PARALLEL LOW-OVERHEAD DATA COLLECTION FRAMEWORK FOR A RESOURCE CENTRIC PERFORMANCE ANALYSIS TOOL

by

Sunil Shrestha

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2012

© 2012 Sunil Shrestha
All Rights Reserved
PARALLEL LOW-OVERHEAD DATA COLLECTION FRAMEWORK FOR A RESOURCE CENTRIC PERFORMANCE ANALYSIS TOOL

by

Sunil Shrestha

Approved: __________________________________________________________

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: __________________________________________________________

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: __________________________________________________________

Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: __________________________________________________________

Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Guang Gao for his support and guidance through out my study and research in Delaware. His vision, encouragement and stories truly paved my way to what I have achieved.

I would also like to thank all my friends in CAPSL. I have learned a lot during my time here from everyone. Sometimes it is intimidating to see how knowledgeable each one is, but at the same time it gave me a reason to try little harder. Thanks for sharing your knowledge.

I would like to thank our Department secretaries Kathleen Forwood and Debbie Nelson for making my life easy by taking care of lots of little stuff for me.

This thesis is part of the work I did during my internship at Pacific Northwest National Laboratory. I would like to thank every one from HPC branch at PNNL. Special thanks to my advisor at PNNL, Andres Marquez for being one of the coolest bosses ever and for helping me through out my time at PNNL.

Besides, I would like to thank some of my awesome friends: Pabindra Rai, Tony Tran, Joseph Manzano, Juergen Ributzka, Eunjung Park, Ryan Taylor, Ivanka Zhou Li, Stephane Zuckerman, Chen Chen, Murat Bolat, Jakob Siegel, Josh Sutterlein, Daniel Orozco and Elkin Garcia for being there for me when I needed. Like the saying goes ”Friends are reflection of who we are”, I feel better when I think I have awesome friends like you guys. I am very fortuneate to have you guys around.

Repeated thanks to Joseph Manzano without whom this thesis would not have been possible.
Special thanks to Georgette and Joe Metro, who are like my family here in United States. I would not have made it through graduate school without their support. Lastly, thanks to my family who are thousands of miles away but are always in my heart.
to my loving parents
TABLE OF CONTENTS

LIST OF FIGURES ......................................................... ix
LIST OF TABLES ............................................................... xi
ABSTRACT ................................................................. xii

Chapter

1 INTRODUCTION ............................................................. 1
  1.1 Background ......................................................... 1
  1.2 Problem Statement ..................................................... 2
  1.3 Methodology ............................................................. 3
  1.4 Contributions ........................................................ 4
  1.5 Related Works ......................................................... 5
  1.6 Conclusion ............................................................. 5

2 BACKGROUND .............................................................. 7
  2.1 Cray XMT History ....................................................... 9
  2.2 Superscalar to Simultaneous Multithreading ......................... 11
    2.2.1 Superscalar Processors .......................................... 11
    2.2.2 Multithreaded Processors .................................... 11
    2.2.3 Simultaneous Multithreading ................................ 11
  2.3 The Memory Wall ....................................................... 12
  2.4 Performance Analysis Tools ................................ .......... 13
    2.4.1 Static Analysis .................................................... 15
    2.4.2 Dynamic Analysis ............................................... 16
      2.4.2.1 Static Instrumentation Tools ............................... 16
6.3.1.2 The XMT Message Format .......................... 52
6.3.1.3 Twin Region Selection and Modification .......... 52

6.4 Runtime Localization ........................................ 52

7 RESULTS .......................................................... 54
8 COMPARISON ...................................................... 62
9 CONCLUSION AND FUTURE WORK ......................... 64
BIBLIOGRAPHY ...................................................... 66

viii
LIST OF FIGURES

2.1 Symmetric Multiprocessing System ................................. 8
2.2 NUMA Architecture .................................................. 8
2.3 Fine-Grain and Simultaneous Multithreading .................. 12
2.4 Classification of Performance Analysis Tools .................. 14
3.1 A High Level Overview of the MODA Framework ............... 25
3.2 A High Level Overview Encoding/Decoding and Visualization . 28
3.3 Memory Access Count through out the memory subsystem ..... 30
5.1 A High Level Overview of the Monitoring Kernel Framework .. 39
5.2 A High Level Overview of the Collection Framework .......... 40
5.3 Latency for Memory Subsystem ................................. 42
5.4 MODA Trace Format .............................................. 42
6.1 Monitoring Kernel Steps ........................................... 50
7.1 Variable’s access count for Matrix Multiply .................. 56
7.2 Memory Access Counts in Memory Subsystem for Matrix-Multiplication .............................................. 57
7.3 Memory Access Counts over time for Matrix-Multiplication .. 58
7.4 Memory Access Counts for Breadth-First-Search ............... 60
LIST OF TABLES

4.1 Symbols and their meaning ........................................ 34
4.2 Values used for table ............................................... 35
4.3 Overhead given an average tracing rate ......................... 36
ABSTRACT

With the advent of multicore technology, computer systems have shifted to a new height of parallelism and computational power. Ramping up the frequency to increase performance on single processor has become a thing of the past. Nowadays, everyday computers are powered by multiple cores that share resources such as memory, network and I/O components. Moreover, they can run a larger gamut of applications at much higher speed. The increase in computational power is not reflected in the usability of such systems. The usage of these components and resources in an effective manner puts an extremely high burden to the programmer and the system software, which increases the complexity in programming models and runtime systems. To alleviate this burden, there is a need for tools that can identify parallel sections in the code, identify bottlenecks and provides hint to the programmers to improve the performance of the overall computing system. Moreover, the tool has to be able to pinpoint resource contentions so we know where uneven distributions of resources are. To address this issue, we introduced a tool called Memory Observant Data Analysis (MODA) [44]. MODA is a performance analysis tool that helps users analyze resource usage and alleviate resource conflicts by pinpointing performance issues in an algorithmic as well as in an architectural level.

