DYNAMIC AND TRANSPARENT SUPPORT FOR MANAGING MEMORY BOTTLENECKS IN OPENMP

A Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Besar Wicaksono
December 2011
DYNAMIC AND TRANSPARENT SUPPORT FOR
MANAGING MEMORY BOTTLENECKS IN OPENMP

Besar Wicaksono

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

Dr. Donna Stokes
Dept. of Physics

Dean, College of Natural Sciences and Mathematics
Acknowledgements

I would like to thank my advisor, Dr. Barbara Chapman, for her help in my studies. Without her guidance and insight, this work would not be possible.

I would also like to thank my thesis committee, Dr. Edgar Gabriel and Dr. Donna Stokes, for their time and comments.

I am also grateful to have worked in such a great research environment, with people whom I consider good friends. Many thanks to Ramachandra Chakenalli Nanjegowda, Sunita Chandrasekaran, Brett Estrade, and Tony Curtis for giving me guidance.

I would like to thank Mahdiyati Syukri, Guillaume Tran-Son-Tay, and David Thanoon for the great friendship during my time in Houston.

To my family, I owe a great deal of gratitude. I would not be where I am today if it were not for the love and support of my parents, parents-in-law, brother, and sister. I am deeply grateful for everything they have done for me.

Last, but certainly not least, I must thank my wife, Fitria, who has been with me every step of the way through this work. She has been supportive and understanding through this entire process.
DYNAMIC AND TRANSPARENT SUPPORT FOR MANAGING MEMORY BOTTLENECKS IN OPENMP

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Besar Wicaksono
December 2011
Abstract

OpenMP is the de-facto standard for shared memory parallel programming. It offers a straightforward approach to utilizing multicores platforms without the need to explicitly manage multiple threads. With respect to their performance, OpenMP applications are rather sensitive to the executing threads’ memory accesses both at cache level as well as at page level. Large thread counts magnify the negative impact of memory bottlenecks at both levels. However, many novice OpenMP developers are not aware of performance problems due to memory bottlenecks and how to overcome them.

To address these problems, and particularly to support the untrained OpenMP application developer, we have developed transparent support for overcoming memory bottlenecks in OpenMP applications. To achieve this, we have extended DARWIN, our dynamic optimization framework for OpenMP code. We have enhanced its design to increase its modularity and have added features to support performance analysis. We have demonstrated how this framework can be used by creating and deploying strategies to overcome data locality problem in OpenMP applications running on a ccNUMA platform and to detect and overcome performance problem caused by false sharing. The experimental results show that we were able to gather meaningful performance-related information with low overheads and that we can utilize this information to find the source of the performance bottleneck for both problems. The results also show that most applications experience noticeable performance improvements as a result of the optimization performed.
## Contents

1 Introduction 1
  1.1 Motivation 2
  1.2 Contributions 3
  1.3 Organization 4

2 Background 5
  2.1 Introduction to OpenMP 5
    2.1.1 OpenMP API 6
    2.1.2 OpenUH compiler suite 7
    2.1.3 OpenMP compiler translation 9
    2.1.4 OpenMP runtime library 9
  2.2 Memory bottlenecks in OpenMP 11
    2.2.1 Data locality in cache coherent non-uniform shared memory architecture 12
    2.2.2 False sharing in shared memory multiprocessors 15
  2.3 OpenMP collector API 18

3 Dynamic Optimization Framework for OpenMP 20
  3.1 DARWIN - Initial design 22
    3.1.1 Framework structure 22
3.1.2 Design limitation ............................................. 24
3.2 DARWIN enhancement ........................................... 26
  3.2.1 Collector tool ............................................ 27
  3.2.2 Performance monitoring .................................... 29
  3.2.3 Data allocation ............................................ 32
  3.2.4 Data management .......................................... 36
  3.2.5 Optimizer .................................................. 38
  3.2.6 Utilities .................................................... 41
3.3 Workflow of DARWIN ............................................ 44

4 Data Placement Optimization ........................................ 47
  4.1 Methodology .................................................. 48
    4.1.1 Data access pattern type identification ..................... 49
    4.1.2 Data locality problem detection ........................... 50
  4.2 Experimental results ......................................... 51
    4.2.1 Monitoring phase results .................................. 51
    4.2.2 Optimization phase results ................................. 53

5 False Sharing Detection ............................................ 57
  5.1 Methodology .................................................. 58
    5.1.1 Stage 1 : Detecting cache coherence problem .............. 58
    5.1.2 Stage 2 : Isolating the data structures ................... 59
  5.2 Experimental results ........................................... 60
    5.2.1 Results of detecting cache coherence problem .............. 61
    5.2.2 Linear_regression ........................................ 63
    5.2.3 String_match ............................................ 65
    5.2.4 Results of memory alignment ............................... 67
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Common directives, routines, and environment variables of OpenMP API</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Parallel reduction implementation using Pthreads vs OpenMP</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>OpenMP compiler translation result of the reduction code in figure 2.2(b)</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Diagram of a ccNUMA platform with 4 processors.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>OpenMP code snippet with false sharing problem.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Initial design of DARWIN</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Modules of DARWIN</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td><code>CCollectorTool</code> class definition.</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>A performance tool for measuring execution time of each thread in OpenMP implicit barrier.</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Implementation of the performance monitoring module.</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>C structure holding the monitoring result.</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Data allocation information.</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Capturing dynamic data allocation by interposing the <code>malloc</code> routine.</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td><code>CDataManager</code> class definition.</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td><code>CPlacementOptimizer</code> class definition.</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>Example content of the TAU profile generated by <code>darwin_exporter</code></td>
<td>42</td>
</tr>
<tr>
<td>3.12</td>
<td>Example Paraprof visualization of DARWIN collected information</td>
<td>42</td>
</tr>
</tbody>
</table>
3.13 DARWIN phases: (a) Monitoring phase. (b) Optimization phase. . .  45

4.1 Data access pattern type: (a) Block. (b) Cyclic with 5 cycles. . . . .  49

4.2 (a) A data structure with latency imbalance issue. (b) A data struc-
ture that is already optimized for ccNUMA platform. . . . . . . .  50

4.3 (a) The slowdown of the monitoring phase. (b) Overhead breakdown.  52

4.4 Optimization result on class A with 4 threads: (a) Performance speedup.
(b) Optimization overhead. . . . . . . . . . . . . . . . . . . . . . . . . .  55

4.5 Performance speedup: (a) class A with 4 threads. (b) class B with 8
threads. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  55

5.1 The speedup of the original program. . . . . . . . . . . . . . . . . .  61

5.2 Linear_regression memory access visualization . . . . . . . . . . . .  64

5.3 Adjusting the data alignment in linear_regression. . . . . . . . . . .  65

5.4 String_match memory access visualization . . . . . . . . . . . . . . .  66

5.5 Adjusting data alignment in string_match. . . . . . . . . . . . . . . .  67

5.6 The speedup after adjusting the memory alignment. . . . . . . . . .  68

5.7 (a) The slowdown of the monitoring phase. (b) Overhead breakdown.  69
# List of Tables

2.1 Several OpenMP runtime library routines from the OpenUH compiler suite ........................................ 11

2.2 ccNUMA effect on CG application ........................................ 13

2.3 Execution time of OpenMP code from Figure 2.5. .................... 17

2.4 OpenMP collector events ................................................. 19

3.1 Data centric information captured from CG of NPB. ............... 37

3.2 Result of aggregating the data centric information. ............... 38

4.1 Captured parallel regions and data allocation. ...................... 52

4.2 Analysis results. ......................................................... 54

5.1 Cache invalidation event measurement result. ...................... 62

5.2 Data allocation information of several variables in `string_match`. 67

5.3 `Reverse_index` cache invalidation event measurement result. .... 68
Chapter 1

Introduction

Exploiting parallelism with multicore architecture has been a widely accepted method to gain more performance. Multicore architecture is used in a wide range of systems starting from consumer products like smart phones, gaming consoles, and personal computers to high performance scientific computing platforms. Majority of modern computers currently are equipped with processors containing dual, triple, or even quad cores sharing the same memory address space. We refer these computers as shared memory parallel computers (SMP).

The OpenMP Application Programming Interface is the de-facto standard for shared memory parallel programming. It offers a straightforward approach to performing parallel computations and relieves the application developers from expending effort to manage multiple threads. As a basic example, developers just need to identify which part of their code that need to and can be parallelized, and then annotate that part of the code with the appropriate OpenMP constructs to express parallelism.
The other tedious tasks such as creating a multi-threaded code and managing the threads can be offloaded to the OpenMP compiler and runtime system.

Despite the benefit of higher productivity, OpenMP applications may still face the common performance problems that exist in shared memory parallel applications. To solve a performance problem, the inefficient code that is responsible for the bottlenecks needs to be found and modified to a better one. The task of detecting and solving performance bottleneck might not be easy, especially for novice developers. Many novice OpenMP developers are not aware of common impediments to performance and need help to create efficient code. To address this issue, and particularly to support the untrained OpenMP application developers, transparent program optimization would be ideal.

1.1 Motivation

With respect to their performance, multi-threaded shared memory applications, including OpenMP, are rather sensitive to the executing threads’ memory accesses both at the cache level as well as at the page level. At the cache level, a performance problem called “false sharing” may occur on multi-core platforms because blocks of data are fetched into cache on per-line basis. When one thread accesses data that happens to be on the same line as the data simultaneously accessed by another thread, both need up-to-date copies of the same cache line. In order to maintain the consistency of the shared data, the processor may then generate additional cache misses that degrade performance.
At page level, data locality can be a significant issue of performance since data is allocated to physical memory bank on per-page basis. This problem can occur particularly on a cache coherent Non-Uniform Memory Access (ccNUMA) system, where different memory banks are connected to different multi-core processors. In such system, a processor has to use the system interconnect to access the memory banks connected to the other processors. This remote memory access has a longer latency and can be a major bottleneck if it happens frequently.

Many novice OpenMP developers are not aware of memory bottlenecks caused by the false sharing effect and the ccNUMA behavior. Even with this awareness, it can be very hard for the developers to correctly identify the source of such performance problems, as it requires some amount of understanding of the underlying system. Furthermore, they may not know how to adapt an application in order to fix the problem.

1.2 Contributions

Our objective in this thesis is to utilize our dynamic optimization framework called DARWIN to optimize data placement on a ccNUMA platform and to find the source of a false sharing problem. A summary of the contributions resulting from the work in DARWIN includes the following:

- We have enhanced the design of DARWIN by making it modular and adding new features to support optimization for several performance problems.
• We have created and deployed a flexible strategy to overcome data locality problems on a ccNUMA platform.

• To further demonstrate the flexibility of the framework, we reused the performance data utilized in the above performance problem to find the source of false sharing problem with low overhead.

