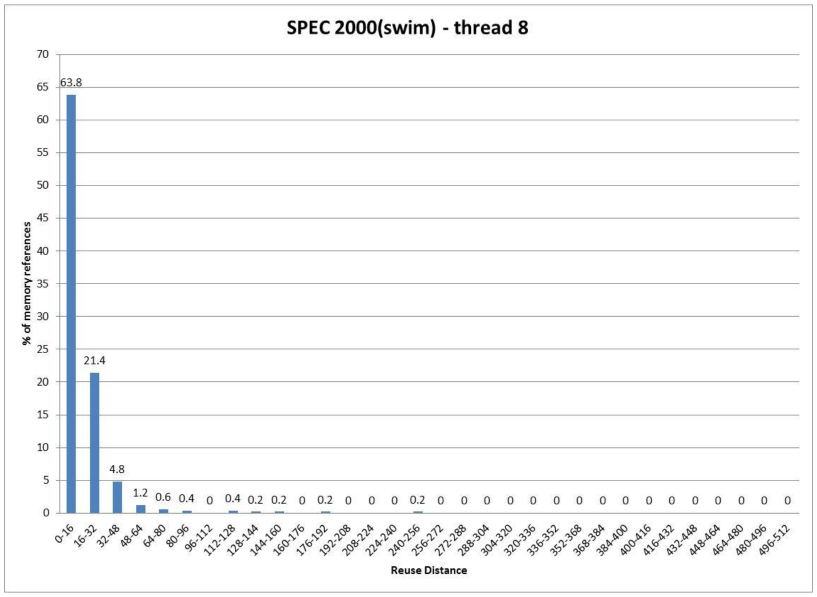
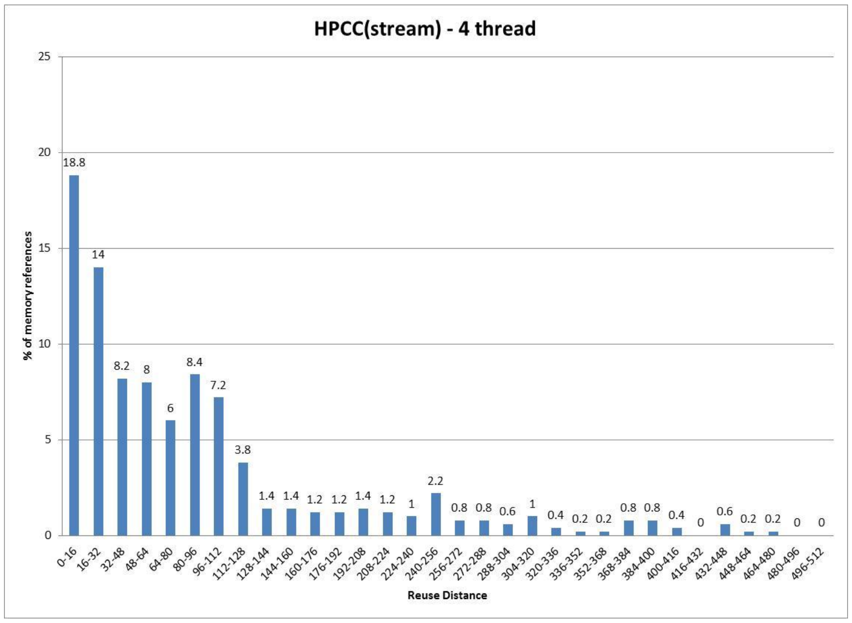


Figure 26: Reuse Distance Histogram of Swim (SPEC 2000)

Figure 26 shows the reuse distance histogram for swim. This profile is somewhat similar to freqmine with a high concentration of refs in the range 0-48. However, there is no significant difference between profiles for 4 threads and 8 threads.