The main challenge of any performance analysis tool is the tool performance itself. Performance analysis tool needs to be such that it introduces minimal to no perturbation of application behavior. This requires analysis tool to achieve information during runtime with a very minimal overhead. This is not easy to achieve because
the tool has to not only monitor the behavior of an application but also record traces, which can later be analyzed. Tool developers in such case need to make a smart decision about the trace format, storage location, data movement and, most importantly, correctness. Under the MODA framework, there are four phases: Instrumentation Phase, Monitoring Phase, Analysis Phase and Visualization Phase. All these phases are equally important for the development of our tool. However, it is the Monitoring Phase that runs online, which brings a need for this phase to be highly optimized. The Monitoring Phase itself is a complex structure that acts like a mini-runtime, collecting all necessary traces and reacting to certain events. This thesis concentrates on the intricacies of creating a monitoring kernel for massively parallel architectures. It goes through the challenges of creating an optimized kernel when several threads are working together. Its contribution includes:

Contribution (1) Parallel trace collection: Monitoring kernel is designed to collect traces from multiple streams running in parallel. There is a multi-layer collection framework in which application streams write to their own local buffers with minimal overhead; while the monitoring threads collect the monitoring data in parallel.

Contribution (2) Differential Compressed Traces: The information collected from each memory operation includes its control information (i.e. program counter), operation’s target, operation’s starter and timing information. In order to reduce communication and space overhead, the framework takes advantage of the operation’s locality in both time and space and encodes the messages to reduce stored messages to 1:3 ratio.

Contribution (3) Low overhead kernel: The framework has a highly optimized monitoring kernel, which intercepts the marked memory operations, takes care of any outstanding events and ensures the consistency of the program.

Contribution (4) Statistical Sampling: This thesis provides an analytical formula
that can be used to calculate the predicted overhead of our framework when running for all memory operations, or when applying sampling to the data. In our result we show 45x more overhead without using statistical sampling, which reduces significantly as statistical sampling is applied.

In this thesis we discuss our solution and show a high level implementation on a massively parallel architecture: the Cray XMT.
In the era of multicore technology, computer systems boast tremendous computational power. Nowadays, computer systems have multiple processors connected to each other with high speed interconnect networks. In addition to that, processors have multiples cores within them connected together as a single unit. These designs create a complex intra/inter network structures that allow access to resources like memory, network, bandwidth and I/O components. Inside the processor components, the complexity does not decrease since there are complex structures like multiple cache hierarchies, out of order engines, load/store queues, etc. Although these components work in tandem to provide large computational power, it increases the complexity of the entire software stack. Moreover, the increasing gap between the memory and processor speeds (see section 2.3) puts additional burden on both the application writer and the system software designer to make efficient use of the available resources.

1.1 Background
Computer systems have taken a huge leap forward in last couple of decades. However, system tools and software stacks have lagged little behind. Most of the existing performance analysis tools are control centric and can identify parallel regions, as well as, sets of threads responsible for performance degradation. Some tools record memory events and provide excellent cache analysis, however their analysis doesn’t go beyond the cache hierarchy. With such limited analysis, it is very difficult to
point out the culprit for performance degradation in current complicated systems. However, there do exist many good analysis tools and they are covered in detail in chapter 2.

In order to provide a resource centric perspective of performance analysis, the Memory Observant Data Analysis (MODA) was introduced. MODA is a memory centric performance analysis tool, first developed on the Cray XMT supercomputer. Some of its important features are postmortem analysis of compressed trace files, client/server based GUI backend, transparent variable selection and highly optimize monitoring kernel. Thanks to these features, MODA is capable of pinpointing contention all way down to the memory bank level.

The target chosen for MODA, the Cray XMT [40], is a highly parallel multithreaded architecture, which trades its locality exploiting hardware for uniform average latency across its machine. Due to these features, it is a perfect target for irregular application. The machine used for MODA has 128 processors connected to each other with a high speed Cray XT network forming a 3D Torus. Each processor had 128 hardware streams and supports 128-way SMT concurrency in hardware. XMT has globally accessible shared distributed memory system and offers fine grain synchronization in memory. All these features offer MODA the platform needed to run large enough application and find out bottlenecks in an application early in the development before scaling out into production.

1.2 Problem Statement

Due to the multicore revolution, the computer paradigm has shifted from being compute bound to resource bound. Application execution times are significantly dominated by the amount of time spent accessing local and remote memories. Several factors such as network congestion, bank contention and load imbalance adds more to the existing problem. However, the software stacks and tools of the current systems are not prepared for this change. Existing tools are mostly control centric,
don’t provide insight about memory usage and most of them incur huge overhead while collecting traces.

When most computer systems were composed of single cores tightly coupled with a specific memory structure, control centric information was enough to point out the source of bottlenecks and would provide enough information for programmer to find a solution. Contrary to this, current systems are much more complex and such tools are not able to pinpoint the source of performance degradation for the underlying hardware. Clearly, there is a need for a tool that can go beyond the virtual memory system and provide detail information about algorithmic as well as architectural bottlenecks. Moreover, such analysis tools have to be designed to run efficiently incurring minimal overhead and maximizing the usage of available resources. This brings us to the following questions:

**Question 1:** In computer systems where multiple threads are running in parallel, can we collect memory traces utilizing parallel resources?

**Question 2:** Can we take advantage of temporal and spatial locality of memory accesses and compress traces to minimize time and space overhead?

**Question 3:** Can we develop an efficient tool that incur minimal overhead?

**Question 4:** Can such tool’s performance be backed up mathematically such that programmer can trade between speed and accuracy of the collected traces?