1.3 Organization

The rest of the thesis is structured as follows: Chapter 2 gives background information about OpenMP and related works on several performance problems caused by memory bottlenecks. Chapter 3 discusses the DARWIN framework and our enhancements to it. Chapter 4 and 5 presents our strategies to overcome data locality problem on a ccNUMA platform and to find the source of a false sharing problem. Chapter 6 is the conclusion that summarizes our findings from this study.
Chapter 2

Background

2.1 Introduction to OpenMP

OpenMP [9] is a widely used API to express shared memory parallelism in C/C++ and Fortran program. It provides high level abstraction to implicitly implement multithreaded application with fork-join model of parallel execution. An OpenMP application starts with a single process, which is referred as the master thread. This thread performs the execution in serial until it encounters a parallel region. When the master thread reaches a parallel region, it forms a team of threads that consist of the master thread and additional worker threads. The master thread distributes the work from the encountered parallel region to all of the threads in the team. The thread team then executes the assigned work in parallel. After the team completes the work, all threads in the team synchronize and terminate, leaving only the master thread to proceed with serial execution of the program. This process repeats itself.
when the master thread encounters another parallel region.

### 2.1.1 OpenMP API

OpenMP API consists of a set of compiler directives, user level library routines, and environment variables. These features form a high level abstraction to hide the complexity of parallelizing a program, which usually involves a large amount of code restructuring. The compiler directives are inserted to the code to specify the parallelization strategy of a code region. The user level routine and environment variable can be used by the application developer to control particular multithreading configuration, such as the number of threads and scheduling strategy of a parallel region. Figure 2.1 presents the commonly used OpenMP directives, user level library routines, and environment variables.

![Diagram of OpenMP API](image)

**Figure 2.1: Common directives, routines, and environment variables of OpenMP API**
Figure 2.2 presents the comparison of parallel reduction implemented using *pthreads* [1] multithreading library and OpenMP API. Both codes accumulates the index of the loop and store the result into the *sum* variable. The parallelization is done by distributing the loop across threads. In the *pthreads* version, the developer has to manually create additional threads by using the *pthread_create* routine, write the parallel version of the code statement inside the loop that perform the accumulation, and destroy the threads by using *pthread_join* routine. With OpenMP, the developer just need to identify the code region that need to be parallelized and annotate it with the *parallel for* directive. The tedious task of writing parallel codes are offloaded to the OpenMP implementation, which consist of a compiler and runtime system. The implication of using OpenMP is that the developer can have more focus on the algorithm, reduce the code cluttering, and manage the code in a better way.

### 2.1.2 OpenUH compiler suite

OpenUH [28] is an open source compiler suite for various programming languages, such as C, C++, and Fortran. It is a branch of Open64 compiler suite, which is a modular and robust optimizing compiler for the Itanium and x86 architectures. The OpenUH compiler suite provides an implementation of OpenMP compiler and runtime library. The compiler side contains a front-end module for parsing the OpenMP directives, a pre-lower and lowering modules for translating the OpenMP directives into explicitly parallel codes. The translation result contains several calls to the OpenMP runtime library, which has various routines primarily for creating and destroying threads, distributing work across the threads, and providing synchronization.
```c
... 
pthread_t thread_handle[num_worker]; 
for(i=0;i<num_worker;i++) 
{ 
    pthread_create(&thread_handle[i],NULL, 
    parallel_work,(void *)(i+1)); 
} 
parallel_work((void*)0); 
for(i=0;i<num_worker;i++) 
{ 
    pthread_join(thread_handle[i],NULL); 
} 

... 
void *parallel_work(void* input) 
{ 
    int i, lower, upper, local_sum, 
    iter_per_thread; 
    long id = (long)input; 
    local_sum = 0; 
    iter_per_thread = N/NUM_THREADS; 
    lower = ((int)id) * iter_per_thread; 
    upper = ((int)id + 1) * iter_per_thread; 
    for(i=lower;i<upper;i++) 
    { 
        local_sum += i; 
    } 
    pthread_mutex_lock(&mut); 
    sum += local_sum; 
    pthread_mutex_unlock(&mut); 
    if(id != 0) 
        pthread_exit(NULL); 
}
```

(a) Pthread library code

```c
int i, sum; 
sum = 0; 
#pragma omp parallel for 
    num_threads(NUM_THREADS) 
    reduction(+:sum) 
    for(i=0;i<N;i++) 
    { 
        sum += i; 
    }
```

(b) OpenMP API code

Figure 2.2: Parallel reduction implementation using Pthreads vs OpenMP
support like the barrier and critical region. OpenUH uses POSIX threads API, usually referred as *pthreads*, to implement the OpenMP runtime library. *Pthreads* API is a portable thread level support available on various UNIX-based operating systems like Linux, Solaris, and FreeBSD.

### 2.1.3 OpenMP compiler translation

As shown by the reduction example in Figure 2.2, the developer uses the OpenMP compiler directives to express parallelism. The OpenMP compiler translates these directives into multithreaded code by replacing the annotated OpenMP compiler directives with calls to OpenMP runtime library routines. The compiler also generates a function that encapsulates the work to be performed by each thread. The pointer of this function is used as an identifier of each parallel region in the program and is passed into the runtime library routine responsible for the forking operation.

Figure 2.3 gives an example of how the the OpenMP reduction code is translated by the OpenUH compiler. It shows that the `parallel for` directive is translated into a call to `__ompc_fork` routine, which is an internal routine provided by the OpenMP runtime library. The fork routine receives a pointer to routine `__omp_do_main_1` that distributes the loop iterations across threads.

### 2.1.4 OpenMP runtime library

The OpenMP runtime library supports the compiler by providing routines for managing threads, including thread creation, synchronization, and scheduling. Figure 2.3
extern _INT32 main(...)
{
    ...
    __ompc_fork(..., &__ompdo_main_1);
    ...
    return 0;
}  /* main */

static void __ompdo_main_1(...)
{
    ...
    __ompc_static_init_4(...);
    ...
    for(__mplocal_i = __ompv_temp_do_lower0;
        __mplocal_i <= __ompv_temp_do_upper0;
        __mplocal_i = __mplocal_i + 1)
    {
        __mplocal_sum = __mplocal_sum + __mplocal_i;
    }
    __ompc_reduction(__ompv_temp_gtid,
                     (_UINT64) & __mplock_1);
    sum = sum + __mplocal_sum;
    __ompc_end_reduction(__ompv_temp_gtid,
                         (_UINT64) & __mplock_1);
    __ompc_barrier();
    return;
}  /* __ompdo_main_1 */

Figure 2.3: OpenMP compiler translation result of the reduction code in figure 2.2(b) gives some usage examples of the OpenMP runtime library routines. The __ompc.fork routine is used to initiate each parallel region. In this routine, the master thread configures additional worker threads to work on the code statements of the parallel region. The __ompc.static.init_4 routine initializes a work sharing construct by calculating the start and end index of the iterations that need to be executed by a thread. The __ompc.reduction and __ompc.end.reduction routines respectively indicate the start and end of a critical region that implement a reduction operation. The __ompc.barrier routine implements a barrier synchronization. Table 2.1 provides a few of commonly used OpenMP runtime routines from the OpenUH compiler suite.
<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__omp_init_rtl</td>
<td>Initializes the OpenMP runtime library</td>
</tr>
<tr>
<td>__omp_fini_rtl</td>
<td>releases the resources allocated by the OpenMP runtime library</td>
</tr>
<tr>
<td>__omp_fork</td>
<td>configure the threads to work on a parallel region</td>
</tr>
<tr>
<td>__omp_barrier</td>
<td>synchronizes all threads in a team</td>
</tr>
<tr>
<td>__omp_static_init_4</td>
<td>initializes the iterations of a loop that need to be executed by each thread</td>
</tr>
</tbody>
</table>

Table 2.1: Several OpenMP runtime library routines from the OpenUH compiler suite

### 2.2 Memory bottlenecks in OpenMP

The previous sections showed the simplicity of OpenMP and how it offers the developers a portable solution to reduce the challenges of writing a multithreaded program and to focus on their algorithm. Developers can write their OpenMP code once and deploy it on different platforms, as long as there is a compiler support. Unfortunately, this transparency of OpenMP can hide some potential problems from the developers [42]. These are particularly the problems related to the way parallelism is applied in the hardware. In this section, we discuss the data locality problem in cache coherent non-uniform shared memory architecture and the false sharing effect. Both are hidden problems that can significantly degrade the performance of shared memory parallel applications, including OpenMP. We explain the cause of both problems and give example of how they can impact the performance.
2.2.1 Data locality in cache coherent non-uniform shared memory architecture

Cache coherent non-uniform memory access (ccNUMA) architecture, as depicted by figure 2.4 is a form of SMP that consists of nodes interconnected by a network, where each node contains its own local memory system used to serve one or more processors. In this architecture, all processors can share the same virtual memory address space and the cache coherency is maintained by a special-purpose hardware.

![Diagram of a ccNUMA platform with 4 processors.](image)

Figure 2.4: Diagram of a ccNUMA platform with 4 processors.

Compared to SMP with uniform memory access (UMA) architecture, a ccNUMA platform can scale to much higher number of processors without being limited to a single shared memory resource. However this architecture introduce another issue related with memory access latency that can hurt the performance of an application. Each processor in a ccNUMA platform can directly access the local main memory, but has to use the system interconnect to access the memory banks of the other processors (remote memory). The remote memory access has a higher latency than the local memory access, depending on the distance between the processor and the
memory bank where the data is located. For that reason, remote memory accesses can be a major bottleneck if the data are not carefully placed [33, 36, 29].

Table 2.2 shows an example of how remote memory accesses can affect the performance of CG application from NAS Parallel Benchmark suite [22]. This application was executed in a ccNUMA platform with four NUMA nodes; each has two processors. The large number of remote memory accesses, clearly reduced the benefit of using four and eight threads.

<table>
<thead>
<tr>
<th>Metric</th>
<th>1-thread</th>
<th>2-threads</th>
<th>4-threads</th>
<th>8-threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (second)</td>
<td>15.4</td>
<td>8.2</td>
<td>8.2</td>
<td>4.8</td>
</tr>
<tr>
<td>Local memory access (count)</td>
<td>119197</td>
<td>120642</td>
<td>91884</td>
<td>25795</td>
</tr>
<tr>
<td>Remote memory access (count)</td>
<td>431</td>
<td>216</td>
<td>39165</td>
<td>49217</td>
</tr>
</tbody>
</table>

Table 2.2: ccNUMA effect on CG application

Some previous works had been made to help developers tune the applications for ccNUMA platform. Operating systems like Linux and Solaris provide first-touch page allocation policy, which stores a data page to the local memory of the processor that first accesses it [25, 8]. A system library, called libnuma, provides developers with routines to allocate data pages on specific memory bank. However, it is difficult for the developers to utilize the operating system or the system library features without first knowing which data page has to be placed at which destination node.