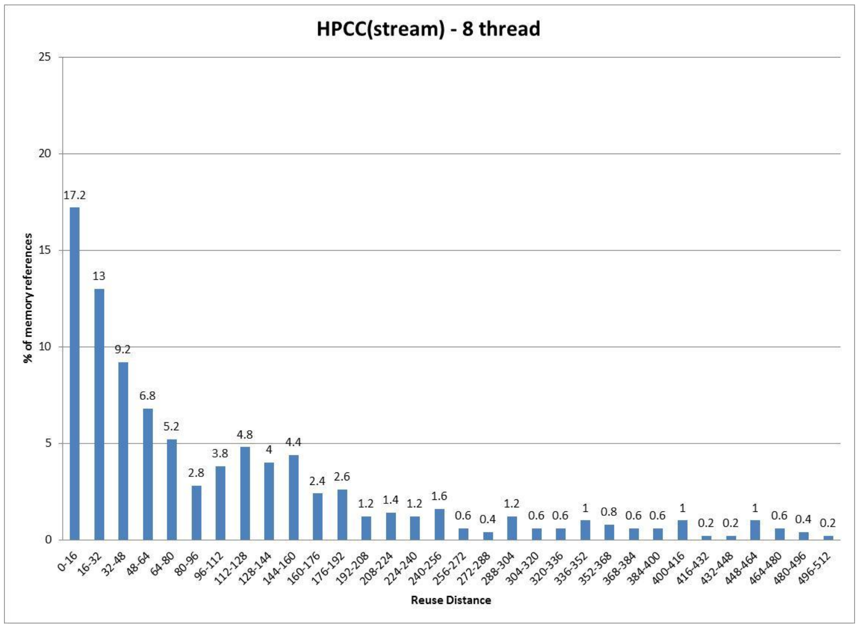


Figure 27: Reuse Distance Histogram of Stream (HPCC)

Figure 27 shows the reuse distance histogram for stream. This profile is somewhat similar to as show in graphs, change in locality varies proportionally while going from 4 threads to 8 threads. For both threads 70 % locality if concentrated within first half of chart.

Benchmarks	L1 missrates	L2 missrates	L2 shared	Total references	Unique references
Mgrid_4	23.23	10.50	32.50	3,79,904	55,515
Mgrid_8				3,79,129	49,188
Blackscholes_4	2.69	.31	7.40	3,57,797	53,330
Blackscholes_8	2.62	.20	7.72	3,54,361	54,662
Frequmine_4	2.71	1.14	9.30	3,39,183	17,386
Frequimen_8	2.31	.93	7.05	3,40,294	16,300
Stream_4	22.86	22.19	2.90	3,60,307	96,868
Stream_8	22.80	22.04	3.13	3,58,405	95,926
Swim_4	14.92	12.45	1.87	3,69,672	1,75,027
Swim_8	14.92	12.46	1.94	3,69,850	1,71,741

Table 1: Missrates and Locality Values for Parallel Benchmarks

Table 1 shows the missrates and locality values for parallel benchmarks. As seen in table, highest locality is for Frequmine benchmark, whereas lowest is for Swim benchmark. There is no much difference in L1and L2 Missrates, when going from 4 threads to 8 threads. Lastly, for Mgrid though the L1 and L2 Missrates are higher, L2 sharing is very high whereas for Stream and Swim though the Missrates are more, sharing is very minimal. Although frequmine and swim exhibit widely varying locality their RD profiles are very similar.

CONCLUSION

This thesis presented two methods of evaluating shared-cache behavior. In first case, we constructed a microbenchmark. Experiments with this benchmark showed, as the overlap region is increased among threads, the number of L2 misses is progressively reduced. However, the sharing pattern among threads shown by L2_CA_SHR was randomly distributed instead of a gradual increase. We also presented a parallel matrix-multiply kernel. The experiments with this kernel showed, that the execution time of parallel version is less sensitive to block size, whereas for sequential this factor shows a random variation. Also, by using blocking optimization the number of L2 misses was reduced by a significant amount. Finally, we presented a method for computing Reuse distance profiles for parallel applications. The collected profiles show significant variations in reuse distances. However, the values of missrates show little variation when number of threads is increased.

BIBLIOGRAPHY

[1] C.Bienia, S. Kumar,J.P. Singh,K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural ImplicationsPrinceton ,”University Technical Report TR-811-08,2008.

[2] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, “Using PAPI for hardware performance monitoring on Linux systems,” in Conference on Linux Clusters: The HPC Revolution, 2001.

[3] C. Ding, Y. Zhong, “Reuse Distance Analysis,” University of Rochester, Rochester, NY, 2001.

[4] X. Fu, R. Wang, “Ctuning: A reuse distance based cache performance tuning tool,” Journal of Electronics (China), 2009.

[5] C. Fang, S. Carr, S. Önder, Z. Wang, “Reuse-distance-based miss-rate prediction on a per instruction basis,” Proceedings of the 2004 workshop on Memory system performance, 2004.