In this thesis, we attempt to address above-mentioned issues and present our solution to the problems.

### 1.3 Methodology

The MODA framework is built with four different phases: Instrumentation Phase, Monitoring Phase, Analysis Phase and Visualization Phase. The Instrumentation Phase marks all the global and dynamic address ranges that we want to monitor. The Monitoring Phase monitors the accesses to those memory addresses during
runtime. It stores compressed traces with information such as memory address, program counter and timestamps. The traces are later used during its Analysis Phase to outline possible performance deterrents and bottlenecks of the given application for the underlying hardware. Finally, the Visualization Phase puts the analyzed traces in a nice visual form showing access counts for different processors, streams, variables and memory banks. Like any other performance analysis tool, MODA faces the challenge of improving its own performance. All MODA’s phases are very crucial for the efficiency of the tool, however it is the Monitoring Phase that runs in parallel with the application, which makes it very important for this phase to be highly optimized.

The Monitoring Phase is a mini-runtime that runs in conjunction with the application. It uses efficient trace format, local storage and efficient work distribution to minimize the overhead. Every time an exception is triggered, the control is handed to the monitoring kernel using low-latency context switching methods. Inside the kernel, traces are recorded in each stream’s respective local buffer. Besides, monitoring kernel also runs interrupted operations into completion and reset the state of the machine back to original before returning back to the application. In order to fully optimize trace collection process, the Monitoring Phase has been written from scratch in assembly and it has been highly optimized.

1.4 Contributions

The work presented in this thesis provides following contributions:

Contribution (1) Parallel trace collection: Monitoring kernel is designed to collect traces from multiple streams running in parallel. Each streams have allocated local buffers where they can read/write traces with minimal overhead (see chapter 5 section 5.1).

Contribution (2) Differential Compressed Traces: Traces collected in each run have some commonality because of operation’s locality in both time and space. Traces are
compressed based on the differential from the previously saved traces (see chapter
5 section 5.4)

Contribution (3) Low overhead kernel: The framework has a highly optimized mon-
itoring kernel, which intercepts the marked memory operations, takes care of any
outstanding events and ensures the consistency of the program. (see chapter 6
section 6.3).

Contribution (4) Statistical Sampling: Recording every memory traces can incur
up to 45x more overhead (chapter 7). In order find the balance between speed and
accuracy statistical sampling is used. This thesis provides an analytical formula that
can be used to calculate the predicted overhead of our framework when running for
all memory operations, or when applying sampling to the data. (see chapter 4 section
4.2).

1.5 Related Works
Currently there exists few tools than can do any sort of memory analysis. Due to
the space limitation, we only compare our tool with two existing tools METRIC
and PIN. Details about the comparison is presented in chapter 8.

1.6 Conclusion
In summary, this thesis focuses on the monitoring kernel of our performance analysis
tool, MODA. It points out the importance for any performance analysis tool to run
efficiently with minimal perturbation of the application behavior, the challenges it
faces in order to achieve this goal and provides the solution in the form of an imple-
mentation on the Cray XMT Supercomputer. Moreover, our solution is described
in detail, including differential compressed message system, runtime optimization
and multi-level fully synchronous buffers. This thesis is organized as follows: First
it gives the background information, secondly it talks about MODA tool and it’s
phases briefly, then it talks about the Monitoring Phase and how it hijacks the Cray
Runtime Primary Trap Handler, optimization done to make this phase efficient and finally some results to show how the traces generated are used by the tool.
Chapter 2

BACKGROUND

Since the first single chip Intel 4004 microprocessor in 1971, computer systems have taken a huge leap forward. Computer systems have gone from 4-bit to 64-bit CPU, 2300 transistors to billions, KHz frequency to GHz and performance just in past couple of decades has improved from gigaflops to petaflops and is still improving. Current computers have multiple processors connected together with a complicated network. Processors in a same system can be homogeneous or heterogeneous with completely different instruction sets. Memory can be shared and centralized such as in Symmetric Multiprocessing System (see figure 2.1) or shared and distributed such as in NUMA Architecture (see figure 2.2). Software stacks are getting more and more complicated. In such case, writing optimal code in assembly has become a trend of past. This was easy when programs were sequential and hardwares were easy to understand. In contrast, programs today are very sophisticated and their behavior is very difficult to predict. Utilizing resources efficiently in such case is extremely difficult.

Compilers today are very advanced and can perform many local, interprocedural, language-independent, machine independent as machine dependent optimizations [13, 39]. Such optimizations do help in taking advantage of available architectural resources. The very important factor in programming nowadays is also scalability. An optimized program might run very efficiently in one core but might not scale as the number of cores is increased. Many factors come into play when core counts is
**Figure 2.1:** Symmetric Multiprocessing System

**Figure 2.2:** NUMA Architecture
increased and cross core communications are required. Contention in memory controller, memory banks and congestion in networks all contribute to the performance degradation and are very difficult to detect.

Many current architectures have superscalar design, support multi-threading, instruction reordering and have multi-level memory hierarchy [34]. These architectural features aim to squeeze every bit to thread-level and instruction-level parallelism out of the executing program. A very good example of a state of the art architecture designed to maximize parallelism is the Cray XMT. It is highly parallel in nature and exhibits large amount of concurrency. It supports simultaneous multithreading, low overhead context-switching and randomized its memory addresses to minimize contention in the network. It is ideal for irregular application and does a great job in latency-hiding by allowing multiple memory in flight and maximizing the utilization of the pipeline with multithreading.

2.1 Cray XMT History

In 1982, Denelcor, Inc. introduced Heterogeneous Element Processor (HEP), which comprised heterogeneous components such as processors, data memory modules and I/O modules. The system could poses up to 16 processors and each processor could handle up to 50 processes in the hardware queue. Upto 8 processors were allowed in the eight staged pipeline [3].