Several works like Memphis [33], UPMLib [36], and Marathe et al. [29] develop tools that can help developer on deciding the best page placement strategy. Memphis helps the developer to find the data structures in the code that have problematic remote memory accesses. Memphis utilizes hardware performance monitoring unit
(PMU) to sample memory accesses. It uses debugging information from the compiler to map each sample to its corresponding variable name and the location in the source code. Memphis then counts the number of remote accesses of each variable and prints the result. To optimize the program, developers need to manually change the code based on the result from Memphis. The information provided by Memphis is limited to global or static variable because only the debugging information is used.

UPMLib offers an online dynamic page migration strategy by moving a particular page to a node that has the most frequent access to it. The information about the page access frequency is obtained during the program execution by using processor PMU. To support its method, UPMLib uses the compiler to identify the memory pages and instrument the code region that needs to be monitored. Similar to Memphis, UPMLib does not handle the page migration of dynamically allocated data.

The work by Marathe et al. is somewhat similar to UPMLib. They use the information from processor PMU to place a page at a CPU node that has most access to the page. Unlike UPMLib, their method has two phases. The first is the profiling phase, that captures the page access frequency information. This phase requires the developer to identify a particular code region and instrument it with calls to their profiling library. The second phase is the profile-guided page placement phase, where the application is re-run and the pages from the captured information is first-touched in order to place them on a particular node.

The data placement method used by UPMLib and Marathe et al. have a limitation. The method is constrained by the page number and CPU node mapping. This constraint makes their method lack flexibility because the same analysis result can
not be used when the concurrency parameters (number of threads, cpu allocation) or the data size configuration is changed. The profiling step must be repeated, which may consume precious development time.

2.2.2 False sharing in shared memory multiprocessors

While the issue with ccNUMA platform is the remote memory accesses that can have significantly higher latency than the local memory accesses, false sharing problem is related with the data sharing among threads that can produce cache coherency misses. When a processor core modifies data that is currently shared by the other cores, the cache coherence mechanism has to invalidate all copies on the other cores. An attempt to read this data by another core shortly after the modification suffers a cache miss and has to wait for the most recent value in order to guarantee data consistency among cores.

The data sharing that occurs when processor cores actually access the same data element is called true sharing. Such accesses typically need to be coordinated in order to ensure the correctness of the program. A variety of synchronization techniques may be employed to do so, depending on the programming interface being used. Some of these techniques are locks, monitors, and semaphores. In the OpenMP API, critical and atomic are two constructs that prevent the code they are associated with, from being executed by multiple threads concurrently. The critical construct provides mutual exclusion for code blocks in a critical section, whereas the atomic construct ensures safe updating of shared variables. When the order of the accesses
is important, the OpenMP `barrier` and `taskwait` constructs can be used to enforce the necessary execution sequence.

In contrast to true sharing, false sharing is an unnecessary condition that can arise as a consequence of the cache coherence mechanism working at cache line granularity [7]. It does not imply that there is any error in the code. This condition may occur when multiple processor cores access different data elements that reside in the same cache line. A write operation to a data element in the cache line will invalidate all the data in all copies of the cache line stored on other cores. A successive read by another core will incur a cache miss, and it will need to fetch the entire cache line from either the main memory or the updating core’s private cache to make sure that it has the up-to-date version of the cache line. Poor scalability of multi-threaded programs can occur if the invalidation and subsequent read to the same cache line happen very frequently.

```c
int *local_count = (int*)malloc(sizeof(int)*NUM_THREADS*PADDING);
int *vector = (int*)malloc(sizeof(int)*VECTOR_SIZE);
for(i=0;i<NUM_THREADS;i++)
{
    #pragma omp parallel
    {
        int tid = omp_get_thread_num()*PADDING;
        if(tid < 0) tid = 0;

        #pragma omp for
        for(j = 0; j < VECTOR_SIZE; j++)
            local_count[tid] += vector[j]*2;

        #pragma omp master
        {
            int k;
            for(k = 0; k<NUM_THREADS; k++)
                result += local_count[k];
        }
    }
}
```

Figure 2.5: OpenMP code snippet with false sharing problem.
Figure 2.5 shows a code snippet from an OpenMP program that exhibits the false sharing problem. This code will read each value of a vector, multiply it by two, and calculate the sum. Its performance is inversely proportional to the number of threads as shown in Table 2.3.

Mitigating the false sharing effect can lead to an astonishing 57x performance improvement for this code. The reason for the poor performance of the unoptimized code lies in the way it accesses the local_count array. When the PADDING variable is set to 1, the size of the array is equal to the number of threads. The cache line size of the underlying machine being used is 128 bytes, so even though the threads access different elements of the array, they fetch the same cache line frequently and interfere with each other by causing unnecessary invalidations. By taking the cache line size into account and increasing the PADDING value, we can prevent threads from accessing the same cache lines continuously.

<table>
<thead>
<tr>
<th>Code version</th>
<th>Execution time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-thread</td>
</tr>
<tr>
<td>Unoptimized</td>
<td>0.503</td>
</tr>
<tr>
<td>Optimized</td>
<td>0.503</td>
</tr>
</tbody>
</table>

Table 2.3: Execution time of OpenMP code from Figure 2.5.

It is difficult for the developers to find which data that may cause significant bottleneck because of false sharing. Detecting false sharing accurately requires in-depth analysis on memory allocation and memory (read and write) operations from each thread. Previous works [14, 32, 30, 31] have developed approaches for memory analysis that use memory tracing and cache simulation. These starts by tracking the memory accesses (both loads and stores) at runtime. A cache simulator then takes
the captured data to analyze the sequence of each memory operation and determine the type and amount of cache misses generated during the simulation. The main drawback of this approach is within the memory tracing part, which can incur a very large overhead. A memory shadowing technique was used [47] in an attempt to reduce the overhead of tracking the changes to the data state. Intel PTU[18] utilizes the event-based sampling support on Intel processors to identify the data address and function that is likely to experience false sharing, but it does not provide information about the actual data objects that suffer greatly from false sharing.

2.3 OpenMP collector API

The OpenMP collector API forms the baseline of DARWIN that we use to manage the bottlenecks because of remote memory accesses and false sharing effects. This API was proposed as a standard means of enabling performance tools to interact with OpenMP implementations [20]. The collector API provides event-based interface to facilitate the communication between the OpenMP runtime library and performance tools in a manner that is transparent to the developer. The API has been implemented in the OpenMP runtime library of OpenUH compiler suite by instrumenting particular runtime routines with calls to OpenMP collector event handlers [6, 15]. These handlers are provided by the performance tool that utilizes the OpenMP collector API. A collector event is associated to a particular state within the OpenMP program execution. Each time a thread reaches a state, the runtime library will throw a notification to the performance tool by calling the provided event handler.
Table 2.4 gives examples of collector events and the states to which they are associated with. The OpenMP master thread triggers the fork event when it enters a parallel region and the join event as soon as it leaves the implicit barrier at the end of the parallel region. In the OpenUH implementation, the slave threads are created during the runtime library initialization and remained alive between parallel regions. Therefore, each of the slave threads triggers the end idle event after it is woken up by the master thread to execute the parallel region and the begin idle event when it leaves the parallel region. The OpenUH runtime distinguishes barrier events for the implicit and explicit barrier through ibar and ebar events.

<table>
<thead>
<tr>
<th>Collector event</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_EVENT_FORK</td>
<td>Master thread enters a parallel region</td>
</tr>
<tr>
<td>OMP_EVENT_JOIN</td>
<td>Master thread leaves a parallel region</td>
</tr>
<tr>
<td>OMP_EVENT_THR_END_IDLE</td>
<td>Slave thread executes a parallel region</td>
</tr>
<tr>
<td>OMP_EVENT_THR_BEGIN_IDLE</td>
<td>Slave thread leaves a parallel region</td>
</tr>
<tr>
<td>OMP_EVENT_THR_BEGIN_IBAR</td>
<td>Master or slave thread enters an implicit barrier</td>
</tr>
<tr>
<td>OMP_EVENT_THR_BEGIN_EBAR</td>
<td>Master or slave thread enters an explicit barrier</td>
</tr>
</tbody>
</table>

Table 2.4: OpenMP collector events
Chapter 3

Dynamic Optimization Framework for OpenMP

As discussed in Section 2.2, deep understanding about an applications dynamic behavior such as data locality, data sharing among threads, and memory latency is needed to get optimal performance. The compiler may face difficulty in generating optimal code due to lack of this information. Another case is when the developer might not know about the characteristic of the target platform. A performance tool, clearly, can fill the gap and help the developers, compilers, or runtime systems deliver a better code.

A performance tool usually starts with collecting performance data and analyzing it to find symptoms of bottleneck that may occur in a program. It continues by performing optimization based on the analysis result. Previous works like COBRA [24], TRIDENT [46], DynamoRIO [3], UPMLib [36], and Autopin [26] focus on
online optimization which perform the data collection, analysis, and optimization concurrently with program execution. The work by Marathe et al. [29], Huck et al. [16], and Memphis [33] are focusing on the offline approach.

To communicate with a program, different tools utilize different techniques. COBRA, TRIDENT, and DynamoRIO depend on binary instrumentation to hook their framework with an application. Huck et al. and UPMLib are using the compiler to identify and instrument the code region that needs to be optimized. Marathe et al. and Memphis rely on user interaction to instrument the code with calls to the functions provided by their tool.

The performance tools also captures various kind of performance related information to support a specific optimization. TRIDENT relies on a special hardware to notify about hot path and hot value required for performing code layout optimization and dynamic value specialization. DynamoRIO takes the binary instruction stream as input to guide object code optimization. COBRA uses information from the hardware performance monitoring unit to determine the data that suffers from frequent cache coherency miss and optimizes the data prefetching by exchanging the `prefetch` instruction with other instruction, such as `nop`. Autopin utilizes information about the amount of retired instruction and the number of instruction cycles to determine the optimal concurrency configuration, such as number of threads and thread affinity. Huck uses hardware performance monitoring unit and execution timing information to identify the performance characteristics of an application. Marathe et al., UPMLib, and Memphis utilize memory access information from the hardware performance monitoring unit to guide the ccNUMA data placement optimization.
This chapter discusses our dynamic optimization framework, called DARWIN (Dynamic Adaptive Runtime Infrastructure) [35, 44]. DARWIN is based on the open-source OpenUH [28] compiler. The core feature of this framework is its usage of the OpenMP collector API, discussed in Section 2.3, to interact transparently with a running OpenMP program. This interaction is transparent to the application developers because no changes to the code are necessary. DARWIN also utilizes the hardware performance monitoring unit to obtain detailed information about the program’s execution. When combined with its other capabilities, such as relating the performance data to the data structures in the source code, DARWIN is able to help the application developers to gain insight about dynamic code behavior, particularly in the hot-spots of an OpenMP program.

3.1 DARWIN - Initial design

Figure 3.1 gives the overview of the original structure of DARWIN, proposed in [35]. It utilizes the OpenMP collector API and other open source libraries to gather performance related information during the runtime execution. After the application completes its execution, DARWIN analyzes the gathered information and passes it to the compiler as feedback for optimization.