[6] C. Fang, S. Carr, S. Önder, Z. Wang, “Path-based reuse distance analysis,” High Performance Computing and Communications, Vol. 4208, 2006.

[7] K. Hazelwood and A. Klauser, “A dynamic binary instrumentation engine for the ARM architecture,” In Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis For Embedded Systems, pp. 261-270,2006.

[8] J. L. Henning, “SPEC2000: Measuring CPU Performance in the New Millennium,” IEEE Computer, 2000.

[9] J. L. Hennessy, D. A. Patterson, A. C. Arpaci-Dusseau, “Computer architecture: a quantitative approach,” ISBN 0123704901, 2007.

[10] T. E. Jeremiassen, S. J. Eggers, “Reducing false sharing on shared memory multiprocessors through compile time data transformations,” Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 179–188, 1995.

[11] M. Kandemir, A. Choudhary, J. Ramanujam, P. Banerjee, “Reducing false sharing and improving spatial locality in a unified compilation framework,” IEEE Transactions on Parallel and Distributed Systems, Volume 14, 2003.

[12] M. Kandemir, S.P. Muralidhara, S.H.K. Narayanan, Y. Zhang, O. Ozturk, “Optimizing shared cache behavior of chip multiprocessors,” IEEE Trans. Parallel Distrib.Syst., pp. 337–354, 2003.

[13] M. Kandemir , S.P. Muralidhara , S. H. K. Narayanan , Y. Zhang , O. Ozturk, “Optimizing shared cache behavior of chip multiprocessors,” Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.

[14] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995.

[15] G. Marin, J. Mellor-Crummey, “Pinpointing and Exploiting Opportunities for Enhancing Data Reuse,” IEEE ISPASS, 2008.

[16] D. Nikolopoulos, "Code and Data Transformations for Improving Shared Cache Performance on SMT Processors," *I*nternational Symposium of High Performance Computing, vol. 2858, pp. 54-69, 2003.

[17] D. Nikolopoulos, “Dynamic tiling for effective use of shared caches on multi-threaded processors,” International Journal of High Performance Computing and Networking, vol. 2, no. 1, pp. 22–35, 2004.

[18] A. Qasem, M. J. Cade, “Balancing Locality and Parallelism on Shared-cache Mulit-core Systems,” High Performance Computing and Communications, 2009.

[19] B. Schauer, “Multicore Processors – A Necessity”, ProQuest LLC, 2008.

[20] G. E. Suh , L. Rudolph , S. Devadas, “Dynamic Partitioning of Shared Cache Memory,” The Journal of Supercomputing, v.28, pp.7-26, 2004.

[21] D. L. Schuff, B. S. Parsons, V. S. Pai, “Multicore –Aware Reuse Distance Analysis” Technical Report TR-ECE-09-08, Purdue University, 2009.

[22] T. Tian, C. Shih, "Software Techniques for Shared-Cache Multi-Core Systems" Intel(R) Software Network, 2009.

[23] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, M. Krentel, “ HPCToolkit: Performance tools for scientific computing,” Journal of Physics: Conference Series, 2008.

[24] R. Taylor, X. Li, "A Micro-benchmark Suite for AMD GPUs," 39th International Conference on Parallel Processing Workshops, pp.387-396, 2010.

[25] S. N. Vadlamani, S. F. Jenks, “The synchronized pipelined parallelism model,” In The 16th IASTED International Conference on Parallel and Distributed Computing and Systems, 2004.

[26] J. Weinberg , M. O. McCracken , E. Strohmaier, A. Snavely, “Quantifying Locality In The Memory Access Patterns of HPC Applications,” Proceedings of the 2005 ACM/IEEE conference on Supercomputing, p.50, 2005.

VITA

Suman Vara was born in Andhra Pradesh, India on February 28, 1984, the son of S.S Rao Vara and Lakshmi Vara. After completing his high school at Sri Chaitanya Jr. College in Andhra Pradesh, India, in 2002, he entered Sree Nidhi Institute of Science and Technology, Ghatkesar, India. He received the degree of Bachelor of Science from Sree Nidhi Institute of Science and Technology in July 2006. In fall 2007, he entered the Graduate College of Texas State University-San Marcos.

Permanent Address: 327 W Wood Street, Apt 406

San Marcos, Texas 78666

This thesis was typed by Suman Vara.