Later in 1987, Burton Smith, architect of HEP, and James Rottolk started the Tera Computer Company. Their first supercomputer was a barrel processor (BP), also know as interleaved multi-threading. BP switched between threads of execution every cycle to hide the memory access latency and used address randomization to minimize contention [9].

On the other hand, Cray Inc. founded by Seymour Cray in 1972, merged with Silicon Graphics, Inc. (SGI) and formed a Cray Research business unit in 1999 to focus
on high-end supercomputing. This branch was later acquired by Tera Computer Company in 2000 and the company was renamed to Cray Inc [2].

Cray Inc. developed the Cray MTA and the Cray MTA-2 supercomputers, which are barrel processors. Both of these machines had thousands of threads, used fine grain synchronization and were ideal for irregular applications. The addresses were randomized to minimize contention and each CPU posed 128 threads. The next step of development was to make MTA scalable so the application can benefit from multiple processors.

Developed by Cray Inc, Cray XMT is the third generation of Cray MTA Supercomputer [14] after Cray MTA and Cray MTA-2 respectively. Cray XMT is a highly parallel system that utilizes the AMD Torrenza Innovation Socket technology with custom threadstorm chips. The system is equipped with up to 512 processors connected to each other with a high speed Cray XT network forming a 3D torus. Each physical processor can supports 128 instruction streams at a time. Instruction streams are programmed like a Wide-instruction RISC processor and can hold up to three operations at a time. Each processor poses 8GB of memory, which are shared by all streams within the processor as well as all processors in the system forming a shared distributed memory system. Each processor’s pipeline is shared by all its streams and when all processors run currently it exhibits simultaneous multithreading producing tremendous computational power.

The Cray XMT system is composed of compute nodes and service nodes. The operating system runs a multithreaded kernel on compute blades. The service nodes and I/O blades use a Linux kernel. Service blades are configured for I/O, login, network and for some scalar processing. Pacific Northwest National Lab (PNNL) and Cray Inc work closely on different application and tool research that could innovate future high performance computing [40, 18, 46]. This thesis is a part of the work done for the MODA tool developed at PNNL.
2.2 Superscalar to Simultaneous Multithreading

Several attempts were made to push the performance of a system to the next higher level. In this mix, superscalar, multithreading, multiprocessors and simultaneous multithreading technologies were developed. The result was the development of systems that could take advantage of both thread level and instruction level parallelism simultaneously.

2.2.1 Superscalar Processors

Superscalar processors are designed to implement instruction level parallelism within a single processor. Superscalars exhibit multiple execution resources that execute multiple instructions in different functional units. Although very popular, this scheme can leave lots of execution slots unused.

2.2.2 Multithreaded Processors

Multithreaded processors are designed to hide the latency of memory accesses. It saves hardware states for different threads of execution and tries to switch between threads each cycle. Multithreading can be coarse-grain multithreading and fine-grain multithreading. Coarse-grain multithreading does context switching only when long-latency event is encountered whereas fine-grain multithreading does context switching every clock cycle.

2.2.3 Simultaneous Multithreading

Simultaneous Multithreading (SMT) is a processor design that helps to increase parallelism by exploiting both thread level and instruction level parallelism. It combines multithreading with hardware features such as wide-issue superscalars. It has hardware state for several threads and can issue multiple of them simultaneously. In addition to that, it can issue multiple instructions from multiple threads each cycle. [61, 27, 42, 60].
SMT is an evolutionary design and help to minimize a waste of resources in wide range superscalars. figure 2.3 shows the resource usage in superscalar machine for fine-grain multithreading and simultaneous multithreading. With such design computation power increases significantly and so is the performance of a machine. However, performance is not a sole factor of computation power, but is also highly dependent on memory access latency.

2.3 The Memory Wall

There exists a huge disparity between processor and memory speed. Since 1980, microprocessor’s performance has increased at the rate of 60 %/year, whereas DRAM access time has improved only by 10 %/year or less [19]. The problem is clearly visible and many cycles are wasted while waiting for memory reads/writes to complete. This trend seems to continue and is expected to remain for at least next few years. In such cases, a bottleneck towards achieving higher performance has been the cost to access the memory. Resources are under utilized when memory accesses
take significantly longer and following memory operations have to wait [63].

In order to solve this problem, there have been many efforts, both in hardware and software that aim to push memory wall further. One of the most important of such attempts has been the implementation of cache/multiple memory hierarchy [26]. Multi-level memory with different access time takes advantage of recently used data and tries to keep them in the closest possible memory for future use. If the same memory reference is reused in a short period, it is called *temporal locality*. If memory references in close proximity to the recent memory accesses are accessed, it is called *spatial locality*. Both, temporal and spatial locality can significantly improve data reuse and improve performance [23]. In addition, there are other efforts such as the implementation of decoupled architectures [41, 56], prefetching with tiling and percolation[30, 38], memory space reuse [22] and some others that can reduce memory access latency. However, given an architecture with limited resources and different optimization for an algorithm, performance depends on many factors such as dependencies, resource availability and contention [28, 33]. This brings a need of tools that can do analysis on code and help improve the performance of an algorithm.

2.4 Performance Analysis Tools

With complexity in modern computers, it is virtually impossible to fine tune applications by hand or by using rudimentary analysis tools. Performance analysis tools have to adapt to changes in architecture and different performance issues that can arise with new designs. Most of the existing performance analysis tools are control or processor-centric. They can capture traces and present an execution evolution, however they attribute the performance issue to particular thread or program phases. In the past, when processors were single core, finding a memory-bound problem using information as CPU utilization and counting memory accesses over certain period of time was enough to find out the source of performance degradation. In contrast, current computers have multiple processors/cores with shared resources
such as network and multiple memory hierarchy. In addition to that, new architectures are accompanied by newer programming, memory and execution models. Analyzing resources for such architectures can be complicated. However, there exist many good tools than can analyze different program phases and shows performance issues.