3.1.1 Framework structure

More details about the collector API and other open source libraries are as follows:
• OpenMP collector API
The collector API acts as the main driver of DARWIN and coordinates the performance data collection process. Callback handlers to particular collector events are used to determine when to start or stop the data collection.

• PAPI
PAPI [27] is an interface that provides functions for accessing hardware performance monitoring units (PMU)[21, 12] available in modern processors. DARWIN utilizes PAPI to count the occurrence of micro-architectural events generated by the PMU to understand the performance characteristics of the application. For example the number of cache misses, TLB misses etc.

• LIBPFM
Similar to PAPI, libpfm [11] is also used to capture information from the PMU. The difference is libpfm can access processor specific features like the Data Event Address Register (DEAR)[17] of Itanium 2, Instruction-based Sampling
(IBS)[10] of Opteron, and Precise Event-based Sampling (PEBS)[19] to capture finer information, like the data addresses related to a PMU event.

- **LIBPSX** and **LIBELF**
  DARWIN utilizes *libpsx* [5] to traverse the program’s call stack and capture the instruction pointer of a particular code statement during the program execution. The library also maps a given instruction pointer to the corresponding source file name, function name, and line number. By using this information, DARWIN is able to distinguish unique parallel regions and relate them to the source code.

  *Libelf* [34] can be used to extract information of static and global variables declared in the program. By using this library, DARWIN can form a mapping between memory address and the corresponding variable’s information, such as starting address, size, and variable name.

- **PIN**
  PIN [23] is a tool for binary instrumentation that can capture specific instruction issued by the application, such as the data read/write, and prefetch instructions.

### 3.1.2 Design limitation

The initial design of DARWIN lacks of modularity. DARWIN only consists of one module, which is the *Collector Based Dynamic Optimizer*. This monolithic design can reduce DARWIN maintainability and reusability. Developers may have difficulties
to work on a particular part of the framework because they need to know about
the system as a whole. They also may not be able to reuse a particular part of the
framework without having a lot of changes to the code. The framework needs to be
broken down into several smaller modules, such that each module has a particular
concern and there are logical boundaries between the modules.

There are some improvements that can be made to DARWIN in terms of its
functionality. During DARWIN development, we found that libpfm and PAPI have
redundant functionality for accessing the processor PMU. PAPI is actually a higher
level interface built on top of libpfm. Zaparanuks et al.[45] stated that using lower
level API (libpfm) to access the processor PMU can improve the result accuracy.
The lower level API can also capture more detailed information from the processor
specific features like the DEAR of Itanium 2 and IBS of Opteron. With the advan-
tages of libpfm over PAPI, accessing the processor PMU directly through libpfm is
considerably better than through PAPI.

A feature of associating the gathered information to particular data structures
in the source code is important to overcome memory bottlenecks. It can signifi-
cantly help developers to pinpoint the variable in the code that is responsible for the
bottleneck and needs to be optimized. However, the initial design is limited to the
information of global and static data. It does not cover the dynamically allocated
data. Another feature that can make DARWIN more appealing is the ability to per-
form optimization during the program execution. Additionally, supporting an online
optimization feature can establish DARWIN as a complete dynamic optimization
framework.
3.2 DARWIN enhancement

We have enhanced the design of DARWIN by making it modular, such that each module has a distinct functionality and abstraction to a particular supporting library. This has resulted in an overall framework that is easier to use and reuse, and also more flexible in the sense that the addition of new functionality is now straightforward.

Figure 3.2: Modules of DARWIN

Figure 3.2 illustrates the main modules of the enhanced DARWIN framework. It includes modules for catching OpenMP collector event notifications, performance monitoring, capturing data allocations, data management, optimization, and utilities for supporting performance data analysis. The details about each module are discussed in the following subsections.
3.2.1 Collector tool

The collector tool is the central part of the DARWIN framework since it coordinates the profiling and optimization activity. This component utilizes the OpenMP collector API to communicate with the OpenMP runtime and thus gain insight about a program's execution. In a common scenario, we configure the collector tool to catch the events associated with the beginning and the end of an OpenMP parallel region. The parallel region, most of the time, is the basis for finding the hot spots in an OpenMP program.

The collector tool module is implemented as a C++ class named \texttt{CCollectorTool} as depicted by Figure 3.3. The class' constructor initializes the collector API by sending a request to start tracking the OpenMP states. The OpenMP runtime only gives notification of a registered collector event, so a \texttt{register\_event} routine is
provided to the developer as an abstraction to register a collector event and its callback handler. The \textit{init\_lib} and \textit{add\_events} are virtual functions that need to be implemented by the developer for the purpose of initializing particular user libraries and determining the required collector events, respectively. The \textit{start} routine triggers the OpenMP runtime to start the notification of the registered collector events.

```cpp
class CBarrierCollector :
public CCollectorTool
{
 protected:
  virtual void add_events();
  virtual void init_lib();
  static void callback(
    OMP_COLLECTORAPI\_EVENT event);

 public:
  CBarrierCollector() : CCollectorTool()
  {
    start();
  }  //CBarrierCollector();
  static CBarrierCollector instance;
  CBarrierCollector CBarrierCollector::instance;

  void CBarrierCollector::add_events()
  {
    this->register_event(OMP\_EVENT\_THR\_BEGIN\_IBAR,
      &CBarrierCollector::callback);
    this->register_event(OMP\_EVENT\_THR\_END\_IBAR,
      &CBarrierCollector::callback);
  }

  void CBarrierCollector::callback(
    OMP\_COLLECTORAPI\_EVENT event)
  {
    switch(event)
    {
      case OMP\_EVENT\_THR\_BEGIN\_IBAR:
        // capture the starting timestamp
      case OMP\_EVENT\_THR\_END\_IBAR:
        // capture the finishing timestamp
        // calculate the difference
    }
  }

  CBarrierCollector::~CBarrierCollector()
  {
    // print the measurement result
  }
```

(a) \textit{CBarrierTool} class definition  

(b) \textit{CBarrierTool} class implementation

Figure 3.4: A performance tool for measuring execution time of each thread in OpenMP implicit barrier.

The \textit{CCollectorTool} class is implemented as an abstract class that needs to be extended by the developer in order to create a particular performance tool. Figure 3.4 gives an implementation of a performance tool that extends the \textit{CCollectorTool} class in order to measure the time spent by each thread in an implicit barrier. The
add_events routine registers the events indicating the beginning and the end of an implicit barrier. In the callback handler of those events, the performance tool records the starting and finishing timestamp when a thread enters and leaves an implicit barrier, respectfully. It then calculates the timestamp difference to determine the execution time of a thread inside an implicit barrier. A static object of CBarrierTool is created to start the event-based communication right after the OpenMP program starts, which is indicated by the call to start routine in the class constructor. The destructor of this class is used to show the measurement result just before the program execution ends.

3.2.2 Performance monitoring

The performance monitoring module encapsulates the functions and data structures to access the processor-specific performance monitoring unit (PMU). In the work we have performed here, an appropriate PMU is used to pinpoint the data structure causing the performance problem. The DARWIN framework has been implemented on the Itanium 2 platform that provides the Data Event Address Register (DEAR) suitable for this purpose. DEAR can track memory load instructions and capture the information of any resulting data cache misses. This information includes the instruction and data addresses, as well as the memory access latency. We can use this information to help programmers optimize the memory behavior of the program, as described further in Chapter 4 and 5.
The performance monitoring module is implemented as a C++ class named `CPerformanceMonitoring`. This class encapsulates the data structure and routines used to capture information from the DEAR PMU. An instance of this class needs to be created for every OpenMP thread in the application in order to provide per-thread monitoring. The `CPerformanceMonitoring` class definition and implementation is shown by Figure 3.5. The class constructor performs necessary initialization, such as acquiring the process id and OpenMP thread id that is going to be monitored. Both are required as the identifier of the `CPerformanceMonitoring` instance and for setting the PMU. Four main routines are provided to setup, start and stop the DEAR PMU, and to retrieve the monitoring result. These routines utilize the `libpfm` library to access the PMU functionality.

In the setup function, the CPU registers that control the PMU are set with a particular hardware event type and sampling period configuration. We use the hardware event with the name in the format of `data_ear_cache_lat[threshold]` to capture information about data cache misses with a particular latency limit. The threshold part in the end of the event's name represents the lowest latency (in cycle) of a cache miss that will be captured by DEAR. For example, setting the PMU register with `data_ear_cache_lat128` means the PMU only captures the data cache misses with access latency of 128 cycles or more. The sampling period configuration represents the rate of cache misses that will be captured. Its value contains the number of cache misses to be ignored before capturing one. More information about DEAR can be found in chapter 10 of the Intel Itanium 2 reference manual [17]. Application developers can use the environment variables `EVENT_NAME` and `SAMPLING_PERIOD`
class CPerformanceMonitoring
{
private:
...int omp_thread_id;
  pid_t linux_thread_pid;
  void *smpl_vaddr;
  vector<SSamplingResult> v_result;
...

public:
  CPerformanceMonitoring(int omp_tid);
  ~CPerformanceMonitoring();
  void setup_pmu();
  void start_sampling(int regid);
  void stop_sampling();
  void process_smpl_buffer();
  vector<SSamplingResult>* get_result();
...};

CPerformanceMonitoring::CPerformanceMonitoring(
    int omp_tid)
{
    omp_thread_id = omp_tid;
    linux_thread_pid = (pid_t)syscall(SYS_gettid);
    ...
}

void CPerformanceMonitoring::setup()
{
    //setup pmu control and data registers
    //setup sample buffer
    //setup overflow handler
    //associate pmu context with the thread pid
}

void CPerformanceMonitoring::start(int regid)
{
    region_id = regid;
    setup();
    pfm_self_start(...);
}

void CPerformanceMonitoring::stop()
{
    pfm_self_stop(...);
    process_smpl_buffer();
}

void CPerformanceMonitoring::process_smpl_buffer()
{
    //get the pointer to kernel buffer
dear_hdr_t *hdr = (dear_hdr_t *)smpl_vaddr;

    int sample_count = hdr->hdr_count;
    for(i=0; i < sample_count; i++)
    {
        SSamplingResult sample;
        //capture information: cpu, data_address,
        //inst_address, access_latency
        v_result.push_back(sample);
    }
}

Figure 3.5: Implementation of the performance monitoring module.
to set the value of each configuration.

After the PMU registers preparation is finished, the setup process will continue to set the size of a kernel buffer that will store the monitoring result. This buffer is a temporary storage and will generate an overflow when it is full. Therefore, we also initialize an overflow handler that will call the `process_smpl_buffer` routine. This routine is used to consume all of the sampling results inside the buffer and store them to a vector data structure. Figure 3.6 shows the structure to hold each sample. Finally, the setup process is ended by associating the PMU registers with the process id of the monitored thread.

```c
struct SSamplingResult {
    int region_id;
    unsigned long data_address;
    unsigned long instruction_address;
    int latency;
    int cpu;
};
```

Figure 3.6: C structure holding the monitoring result.