**Figure 2.4:** Classification of Performance Analysis Tools

figure 2.4 shows the classification of performance analysis tools. Existing tools can be classified into two main categories based on their analysis scheme. They are static
analysis tools and dynamic analysis tools. Dynamic analysis tools can be further divided into three sub categories based on how the tool does the instrumentation. They are static instrumentation, dynamic instrumentation and the hybrid of the two [4].

2.4.1 Static Analysis
Static analysis tools analyze the code without executing them. Such tools are used to find race conditions, deadlocks, memory leaks, unreachable code, security vulnerabilities, glitches, redundancies and more [15, 16, 17]. Such tools attempt to find all possibilities where a code can be misinterpreted. They don’t depend on any input sets, but instead it prevents future failure by early detection of glitches without running the code. Compilers in general do a very good job of performing static analysis. However, their analysis is mostly approximation of runtime behavior and is always conservative. Thus, such analysis can still miss out on inconsistencies, bugs and optimization opportunities that requires one to look across multiple modules. Next we will briefly discuss some static analyzers used today.

PC-lint [5] is a static analyzer for C/C++ and supports platforms such as Windows family, MS-DOS and OS/2. It does analysis such as value tracking, type checking, semantic checking, control flow checking etc and produces an output that can be used by tools like A Lint Output Analyzer (ALOA). ALOA can produce various useful metrics and help user to see the overview of the Lint issues.

Clang [1] is a static analysis tool for C and Objective-C programs and supports Windows, Linux and Mac OS X. It is implemented as a C++ library. It helps to analyze and automatically find bugs. It can be used as a standalone tool in which case it is used using the command line. It is also used as a frontend from the LLVM compiler.

Frama-C [21] is designed to do automatic analysis on source code and supports Windows and Mac OS X. It lets the user navigate through the dataflow of a program,
slice the program to simple ones and observe possible values for the variable in a program. It claims to guarantee that the program is bug free. Above mentioned tools do good static analysis and can find bugs, inconsistencies and optimization opportunities. However, every static analysis tools have their limitations. They are slow and are not always accurate. Since, the code never gets executed, some analysis such as precise pointer or alias analysis, resource usage, bottlenecks, workload imbalance etc are not possible.

2.4.2 Dynamic Analysis
Dynamic analysis tools do code analysis by executing the program. Such analysis is precise because the runtime behavior is recorded and no approximation is required. However, it requires some additional code insertion to the original binary that can affect the behavior of the executing application. Instrumentation for such analysis can be static, dynamic or the hybrid of the two.

2.4.2.1 Static Instrumentation Tools
Static instrumentation tools do not modify the binary image of an application. Instead, they rely on source code instrumentation. The user has to know in advance about the kind of information they’d like to gather prior to instrumentation. Traces can be as simple as gathering timing data or complex as gathering detail report about application and it’s interaction with libraries and kernel. This information can be very useful in studying the behavior of an application. However, the shortcoming of static instrumentation is, it is not possible to provide real time feedback to gather sophisticated data during runtime. In addition to that, static instrumentation scheme requires additional data collection routines that can significantly slow down the application because of the overhead it incurs. Nonetheless, static instrumentation tools still provide valuable information that can help in performance improvement and is one of the simplest and easiest of its kind. Static
instrumentation tools can be sub-divided into three different types depending on how instrumentation is done.

The first one is the *Compile-time instrumentation tools*. These tools can instrument an application during compile time. Before the binary is generated, instrumentation codes are injected. These injected code triggers function calls that can gather statistical data such as number of times a function is called. However, such tools requires a source code for application and will not work if only binary is available such as for libraries and kernels. e.g. Scalasca.

Scalasca [8] lets manual instrumentation inserting directive and automatic instrumentation using compiler or (Tuning and Analysis Utilities) TAU [53]. It collects measurements for processes and threads and attributes them to events. Multiple traces, one per process are generated that can be transformed into compact call-path profile. These traces can be analyzed in parallel by replaying the traces to understand the communication behavior. Scalasca supports MPI and uses EPILOG format. The results are presented to show performance problem, call path and the system resources.

The second type is the *Hardware Counter tools*. The shrinking transistor size let the new design to include lot more functionality than past designs. In such case, recent architectures are equipped with on-chip programmable hardware counters that can be used to count different events and that can provide information about the internal state of processors. These counters can provide more accurate information such as cache misses, different types of operation count, as they are all hardware controlled. Since hardware resources are limited, different statistical data can be collected at multiple runs and can be integrated together later. e.g. DECAN.

Decremental Analysis (DECAN) [20] uses hardware counters to find specific instruction responsible for certain memory behavior. This is done by first measuring timing
for original version of the code and then measuring code after some expression removal. The timing difference gives the overall instruction effect on the behavior.

Hardware counters can also be used in *Sampling tools*. These tools wrap around the execution such that execution is halted to record statistical data such as timing information or function call counts. In such case, these tools can also monitor libraries that are only available as binaries, however they are not able to monitor kernel code. It is very important to find a balance between accuracy and the performance for such tools because sampling every execution can jeopardize the performance significantly and sampling very few can skew the results collected. e.g. HPCToolkit.

Rice HPCToolkit [50] provides language independent tool suite that sample execution of multithreaded programs. It uses hardware performance counters and call stack unwinding to do the measurements of the sampling event, which it links to the calling context. Metrics collected can be operation count, cache misses, pipeline stalls and inter-cache communication. This information is used to understand the behavior of a parallel program.

Finally, there are static instrumentation tools, than can combine above mentioned static instrumentation method into one to form a compound type called *Compound tools*. Such tools can provide multiple features but also come with the shortcomings associated with each of the above kinds. e.g. Intel VTune.