The starting and stopping process is very straightforward. To start the PMU we first perform the setup process and then call the `pfm_self_start` routine from `libpfm`. Stopping the PMU can be done by just calling the `pfm_self_stop` routine and calling the `process_smpl_buffer` to save the result.

### 3.2.3 Data allocation

The data allocation module is used to capture the allocation information of global, static, and dynamic data. The captured information, as depicted by Figure 3.7,
includes the starting and ending virtual memory address, parallel region id where the allocation takes place, thread id that performs the allocation, allocation size, a flag that distinguishes dynamic and static allocation, and an identifier. For global and static data, this module uses the variable name as the identifier. For dynamic data, it uses the function name that calls the memory allocation routine and the line number in the source code as the identifier. The allocation information is stored into a B-Tree [2] that offers very good performance for searching, especially when the number of allocation records, and hence the search space, is very large.

```c
typedef struct {
    unsigned long lb_address;
    unsigned long ub_address;
    short regid;
    short threadid;
    int size_alloc;
    short dynamic;
    char name[34];
} __attribute__((packed)) data_alloc_t;
```

Figure 3.7: Data allocation information.

The data allocation module utilizes libelf library to access an application’s symbol table, which contains the starting virtual memory address, allocation size, and variable name of global and static data. The ending address for this data is calculated by summing the starting address with its allocation size. The parallel region id, thread id, and the dynamic flag of the static and global data structure is set to 0 because this data is allocated statically during the compile time.

To retrieve information about a dynamic data allocation, at first, we used the `malloc hooks` [13] from the GNU C library. With these hooks, we can specify a hook routine that will be called by the C-runtime whenever a `malloc` is called. When
called, the hook routine receives two input parameters, which are the starting address
of the allocation, and its size. This information can be recorded, and then the real
allocation routine is called to actually perform the dynamic data allocation. As the
call to the real allocation routine will also trigger the hook routine, it is necessary
to protect it from a never ending recursive calls to the hook routine. The protection
can be done by simply disabling and enabling the hook just before and after the real
allocation. However this protection has a drawback when multiple threads perform
dynamic data allocation in the same time. It is possible that not all allocations
trigger the hook routine when the hook is disabled. One way to solve this issue is to
perform the dynamic data allocation in serial manner. Unfortunately, this is not a
good solution because the developer has to change their code.

We decided to use another solution by interposing the dynamic memory allocation
routine. This is done by overloading the allocation routine. Within the overloading
routine, we can call the real allocation routine without any protection. The actual
memory allocation is performed by first getting the address of the real allocation
routine by using `dlsym` [40]. Then a call to the function pointer of the real routine
is made. Figure 3.8 gives an example of how the interposition is performed. The
pointer `real_malloc` holds the function pointer to the actual `malloc` routine. A call
to `real_malloc` is made in the end of the overloading routine in order to perform the
actual allocation.

Unlike the application’s symbol table, the dynamic memory allocation routine
does not provide any variable name. To get over with this issue, we use the function
name that calls the dynamic allocation routine and the line number in the source

34
```c
static void* (*real_malloc)(size_t) = NULL;
void* malloc(size_t size)
{
    if (real_malloc == NULL)
        real_malloc = dladdr(RTLD_NEXT, "malloc");

    // get the program counter of the caller
    caddr_t pc[10]; // array of program counter
    int ret = psx_rtc_cs(..., pc, ...);
    ps_ip_map_t mymap[MAXSTACK];
    // map the program counter to the debugging info
    psx_bfd_map(bfd, pc, 1, mymap);
    // record the caller name and line number
    sprintf(temp_name, "%s_%d", mymap[0].function, mymap[0].line);

    intmap_t alloc_data;
    alloc_data.regid = parallel_region_id;
    alloc_data.threadid = omp_get_thread_num();
    alloc_data.size_alloc = size;
    alloc_data.dynamic = 1;
    alloc_data.lb_address = result;
    alloc_data.ub_address = (result +size-1);
    strcpy(alloc_data.name, temp_name);
    // store the allocation info to btree
    kb_put_darwin(..., alloc_data);
    // perform actual allocation
    void* p = real_malloc(size);
    return p;
}
```

Figure 3.8: Capturing dynamic data allocation by interposing the `malloc` routine.

code as the variable name. We utilize the `libpsx` library to traverse the call stack and capture the program counter of the allocation routine caller. Then we use the debugging information to map the program counter its corresponding caller’s name and line number. It is possible that more than one dynamic data allocations are attributed with the same variable name, which may indicate that these allocations are elements of an array, list, or tree.
### 3.2.4 Data management

The data management module is responsible for associating the captured performance data to the appropriate data structure in the source code and thus for producing data-centric[4] information that is comprehensible by the application developer.

```cpp
class CDataManager
{
    void create_data_centric(int count,
                               CPerformanceMonitoring* mon_module[]);
    void safe_data_allocation()
    void insert_ref_to_db(int regid, int threadid,
                           int cpuid, long pg_nmbr, long data_addr,
                           unsigned long instr_addr, int latency,
                           char* name);
    void insert_alloc_to_db(int regid, int threadid,
                             unsigned long lb_address,
                             unsigned long ub_address, size_t size,
                             char* name, int dynamic, int pagesize);
    vector<SMemRefAggrInfo_t> aggrPageRef(int regid,
                                          char* dbname);
    vector<SMemRefAggrInfo_t> aggrPageRefPerSymThread(  
                                                int regid, int threadid,  
                                                const char* name, 
                                                char* dbname);
    ...
}
```

Figure 3.9: `CDataManager` class definition.

Class `CDataManager`, as depicted by Figure 3.9, is used to implement this module. The `create_data_centric` routine is provided in order to read the data cache miss samples from each of the `CPerformanceMonitoring` instances. It searches the B-Tree of the data allocation module to find the data structure with address range covering the data address of each cache miss sample. If the search operation is successful, we associate the sample with the data structure identifier. Otherwise, we associate the sample with a default identifier named "NULL", which means an unknown data structure. Finally each of the associated sample is stored into a flat binary file.

Table 3.1 shows the examples of a generated data-centric information. It lists the
page addresses accessed by thread id '0' and '1' and its latency at the fourth parallel region of an OpenMP application.

<table>
<thead>
<tr>
<th>Parallel region id</th>
<th>Thread id</th>
<th>Page address</th>
<th>Latency</th>
<th>Variable name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0x600000000000087f80</td>
<td>223</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0x600000000000087f80</td>
<td>184</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0x600000000000087f80</td>
<td>176</td>
<td>p</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x600000000012db500</td>
<td>223</td>
<td>colidx</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x600000000012db500</td>
<td>223</td>
<td>colidx</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x6000000001357f80</td>
<td>223</td>
<td>colidx</td>
</tr>
</tbody>
</table>

Table 3.1: Data centric information captured from CG of NPB.

Other than generating data-centric information, the CDataManager class also provides the following important routines.

- **void safe_data_allocation()**

  This routine traverses the B-Tree of the data allocation module and store the content of each tree node to a flat binary file.

- **void insert_ref_to_db(...)** and **void insert_alloc_to_db(...)**

  Both of these routines are used to load the generated data-centric information and the data structure allocation detail to an SQLite[39] database. SQLite is a stand-alone portable database that does not need network connection. Its support for the SQL language offers a convenient way to access and analyze the information collected by DARWIN.

- **vector<SMemRefAggrInfo_t> aggrPageRef(...)** and **vector<SMemRefAggrInfo_t> aggrPageRefPerSymThread(...)**
Both of these routines are used to aggregate the data-centric information based on a particular criteria. The aggrPageRef routine groups the cache miss samples of each parallel region by thread id and data address. The aggrPageRefPerSymThread routine performs similar aggregation operation, but it also adds the variable name to the grouping criteria. Table 3.2 shows the example of aggregation results of the data-centric information in Table 3.1. In this example, the aggregation operation accumulates the latency of the accesses with the same region id, thread id, page address, and variable name.

<table>
<thead>
<tr>
<th>Parallel region id</th>
<th>Thread id</th>
<th>Page address</th>
<th>Sum of latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0x6000000000087f80</td>
<td>583</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x60000000012db500</td>
<td>456</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x6000000001357f80</td>
<td>126</td>
</tr>
</tbody>
</table>

(a) Aggregation result of the aggrPageRef routine.

<table>
<thead>
<tr>
<th>Parallel region id</th>
<th>Thread id</th>
<th>Page address</th>
<th>Sum of latency (cycles)</th>
<th>Var. name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0x6000000000087f80</td>
<td>407</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0x6000000000087f80</td>
<td>176</td>
<td>p</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x60000000012db500</td>
<td>456</td>
<td>colidx</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x6000000001357f80</td>
<td>129</td>
<td>colidx</td>
</tr>
</tbody>
</table>

(b) Aggregation result of the aggrPageRefPerSymThread routine.

Table 3.2: Result of aggregating the data centric information.

### 3.2.5 Optimizer

The optimizer module provides an implementation of several optimization strategies. Two kinds of optimization strategies, high level and low level, are distinguished. The high level optimizations essentially utilize OS routines, such as those for setting
thread affinity, memory page migration, or make calls to other library routines; e.g. for modifying the number of threads, adjusting core frequency, or accessing a specialized malloc library. The low level optimizations are applied by transforming the source code or modifying instructions in the binary.

```cpp
class CPlacementOptimizer
{
private:
    int master_thread_pid, original_cpu_id, num_threads, page_size;
    map<int, int>* map_thread_cpu;
    map<int, vector<unsigned long>> map_thread_pages;
    CNumaUtil* cnuma_util;
public:
    CDataPlacer(int pid, int cpu, int n_threads, CNumaUtil* numaUtil,
                map<int, int>* m_thread_cpu);
    ~CDataPlacer();
    int distribute_data(int first_touch);
    /*
     * num_cycle : number of cycles
     * block_size : size of each chunk in a cycle
     * stride : size of a cycle
     * direction : 0 (data is distributed starting from the lowest to highest address)
                 1 (data is distributed starting from the highest to lowest address)
     */
    void insert_data_block_dist(int first_touch, int block_size, int direction,
                                 unsigned long page_address);
    void insert_data_cyclic_dist(int first_touch, int num_cycle, int block_size,
                                int stride, int direction,
                                unsigned long page_address);
};
```

Figure 3.10: CPlacementOptimizer class definition.

CPlacementOptimizer class, as depicted in Figure 3.10, is implemented to enable high level optimization strategy for distributing data on a ccNUMA platform. This class encapsulates the following data structure and routines.

- `map<int, int>* map_thread_cpu`

This is a map data structure that contains the mapping of thread id and its
corresponding CPU id.

- $\text{map<int, vector<unsigned long>} > \text{ map_thread_pages}$
  This is a map data structure that contains the mapping of thread id and collection of data page address.