Intel VTune [36], implemented for x86 based machines, can do stack sampling, thread profiling and event sampling. It can provide the information about time spent in each sub routine. Time spent for instructions provide information about pipeline stalls. It is capable of providing tuning opportunities for better cache hits and better branch predictions. In addition, Intel Performance Tuning Utility (PTU) offers hardware counters than can be used to do in-depth analysis on memory system.
2.4.2.2 Dynamic Instrumentation Tools

Different from the static instrumentation is the Dynamic instrumentation tools. These tools can modify the binary image of an application in order to record statistical data. These tools provide a flexibility of incrementing a running process by inserting analysis routine dynamically. With such method, highly accurate statistics can be recorded. However, insertion of additional code can significantly slow down an application. In addition to that, additional code alters the program flow and the instruction flow in the pipeline. This can vary the program behavior giving an illusion of different execution characteristics. It is therefore very important to select segments with higher chances of performance issues so only those segments can be instrumented. Dynamic instrumentation tools can be sub-divided into two different types.

The first kind is the *Binary instrumentation tools*. These tools can do binary rewriting in different program phases and libraries to do custom analysis during the execution of an application. The advantage of this method is that the user can insert analysis code dynamically when statistical data is needed. In case when statistical data is not needed, the application can run without any interruption and without any overhead. With such tools, users have more freedom to do instrumentation and custom analysis at multi-level software hierarchy. e.g. METRIC.

METRIC [45] provides partial memory traces that can be used to isolate and understand potential memory issues. The traces can be used for memory hierarchy simulation to do memory characterization. The tracing is enabled using dynamic binary rewriting which let user to extract traces without recompiling. The analysis can be done offline.

The second type of dynamic instrumentation tool is the *Probing tools*. These tools use pre-defined probes that can be used to monitor different events and gather data for analysis. Such tools dont give much freedom to the users to customize the probe.
They rely heavily on existing probes supported by libraries and kernels. Such tools make instrumentation much easier but not as optimal as desired. e.g. DynaProf. DynaProf [25] is dynamic profiling tool based on Dynamic Probe Class Library (DPCL) that installs probes at different location to extract profiling data. In the current state, probes are available for wall clock profiling, hardware counter profiling, and statistical sampling of program counter. It supports MPI and OpenMP.

2.4.2.3 Hybrid Instrumentation Tools

There are tools that can combine the approach from dynamic and static instrumentation, called the Hybrid instrumentation tools. Such tools are more flexible and provide freedom as it can take advantage of resources like hardware counters and mix them with dynamic instrumentation. Such tools on one hand combine the benefits of two different approaches and on the other hand have the shortcomings of both approaches. Such tools can only excel if there is a balance between the two kinds of instrumentation and are less frequently used. e.g. Vulcan.

Vulcan [54] uses both static and dynamic binary code modifications and provides rich API for instrumentation. It can be used for basic block counting, path profiling, data reference profiling and for several optimization opportunities and software management. It allows partial compilation, uses abstract representation to allow mix instruction set binaries and also allows cross-component optimization for heterogeneous architecture. In its current form it can process x86, IA64 and MSIL binaries.

In addition to static, dynamic and hybrid instrumentation tools, there also exist debuggers that can help to detect errors in programs through step-by-step execution. In addition to that, they can also do stack traces and register, symbols, memory location and expression watches. Such information can be useful in different kind of analysis to find out program’s behavior. e.g. Totalview.
Totalview [31] gives user the control over threads and processes execution and provides details about program states and variables. It helps reproducing problems in concurrent programs to help with debugging. In addition to that, it provides some memory analysis to find memory errors and diagnose deadlock and race conditions. It supports OpenMP and MPI and also is useful in systems with GPUs.

Besides the above-mentioned performance analysis tool, there are many other analysis tools such as HPM Toolkit [24], PE Benchmarker Toolset [6], Paraver and Dimemas [10], Performance Toolbox [6] and more. HPM Toolkit is a set of libraries provided by IBM that lets the user access hardware event counts and collect hardware events such as clock cycles, L1/L2 misses, TLB misses and branch miss prediction. PE Benchmarker Toolset is designed to analyze programs performance within IBM Parallel Environment for AIX. Paraver is used for hardware counter profiling and operating system activity, whereas Dimemas is used for message-passing programs providing prediction of performance. Performance Toolbox provides graphical interface for monitoring and analyzing resource usage. It provides statistical information such as CPU utilization, disk, paging and network activity.

There also exist tools like PIN [43], ATOM [55] and Valgrind [47] that provides the framework for building tools. PIN provides a framework to build different performance analysis tools and perform simulation studies. With ATOM, user just needs to define tool-specific details in instrumentation and analysis routine. Similarly, Valgrind is a framework for creating supervision tools for profiling and bug detection.

Also in existence are profilers such as Prof, Gprof, Tprof, Monitor, Xprofiler and MpiP [32, 6, 7]. Prof is used to profile execution at the procedure level. It provides information such as CPU time, execution time used by procedures and the frequency of those calls. Like Prof, Gprof also profile execution at procedure level. However, it does so according to their call graphs. It provides information such as parent of
each procedures, index for procedure and breakdown of time between procedure and descendants. Tprof provides CPU usage for individual program and the system, so user can check for competing processes. Monitor shows real time display on how process are utilizing machine resources. Xprofiler is based on Gprof and provides graphical utility. MpiP is a lightweight profiling library for MPI applications. It captures statistical information such as where MPI call is made, time taken for call and high frequency call sites.