- $\text{void insert_data_block(...)}$ and $\text{void insert_data_cyclic(...)}$
  Both routines break a given page address range into chunks and map them to OpenMP threads based on block or cyclic pattern. The result of this mapping process is stored into the $\text{map_thread_pages}$ to be used by the $\text{distribute_data}$ routine as input for the data placement optimization. More details about block and cyclic access pattern is discussed in Chapter 4.

- $\text{int distribute_data(...)}$
  This routine can perform two kinds of data placement optimization methods, which are the first-touch and next-touch methods. The first-touch method is appropriate for distributing a data structure that has not been initialized beforehand. To apply this method, the routine $\text{dist_data()}$ first sets the affinity of the master thread to a destination CPU, whose id is retrieved from a map data structure $\text{map_thread_cpu}$. Then the routine will perform a write operation to each of the pages that needs to be placed on the destination CPU. The thread affinity of the master thread is restored to its original CPU after the first touch optimization is finished.

  The next-touch method is useful for moving pages of data from one CPU’s local memory to another. The optimizer uses the $\text{move_pages}$ routine from
libnuma library to perform the next-touch method. This routine requires two data arrays containing a list of the page numbers that are going to be moved and a list of CPU id that will be the destination of each page of the first array.

3.2.6 Utilities

The utilities module provides tools for offline data analysis. One tool, called *darwin_exporter*, utilizes the data manager module of DARWIN to export the collected data in the flat binary file into an SQLite database file. There are two tables in this database, which are the *data_allocation* and *data_reference* table. The first table stores the recorded data allocation information and the second table stores the collected performance data from the PMU. *Darwin_exporter* also exports DARWIN collected information into a particular format by reading the contents of the SQLite database file, aggregate them, and write the result into text files. These text files are used further as input of a visualization tool so that the developer can perform analysis. Currently, the format that we use is the TAU [38] profile format.

Each of the resulting text files of the *darwin_exporter* is named with the following format: *profile.0.0.[id]*. The *id* in the last part of the file name contains the OpenMP thread id. The text file contains rows of information about the accesses to particular memory addresses; each row is filled with the referenced memory address, sum of access latency, 0, count of memory reference, average access latency, 0, 0, data structure identifier. Every column in each row is separated with a single whitespace. Figure 3.11 shows an example of a TAU profile generated by our tool.
TAU is a portable toolkit for performance analysis of parallel programs that are written in Fortran, C, C++, and Python. The programmer can run TAU’s Paraprof [41] to observe the behavior of the program through its 3D visualization capabilities, and draw conclusions about the observed performance bottlenecks. Figure 3.12 gives an example of visualization by Paraprof. The horizontal, depth, and vertical axis gives the memory address, thread id, and reference count, respectively.

A second tool, called darwin_alloc_reader, is created to help developers analyze the data allocation information. This is a simple console application that accepts a cache line address as input and search for any data structure that has element residing in that cache line. It basically utilizes DARWIN’s data_manager module to query
the data_allocation table in the SQLite database file generated by darwin_exporter.

A third tool, called darwin_import_analysis_data is used to insert analysis result into another SQLite database file, which is used as input for DARWIN’s optimizer module. The database file contains two tables, which are the data_access_pattern and optimization_decision table. The first table represents the access pattern of a data structure. It has the following five columns:

- **Id**
  This is a unique id for each row in the table.

- **Data structure identifier**
  This column contains the variable name.

- **Type**
  This column can contain either '0' or '1'. '0' means the data structure in this row has block access pattern type, while '1' means cyclic access pattern type.

- **Size**
  This is the size of the data structure that follows the access pattern type of the third column. The value of this column is in percentage, where 100 means the whole area of a data structure follows a particular access pattern type.

- **Number of cycles**
  This is applied for data structure that has cyclic access pattern type.

The optimization_decision table represents the data placement strategy that need to be applied on particular data structures. It has the following four columns
Id
This column points to one of the id in the data_access_pattern table. It indicates the data structure that needs to be optimized.

Method
This column can contain either 'first_touch' or 'next_touch'. Each of them determines the data placement method that will be applied by the optimizer module.

Location 1
This column determines place and time during the execution to perform the optimization. It can contain either 'init' or 'parallel_region'. The first value orders the optimization to be done during DARWIN initialization, see the discussion of init_library routine of CCollectorTool class in Section 3.2.1. The second value means that the optimization will be done just before a parallel region.

Location 2
This column indicates which parallel region to perform the optimization.

3.3 Workflow of DARWIN

DARWIN is currently used as a feedback dynamic optimization system that has two execution phases, the monitoring phase and the following optimization phase. In the monitoring phase, the collector tool module starts and stops the performance
monitoring module when it receives a collector event notification about the start and end of an OpenMP parallel region, respectively. As discussed in Section 3.2.2, the performance monitoring module is used to capture samples of load memory operations. At the end of the monitoring phase, the collector tool invokes the data manager module to create data-centric information and store all of the captured information into flat binary files.

Developers can analyze the collected information by using TAU Paraprof visualization tool. Since the monitoring phase only gives a flat binary files, the developers need to use the provided utilities to export the flat binary files into TAU compatible format and also to store the analysis result into an SQLite database. Chapter 4 and 5 give further discussion of how we use the data visualization to find the data structures that cause memory bottleneck in the program.

DARWIN’s optimization phase is performed during the program’s subsequent
runs. The collector tool utilizes the data manager module to retrieve the optimization parameters from an SQLite database file. The collector tool then invokes the optimizer module to apply the appropriate optimization strategy based on its parameter.
Chapter 4

Data Placement Optimization

The placement in memory of the pages holding a program data can have a major impact on the performance of an application. The effects of data placement are more evident on ccNUMA platforms, which now prevail, than those with symmetric memory access. On such platforms, each processor can directly access the local main memory, but has to use the system interconnect to access the memory banks of the other processors (remote memory). Remote memory accesses become a major bottleneck if the data is not carefully placed. The traditional first-touch policy implemented within the operating system is sometimes very effective, but can also lead to an inefficient page placement, especially if the programmer is not aware of this policy. With knowledge of the memory access patterns, one can devise an efficient memory placement strategy and reduce the number of remote memory accesses.
4.1 Methodology

To deal with the data placement problem on a ccNUMA platform, we use DARWIN’s monitoring phase to collect information of memory references for each thread and present them to the application developers via data visualization. This visualization enables the developers to identify the data mappings and access patterns type of a variable, which further enable the developers to find variables that have data locality problem. Section 4.1.1 and 4.1.2 show the steps of identifying variable access pattern and data placement problem on a particular variable.

The knowledge of data access pattern enables DARWIN’s optimization phase to properly calculate and map the pages storing a variable to the destination memory nodes. Based on the page-destination memory node mapping, DARWIN’s optimizer module performs the necessary optimization by using the first-touch method for initial placement and next-touch method on dynamic data structures or data structures that have multiple access pattern types across different parallel regions.

Our methodology is flexible in the sense that it is not sensitive to the runtime configuration, such as the data size and the number of threads. We do not need to repeat the monitoring phase even if the runtime configuration is changed because the access pattern of a variable remains constant. This is very useful for monitoring the application on a reduced data set and subsequently optimizing the application in the production environment, which typically will require a larger data size and more threads.
4.1.1 Data access pattern type identification

As discussed in Section 3.2.6, the information of per thread memory references in each parallel regions can be exported into TAU profiles by the `darwin_exporter` tool. One type of information provided by the TAU profiles is the access count to a particular page address. This information can be used by the developers to classify the access pattern type of each data structure in the application. Our work currently focuses on block and cyclic patterns. For the block type, the pages of a variable is divided into contiguous, equal-sized chunks of size $N/P$. $N$ is the number of pages of a variable and $P$ is the number of accessing threads. Each chunk is accessed by only one thread, with the order according to the thread ID. For the cyclic type, the pages of a variable is first divided into a number of cycles. Just like the block type, each cycle contains the same number of chunks that will be accessed by the threads in a round robin fashion.

Figure 4.1 gives the examples of visualization of the variables with the block and cyclic access pattern. The vertical axis gives the total amount of references on each page.

Figure 4.1: Data access pattern type: (a) Block. (b) Cyclic with 5 cycles.
page. The horizontal axis contains the page numbers of the variable. The depth axis provides the thread ID that accesses the variable. Figure 4.1 (a) shows a data structure with a block access pattern type that is accessed by four threads. Figure 4.1 (b) shows a data structure with a cyclic pattern that is also accessed by four threads in five cycles.

4.1.2 Data locality problem detection

The generated TAU profiles also has information about the latency of the memory access to a particular page address. The developers can use this information to find which data structures that have a data locality problem and is worthwhile optimizing the accesses. A ccNUMA aware data structure usually has balanced latency distribution that makes data placement optimization attempt may not be effective. Therefore, this kind of data structure does not need to be optimized by this framework.

Figure 4.2: (a) A data structure with latency imbalance issue. (b) A data structure that is already optimized for ccNUMA platform.
Figure 4.2 shows the memory access latency visualization. The vertical axis provides the average latency of the accesses to a page. The horizontal axis contains the page number starting from the left. The depth axis shows the thread ID. The data structure shown by Figure 4.2 (a) has a latency imbalance issue, where the accesses by thread ID 2 and 3 have higher latency than the accesses by thread ID 0 and 1. The data structure in Figure 4.2 (b) is already optimized for ccNUMA platform because the pages that follow the cyclic access pattern have relatively the same latency.

4.2 Experimental results

We tested our data placement optimization strategy with seven programs from the OpenMP C version of the NPB-2.3 benchmark: CG, BT, MT, FT, IS, LU, and MG. All programs were compiled with the OpenUH compiler using optimization level O2 and the class A and B data set. They were run on an SGI Altix 3700 consisting of 32 nodes with dual 1.3 GHz Intel Itanium2 processors per node running the SUSE 10 operating system. The experiments were conducted using an interactive PBS\textsuperscript{1} with two compute nodes, four physical CPUs and four OpenMP threads.

4.2.1 Monitoring phase results

For each program, Table 4.1 gives the number of parallel regions and data structure allocations. Almost all of them rely on static or globally declared data; only the

\textsuperscript{1}Portable Batch Session
MG benchmark performs dynamic data allocation. Further results show that a large amount of dynamic data allocation can produce a significant overheads.

To determine overheads, we compare the execution time of the monitoring phase with the baseline program execution time. Figure 4.3(a) shows the slowdown of the monitoring phase, defined as the monitoring execution time divided by the original program execution time. The monitoring overhead consisted of the time taken to capture data allocation, collect the data cache miss samples, and create data-centric information. Each of them was measured using the gettimeofday routine. Figure 4.3(b) gives the breakdown of the monitoring phase overhead.  

According to Figure 4.3(a) the monitoring phase generated a slowdown ranging from 1.01x to 1.37x, with the average around 1.10x. MG had the highest overhead with 1.37x slowdown, while the rest of the programs had less than 1.1x slowdown.

<table>
<thead>
<tr>
<th>Metric</th>
<th>CG</th>
<th>SP</th>
<th>FT</th>
<th>BT</th>
<th>LU</th>
<th>MG</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td># of parallel regions (counts)</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td># of static data allocations (counts)</td>
<td>41</td>
<td>126</td>
<td>39</td>
<td>128</td>
<td>76</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td># of dynamic data allocations (counts)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>246797</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Captured parallel regions and data allocation.
Figure 4.3(b) shows that majority of the overhead of most programs was coming from generating data-centric information, where every cache miss sample is associated with the corresponding data structure and then stored into a persistent storage. Unlike the other programs, the majority of MG’s overhead was the time required to capture the data allocation information. According to Table 4.1, MG had 246,797 of dynamic data allocations. This is a very large number compared to the other programs that had zero dynamic data allocations. Tracking the dynamic data allocation can be an expensive operation, especially when the program has a large number of dynamic allocations. Based on our investigation, traversing the stack frame during the dynamic data allocation can consume a significant amount of time.

As a result of our observation on the memory access visualization, Table 4.2 lists each data identifier that was profitable for optimization, its access pattern type, and the program execution state to apply the optimization. For LU, we can not find any data structure that is considered as profitable for data placement optimization. The data structure in MG is dynamically allocated, so the DARWIN framework uses the function name that calls malloc and allocation line number as identifier.

### 4.2.2 Optimization phase results

The speedup gained from the optimization attempt is defined as the original version execution time divided by the optimized version execution time. The overhead of the optimization is the percentage of time required to do optimization over the wall clock time of the optimized version.
<table>
<thead>
<tr>
<th>Var name</th>
<th>Access type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>block</td>
<td>init</td>
</tr>
<tr>
<td>colidx</td>
<td>block</td>
<td>init</td>
</tr>
</tbody>
</table>

(a) CG

<table>
<thead>
<tr>
<th>Var name</th>
<th>Access type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>fjac</td>
<td>block</td>
<td>init</td>
</tr>
<tr>
<td>njac</td>
<td>block</td>
<td>init</td>
</tr>
</tbody>
</table>

(b) BT

<table>
<thead>
<tr>
<th>Var name</th>
<th>Access type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_array</td>
<td>block</td>
<td>init</td>
</tr>
</tbody>
</table>

(c) IS

<table>
<thead>
<tr>
<th>Var name</th>
<th>Access type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>block</td>
<td>init</td>
</tr>
</tbody>
</table>

(d) FT

<table>
<thead>
<tr>
<th>Var name</th>
<th>Access type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhs</td>
<td>cyclic (15 cycles)</td>
<td>init</td>
</tr>
<tr>
<td>forcing</td>
<td>cyclic (5 cycles)</td>
<td>init</td>
</tr>
<tr>
<td>u</td>
<td>cyclic (5 cycles)</td>
<td>init</td>
</tr>
</tbody>
</table>

(e) SP

<table>
<thead>
<tr>
<th>Var name</th>
<th>Access type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>main_211</td>
<td>block</td>
<td>parallel region 0</td>
</tr>
<tr>
<td>main_222</td>
<td>block</td>
<td>parallel region 0</td>
</tr>
<tr>
<td>main_235</td>
<td>block</td>
<td>parallel region 0</td>
</tr>
</tbody>
</table>

(f) MG

Table 4.2: Analysis results.

Figure 4.4(a) shows the speedup on the overall wall clock execution time and the exclusive computation time after DARWIN applied the optimization in subsequent run. The wall clock execution time includes the overhead, initialization, and the computation time. The optimization attempt had a positive impact on the wall clock and computation time of most programs, with up to 1.70x speedup. However, with this small problem size, that was not the case for MG and IS.

MG did not experience an improvement of its wall clock time because the overhead of performing the optimization with the next-touch method and capturing the dynamic data allocation exceeded the optimization benefit, as shown by the overhead breakdown in Figure 6(b). We can reduce the overhead significantly by removing the requirement of capturing the data allocation information if the data size is consistent, e.g. in a production environment.
Figure 4.4: Optimization result on class A with 4 threads: (a) Performance speedup. (b) Optimization overhead.

*IS* did not experience much improvement of its wall clock time because 80% of its execution time was spent in the initialization stage, and this was carried out in serial mode. Adjusting the data placement actually increased the number of remote memory accesses during the initialization of *IS*, which also reduced the optimization benefit.

Figure 4.5: Performance speedup: (a) class A with 4 threads. (b) class B with 8 threads.

To show the flexibility of the optimization method, i.e. that the monitoring results
can be applied to different runtime configurations, we performed another optimization attempt on these benchmark programs with a larger data size and with more threads. In this case, MG and IS also benefit from our optimization. The optimization experiment was performed on class B data size with four compute nodes, eight physical CPUs, and eight OpenMP threads. We reused the same optimization strategy as on the class A programs. Figure 4.5 shows the speedup of this run. It clearly shows that all programs gained a higher speedup if compared to the experiment on the class A data size.
Chapter 5

False Sharing Detection

As discussed earlier in Section 2.2.2, false sharing can lead to a substantial performance problems. To detect it, traditional tools track memory accesses at runtime and then analyze the sequence of memory read and write operations. This method may yield detailed information about the type and amount of cache misses that occur during execution. However the overhead of tracing memory operations can be extremely large, rendering the analysis intractable for large data sizes.

Similar to our method of overcoming the data locality problem on ccNUMA platform, our approach to detect and overcome false sharing with minimal overhead is based upon exploiting higher level information captured by DARWIN, such as the latency and the number of memory access references per thread. By observing the memory access visualization, the developers can detect the occurrence of false sharing and identify the data objects involved.
5.1 Methodology

Our approach of determining the data that exhibits false sharing consists of two stages. The first stage checks whether the cache coherence miss contributes to a major bottleneck in the program. The second stage isolates the data structures that cause the false sharing problem.

5.1.1 Stage 1 : Detecting cache coherence problem

False sharing is a cache coherence problem related to the way processors maintain memory consistency. Therefore, observing the hardware behavior is a good way to determine if a program is suffering from coherence misses. Modern processors accommodate a performance monitoring unit (PMU) that can provide hints about the potential existence of cache coherence problems.

For example, the Intel Itanium 2 processor supports the PMU event \texttt{BUS\_MEM\_READ\_BRIL\_SELF} that gives the number of cache line invalidation transactions [43]. The Intel Core i7 family supports an event called \texttt{MEM\_UNCORE\_RETIRED.OTHER\_CORE\_L2\_HITM} that indicates the number of retired memory load instructions that hit dirty data in sibling cores [18]. If a large number of each event is detected during a program’s execution, it indicates that a serious cache coherence problem can occur when the program is executed with multiple threads.
5.1.2 Stage 2: Isolating the data structures

To identify those data structures that have a major false sharing problem, our method starts by observing the accesses to cache lines with high level symptoms, which are high memory access latency, and large number of references. When a cache coherency miss occurs, a read operation from the processor fetches the data from another processor’s cache or will wait until the memory is updated with the latest version, after which the processor reads directly from memory. This means that a cache coherency miss has longer latency than other kinds of cache misses. It has been reported [24] that a cache coherence miss on the Itanium 2 processor can exceed 180-200 cycles, while the average latency for reading from local main memory is 120-150 cycles. We are also aware that a modest amount of false sharing misses will not lead to a significant problem for application performance. Therefore, cache lines with a low number of references are ignored.

After the data structures showing the symptoms above have been identified, we examine the data allocation information to look for other data structures that share the problematic cache line. The search is also performed within the elements of the data structure if it is a linear or non-linear data structure. As discussed in Section 3.2.3, these kinds of data structures are represented by multiple allocations with the same variable name. If the search attempt returns one or more results, we conclude that the problem is due to false sharing. Otherwise, we observe the data access pattern of the data structures that experience the false sharing symptoms.

False sharing can exist on a data structure that is accessed by multiple threads,
where each thread only accesses a portion of the data. It is possible that some elements near the boundary of two disjoint data portions are in the same cache line. If the data structure size is small, e.g. it takes up about as many bytes as there are in a cache line, most elements of the data structure might be contained in the same cache line causing an increased risk of false sharing.

The results of our false sharing detection are validated by performing manual optimization to the source code that allocates the falsely shared data, without making extensive program restructuring. The optimization is performed by modifying the data layout to prevent falsely shared data from residing on the same cache line. We use GCC’s aligned variable attribute and posix_memalign function to allocate data on the cache line boundary. The result of the detection is considered to be valid when the performance of the optimized code is substantially better.

5.2 Experimental results

To evaluate our methodology, we performed experiments using the Phoenix suite [37] that implements MapReduce for shared memory systems. Phoenix provides eight sample applications parallelized with the Pthreads library that we have ported to OpenMP. The programs were compiled using the OpenUH compiler with optimization level O2. Each sample program included several input configurations. We chose the one that results in a reasonable execution time (not too long nor too short). The platform that we used was an SGI Altix 3700 consisting of 32 nodes with dual 1.3 GHz Intel Itanium 2 processors per node running the SUSE 10 operating system.
The experiment was conducted using an interactive PBS\textsuperscript{1} with four compute nodes and eight physical CPUs. Each program was executed with the *dplace* command to enforce thread affinity. The overall wall clock execution time was measured using the shell’s *time* function.

5.2.1 Results of detecting cache coherence problem

Figure 5.1 presents the speedup for each program executed with different numbers of threads. The speedup is defined as the execution time of the serial run divided by the execution time of the multi-threaded version. The experiment for each configuration was performed three times and the average execution time was determined.

![Figure 5.1: The speedup of the original program.](image)

From all of the programs, only *kmeans* experienced an ideal speedup. The *matrix_multiply* and *pca* programs had reasonable results but did not come close to the ideal speedup. The speedup of *histogram*, *reverse_index*, and *word_count* programs

\textsuperscript{1}Portable Batch Session
was even lower than that obtained by these benchmarks. The `linear_regression`, and `string_match` programs suffered from a heavy slowdown when using more than one thread.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Total Cache Invalidation Count</th>
<th>1-thread</th>
<th>2-threads</th>
<th>4-threads</th>
<th>8-threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td></td>
<td>13</td>
<td>7,820,000</td>
<td>16,532,800</td>
<td>5,959,190</td>
</tr>
<tr>
<td>kmeans</td>
<td></td>
<td>383</td>
<td>28,590</td>
<td>47,541</td>
<td>54,345</td>
</tr>
<tr>
<td>linear_regression</td>
<td></td>
<td>9</td>
<td>417,225,000</td>
<td>254,442,000</td>
<td>154,970,000</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td></td>
<td>31,139</td>
<td>31,152</td>
<td>84,227</td>
<td>101,094</td>
</tr>
<tr>
<td>pca</td>
<td></td>
<td>44,517</td>
<td>46,757</td>
<td>80,373</td>
<td>122,288</td>
</tr>
<tr>
<td>reverse_index</td>
<td></td>
<td>4,284</td>
<td>89,466</td>
<td>217,884</td>
<td>590,013</td>
</tr>
<tr>
<td>string_match</td>
<td></td>
<td>82</td>
<td>82,503,000</td>
<td>73,178,800</td>
<td>221,882,000</td>
</tr>
<tr>
<td>word_count</td>
<td></td>
<td>4,877</td>
<td>6,531,793</td>
<td>18,071,086</td>
<td>68,801,742</td>
</tr>
</tbody>
</table>

Table 5.1: Cache invalidation event measurement result.