In its current form, Cray XMT comes with Apprentice tool-suite [49] that helps in program analysis using compiler analysis tool (canal), a block-profiler (Bprof) and a trace analyzer. Canal helps the user identify parallel and sequential loops within the program. In addition to that, it also provides information about the limitations that hinders concurrency. Bprof helps to identify functions that take longest time to execute. It uses Tview to present the performance metric during program execution over time. The performance metrics provided by apprentice tool, however depends solely on the traps initiated during execution and has very limited memory analysis functionality. However there are tools like ThreadSpotter, METRICS that we discussed earlier that can do some sort of memory analysis. In addition to that, there are tools like PSigma that integrate data centric view to the existing control centric approach.

Rogue Wave’s Threadspotter [51] can identify hot spot by analyzing memory access pattern. It generates traces that shows cache related information and can pinpoint to the original source code using a HTML based report. However, the analysis doesn’t go beyond cache.

PSigma [52] interrupts the control-flow at specified location letting user to probe into the execution of the application. Its framework comprises of event catalogue, event handlers and the module psigmaInst that does the instrumentation. It helps to examine performance events and map them to data and control components at
source level to find out the source of performance degradation.

Despite the fact that, all above-mentioned tools provide invaluable information that can be used to find out performance issues. However, most of these tools are processor centric and attributes degradation source to a certain loops or threads without much information about the source of bottlenecks. Memory analysis done by some of these tools are limited to cache and don't go beyond the virtual address space. In other words, memory centric tool like MODA [44] that can do memory analysis and pinpoint the source of contention is scant. In multicore era, where resources such as bandwidth, I/O and memory are invaluable assets, we believe resource centric analysis can provide better insights on execution evolution and a full picture of resource usage.

Note: The classification done in this thesis is just one of many ways of classifying tools and there exist many other ways as well.
Chapter 3

MEMORY OBSERVANT DATA ANALYSIS: MODA

MODA is a memory centric performance analysis tool currently implemented on the Cray XMT architecture [29]. MODA hijacks the runtime system and takes control of the running threads to generate memory traces. It is designed to instrument and monitor applications and to help to analyze sources of performance degradation. In its current state, analysis is done offline and is followed by an interactive Visualization Phase that can display the execution evolution as well as contention in different memory subsystems. The entire process is done in parallel and has a minimal perturbation in the application behavior. The MODA framework is built with four different phases: Instrumentation Phase, Monitoring Phase, Analysis Phase and Visualization Phase.

An overview of MODA is shown in figure 3.1. The framework uses the application binary’s symbol information to construct a data structure dictionary for global and static variables and runtime wrapper functions to log heap-allocated data structures into another dictionary. The dictionaries are called static and dynamic, respectively. The Monitoring Phase reads these dictionaries and registers memory addresses so they can later be monitored.

When such variables are accessed, compressed traces are generated. These traces are used by the Analysis Phase where traces are decoded and reconstructed. The tool's user will select different histogram charts to display during the Visualization Phase. In addition to these phases, a very important aspect of the MODA tool is
its ability to map the address in the virtual address space to the physical address space during its analysis using the MODA memory model (see section 3.5). This feature enables MODA to pinpoint the contention down to the memory bank level.

![Diagram of the MODA Framework]

**Figure 3.1:** A High Level Overview of the MODA Framework

### 3.1 Instrumentation Phase

MODA’s first phase is the Instrumentation Phase. During this phase, certain parts of the framework are set into place. First, the compiler’s code generator and the binary rewriting engine will create twin code sections for the regions that need to be monitored. Next, it will create list of the all the resolved function names and variables and replaces “faux” targets inside the monitor kernel with resolved names. Finally, it replaces the program’s entry point with a pointer to a code segment that initialize, run the desired application and clean-up the framework.
Next, the binary rewriting engine uses the collected information for global and static data structures (i.e. size, virtual address and name) to create a file. After this process, the static dictionary file is created and pruned of “default” data (like runtime and OS related global data variables and structures). These memory ranges (i.e. data structures) will be marked for tracing when the framework initializes its structures during the Monitoring Phase. The process of marking these structures is called “registration”. During the Monitoring Phase, a similar file will be created which will contain the heap-allocated structures as they are created and destroyed during execution. When a memory range has ended its lifetime (either at the end of the execution or by de-allocating a heap allocated variable) the range will be unmarked in a process called de-registration.

After the static dictionary and the modified binary have been created, the next phase starts as soon as the user run its modified application.

### 3.2 Monitoring Phase

When the initialization function of the Monitoring Phase starts, it has two main jobs to do: to create the tracing buffers and to initialize the “reconstruction streams.” The MODA framework allocates memory space where traces can be saved by application streams inside the monitoring kernel. These allocations are local to each processing stream. This way the memory access time is minimal and they can all be processed in parallel. Each allocated local memory is divided into two buffers. The first buffer is called the trace buffer where application streams record its traces during runtime. The recording happens in round robin fashion as it loops through the limited buffer space. In order to guarantee no loss of traces, the framework uses synchronization construct i.e. full/empty bits, provided by XMT. The second buffer is the collection buffer, which is further divided into two sub-buffers to provide double buffering. When a first collection buffer hits the predetermined threshold, the traces are transferred to the permanent location using protocols provided by the
Lightweight User Communication (LUC) library [35]. During this process, the second buffer can be used to save the traces and vice versa. This helps the continuous collection and flow of the data streams with minimal overhead.

Next, the initialization function creates helper streams called the “reconstruction streams”. For each processor in the system, the framework creates a fixed number of reconstruction streams that can run in parallel with the application streams. As the traces are recorded, reconstruction streams are responsible for trace data movement in parallel with minimal perturbation of application behavior. It is very important to pick the right number of reconstruction streams. Too many reconstruction streams will reduce the number of application streams reducing parallelism and too few will get overwhelmed by the enormous amount of traces that application streams produce. In our current implementation, we used 16 reconstruction streams for each processor. This means that there can be 112 application streams running in parallel and generating traces. These traces are recorded in trace buffer and are read by 16 reconstruction streams. Each reconstruction stream therefore loops through 7 different trace buffers to check for traces.