To determine whether the slowdown or poor scaling came from the cache coherency problem, we observe the cache invalidation event measurement. Table 5.1 shows the measurement results. It shows that `histogram`, `linear_regression`, `reverse_index`, `string_match`, and `word_count` had a very large number of cache invalidation events when using higher numbers of threads. This is an indication that these programs suffer from a cache coherence problem. Although both programs experienced sub-linear speedup in the number of threads as shown in Figure 5.1, `matrix_multiply` and `pca` had a low number of cache invalidation events.

An examination of the code reveals that the `matrix_multiply` program writes its results into an output file at the end of the program and that the `pca` program has many synchronizations using critical regions, which limits the speedup of both of them. The number of cache invalidation events in `reverse_index` program was pretty
high when executed with eight threads, but the increase was not as extreme as with histogram, linear_regression, string_match, and word_count. We conclude that the coherence problem may not contribute significantly to the overall performance of reverse_index.

We then continue to the second stage of our method on the programs with a high number of cache invalidation events. As case studies, we present the analysis results based on TAU Paraprof visualization for linear_regression and string_match as shown in Figure 5.2 and 5.4 in the following subsections. Both programs experienced an enormous amount of cache invalidations and were slower when executed with multiple threads.

5.2.2 Linear_regression

Figure 5.2(a) shows the average memory latency of the accesses to each cache line. The horizontal axis contains the cache line number. The vertical axis provides the memory access latency. The depth axis shows the thread id. There are two distinct data regions that can be identified in this figure. In data region 1, two cache lines referenced by thread 2 experienced a high latency, while the accesses from all threads to the cache lines in data region 2 had higher latency than most accesses to cache line in data region 1. Next we examine whether the two cache lines in data region 1 and the cache lines in data region 2 also had a high number of references.

Figure 5.2(b) shows the number of memory references to each cache line. The horizontal axis contains the cache line number. The vertical axis provides the number
of references. The depth axis shows the thread id. The accesses to data region 1 were not shown by Paraprof because the number of references to this data region was very small compared to the number of references of region 2.

Figure 5.2(c) gives the total amount of memory latency for each cache line. The total latency is defined as the average memory latency multiplied by the total number of references. Since data region 2 dominated the total memory access latency, we suspected the data structures in this data region to be the leading cause of the false sharing problem.

According to DARWIN’s data-centric information, data region 2 contained accesses to a variable named main\_155. It was a dynamic data allocated by the master
thread in the main function, at line number 155. Its access pattern, presented in Figure 5.2(d), shows that updates to this data structure were distributed among the threads, and that in some cases, multiple threads were accessing the same cache line. The accesses from thread 1 to 3 hit elements of main_155 that reside in the same cache line and have much higher latency than the accesses from thread 1. Therefore, we concluded that accesses to main_155 caused the main false sharing problem in linear_regression. We also found a similar situation with histogram, where dynamic data shared among threads was causing the most significant bottleneck. We validated the findings by adjusting the data allocation alignment using the aligned attribute, as shown in Figure 5.3. The validation results are presented in Section 5.2.4.

```c
typedef struct {
    POINT_T *points __attribute__ (( aligned (256)));
    int num_elems;
    long long SX, SY, SXX, SYY, SXY;
} lreg_args;
...
tid_args = (lreg_args *) calloc(sizeof(lreg_args), num_procs);
```

Figure 5.3: Adjusting the data alignment in linear_regression.

5.2.3 String_match

Figures 5.4(a) and 5.4(b) show the average and total memory access latency of each cache line, respectively. As with linear_regression, we identified two distinct data regions with different access patterns. Both figures clearly show that the accesses to data structures in data region 2 were causing the major bottleneck of this program. According to the data-centric information, data region 2 contained the memory accesses to variable key2_final and string_match_map_266. The first variable was a
global variable allocated in the same cache line with other global variables. We encountered the same situation in reverse_index, and word_count, where some of their global variables resided in the same cache line and were accessed frequently and had high latency.

string_match_map_266 was a dynamic data object allocated in the string_match_map function at line number 266. In contrast to the main_155 variable, whose accesses were distributed among threads in linear_regression, string_match_map_266 was allocated only by thread 3 and privately used within this thread. However, the problematic cache line of this variable was shared with a dynamic variable allocated by other thread.

Table 5.2 presents several data structures allocated in this program. It confirms that key2_final resided in the same cache line with fdata_keys. The first cache line of string_match_map_266, which is the problematic one, was shared with string_match_map_268 that was allocated by thread 2.

The findings were validated by adjusting the data allocation alignment using the
<table>
<thead>
<tr>
<th>Parallel region id</th>
<th>Thread id</th>
<th>Variable name</th>
<th>Starting cache line</th>
<th>Last cache line</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>fdata_keys</td>
<td>0x00004a80</td>
<td>0x00004a80</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>key_2_final</td>
<td>0x00004a80</td>
<td>0x00004a80</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>key_3_final</td>
<td>0x00004b00</td>
<td>0x00004b00</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>string_match_map_268</td>
<td>0x0c031f00</td>
<td>0x0c032300</td>
<td>1024</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>string_match_map_266</td>
<td>0x0c032300</td>
<td>0x0c032700</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 5.2: Data allocation information of several variables in `string_match`.

The `aligned` attribute on `key2_final` and `fdata_keys`. We substitute the `malloc` routine for the allocation of `string_match_map_266` with the `posix_memalign` routine. These attempts are presented in figure 5.5.

```c
char *key2_final __attribute__((aligned (256)));  
char *fdata_keys __attribute__((aligned (256)));  
...  
posix_memalign(&cur_word,256,MAX_REC_LEN);
```

Figure 5.5: Adjusting data alignment in `string_match`.

### 5.2.4 Results of memory alignment

Figure 5.6 shows the speedup of each program over the original one after we performed the adjustments to the source code. The speedup is defined as the execution time of the original program divided by the execution time of the modified one. The memory alignment attempt produced visible improvement of the performance of `histogram`, `linear_regression`, `string_match`, and `word_count` with up to 30x speedup. The `reverse_index` program did not experience any significant improvement. However, this does not mean that our finding on `reverse_index` is invalid.
Table 5.3 presents the cache invalidation count of reverse_index after the memory alignment. It shows that the memory alignment successfully reduced the number of cache invalidations.

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Total Cache Invalidation Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-thread</td>
</tr>
<tr>
<td>Unoptimized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4,284</td>
</tr>
<tr>
<td>Optimized</td>
<td>4,275</td>
</tr>
</tbody>
</table>

Table 5.3: Reverse_index cache invalidation event measurement result.

5.2.5 Performance overhead

To determine overheads, we compare the execution time of the monitoring attempt with the original program execution time. Figure 5.7(a) shows the slowdown of the monitoring phase, defined as the monitoring execution time divided by the original program execution time. The monitoring overhead consisted of the time taken to
capture data allocation, collect the data cache miss samples, and create data-centric information. Each of them was measured using the `gettimeofday` routine. Figure 5.7(b) gives the percentage of each overhead component in the total overhead time.

According to figure 5.7(a) the monitoring phase generated a slowdown ranging from 1.02x to 1.72x, with the average around 1.17x. `reverse_index` had the highest overhead with 1.72x slowdown, while the rest of the programs had less than 1.1x slowdown.

Figure 5.7(b) shows that the majority of `reverse_index`’s overhead was incurred during capture of the data allocation information. The reason for this is because `reverse_index` contained an excessive number of dynamic data allocations. `reverse_index` had 99,675 allocations, while the other programs had less than 50 allocations. Just like our experiment result of the data placement optimization discussed in Section 4.2.1, tracking lots of dynamic data allocation is very expensive.
Chapter 6

Conclusion and Future Work

In this thesis, we introduce the enhancement on our dynamic optimization framework called DARWIN. We improve its modularity and add new functionalities to help OpenMP application developers perform analysis to the application’s dynamic behavior. We also demonstrate how DARWIN can be utilized to overcome false sharing effect and ccNUMA behavior in OpenMP applications. We show that we are able to gather meaningful performance information with low overheads using our framework and that we can utilize it to find the source of the performance bottleneck for both problems. We develop and employ a novel approach for doing so. We are also able to show that our framework can be used to modify runtime behavior during the program’s execution.

To improve the efficiency of the ccNUMA data placement optimization, we collect information on memory references for each thread in order to identify the data access pattern type of a variable. Knowledge of the data access pattern type enables us
to properly map the pages storing a variable to the destination memory nodes in order to reduce the remote memory accesses. The collected information also includes the latency of a single access to a particular page, which can show whether one or more threads have a higher latency than the others. This latency imbalance information help us determine which variables need to be optimized for ccNUMA platform. Our method can be adapted to different runtime configurations because the access pattern of a variable remains constant. The idea is that the pattern can be detected using a small test case and platform configuration, and the resulting information can be subsequently applied to the executions of the full-scale code without further user intervention or re-evaluation needed. The results from our experiments showed significant improvement in application performance and also demonstrated the flexibility of our method.

The second problem we have tackled in this thesis aims to detect and overcome false sharing, another performance issue that is generally unknown to the non-expert programmer. Here, our current work focused on the detection of the occurrence of false sharing and identification of the data objects involved. Our technique consists of two stages, 1) detection of coherence bottlenecks in the program with the help of hardware counters and 2) identification of data objects that cause the false sharing problem. The second stage utilizes the data allocation and access pattern information collected by DARWIN to distinguish the data structures that cause major bottlenecks due to false sharing. Our experimental results shows that our technique has much lower overheads than existing strategies.

Some work still need to be done to cover the limitation of our work. First, our
methodology still requires interaction from the developer, e.g. analyzing the data visualization to characterize data access pattern. Incorporating this into a completely automated strategy will make DARWIN more appealing to the application developers. The second improvement that need to be done is reducing the monitoring overhead. Our experimental result shows traversing the stack frame when capturing the dynamic data allocation can incur large overhead, especially when the program has a large number of dynamic data allocations. To reduce this overhead, we need to devise a way to avoid repeating traversal of stack frames that are already known. Third, we need to evaluate DARWIN with other kinds of performance problem. A work on exploring a performance bottleneck caused by inefficient prefetching is in progress. Lastly, the Itanium system used in this work is relatively old. We need to extend DARWIN to support more modern hardwares like the Intel Core and AMD Opteron processors. A work on incorporating the Instruction Base Sampling (IBS) of AMD Opteron is still in progress. We still have an issue related to the operating system’s kernel; a kernel panic is generated shortly after we use the IBS feature.
Bibliography