When the user code starts, the MODA runtime monitors the accesses to the registered (i.e. marked) memory references. This is done by hijacking the original runtime system by overwriting the addresses at predetermined location using binary rewriting. This process was accomplished during the last phase. When the registered addresses are accessed in the application, it initiates an exception and control is given to the runtime’s primary trap handler. Here, it checks the type of exception that was raised. If the exception is of a predefined type, the framework takes control away from primary trap handler and executes the monitoring kernel. The kernel is responsible for temporal sampling, trace collection and re-execution of the interrupted memory operations. This phase runs in parallel with the application and relies on reconstruction streams for data movement. The monitoring kernel’s
complexity is rather high because it is designed to reproduce several aspects of the given runtime. Thus, it has been written in assembly in order to apply the maximum possible number of optimizations. The implementation details covering all the issues and solutions can be found in chapter 6.

The information recorded inside the monitoring kernel is the compressed traces with information such as memory address, program counter, processor identifier, stream identifier and timestamps. As the traces are being generated, they are read and saved in parallel by the helper reconstruction streams. The data movement happens as explained in section 5.1

The traces are later used in the Analysis and the Visualization Phase, which is explained briefly in the following sections.

![Figure 3.2: A High Level Overview Encoding/Decoding and Visualization](image)

### 3.3 Analysis Phase

The traces generated by the Monitoring Phase are stored in the disk with the marker to identify the processors and streams it belongs to. These traces are parsed, sorted, compressed and analyzed in the Analysis Phase.
3.3.1 Encoding and Decoding

The traces generated by the MODA framework can be in tera or giga bytes in size. In order to carefully handle this huge amount of traces, MODA implements the Parallel Compression Encoder/Decoder (PCED) system. The idea here is to reduce the file size to a manageable size and still preserve maximum amount of memory references recorded.

A high level overview of PCED is shown in figure 3.2. Once all the traces are saved in a disk, PCED starts to process the traces by parsing them and recreating the recorded message such as memory address, time stamp and program counter. These messages are then sorted using our parallel 64-bit radix sort. The sorting is done using the time stamp from the traces. The sorted messages are then passed through redundancy optimizer to remove redundant messages. Finally it is passed to a component where it does lossless encoding using adaptive arithmetic coding (AC) algorithm. In this algorithm, probability of symbol is calculated and bits are assigned on the basis of the frequency of occurrences. More information about AC can be found in [62]

The decoding follows a reverse path of encoding and is triggered by the user’s queries. The decoded data is used by the Visualization Phase to display the memory execution evolution. This work was lead by Chun-Yi Su from Virginia Tech.

3.4 Visualization Phase

MODA allows user to make a query that returns the memory usage pattern in visual form. The memory access patterns are plotted over time, per processors, streams and memory banks. The framework uses a client/server remote procedure call to process user requests in parallel. The communication is done using the network endpoint over high-speed network. During analysis, a request is issued at the server side. The client side runs at the control node and provide the Graphical User Interface (GUI). When a request is issued, the communication between GUI and network endpoint is
done using the Java Native Interface (JNI). During this process, the PCED decoding path is taken where required information are obtained and decoded in parallel. The end result is the visual display of the queries in the form of histograms.

During the process of decoding and visualization, MODA uses the memory model to map the virtual address space to the physical address and pinpoint the source of contention. This work was lead by Amanda White from Pacific Northwest National Laboratories.

![Figure 3.3: Memory Access Count throughout the memory subsystem](image)

**Figure 3.3:** Memory Access Count throughout the memory subsystem

### 3.5 MODA Memory Model

In order to present the memory centric view of MODA, the tool has to be aware of architectural memory model. MODA achieves this by using machine address translation mechanism during the Visualization Phase. As the encoded traces get decoded, the trace information is mapped to the physical address. The addresses are then further mapped to individual memory banks using machine characteristics such as its memory bank and page size. The entire process is a reverse process of address translation that outputs the physical address given the virtual memory.
address. This process is very important because during the Visualization Phase, once memory addresses are decoded they are translated to physical addresses to find out the memory banks they belong to. Using this information along with the frequency of access and timing information we can pinpoint the contention to the individual memory banks. Figure 3.3 shows the distribution of memory references throughout the memory subsystem on Cray XMT.
Chapter 4

INSTRUMENTATION PHASE

The Instrumentation Phase is the first phase of the MODA framework. This phase gets the framework ready for monitoring, generating memory traces and moving them to the permanent location in a disk. As explained in chapter 3, this phase does the following tasks: It does (partial) variable selection for instrumentation; and the certain code segments are rewritten to set up certain parts of MODA’s runtime.

4.1 Variable Selection

Every memory address in Cray XMT has an allocated bit known as user defined trap bit. As the name suggests, this bit can be controlled by the user in order to force an exception during execution. This feature let MODA to interrupt the execution process and record desired traces before returning to the normal execution. The details about runtime hijacking are explained chapter 6. During the registration process, MODA framework selects and marks (raise user defined trap bit) the variables to be instrumented during the execution. Static and global variables are selected using an ELF reader. On the other hand, dynamic variables are marked by replacing malloc and free calls with a wrapper malloc and wrapper free functions. With this process, static and global variables are registered and deregistered only once, as they don’t change through the execution. On the other hand, dynamic variables are registered during allocation and deregistered during deallocation. For the dynamic variables, this happens during the execution of the program.
During the process of variable selection, certain variables or memory addresses can be left unmarked providing a feature of spatial sampling. This will reduce the overhead by reducing the amount of traces generated as unmarked variables are not monitored. However, such sampling method can easily ignore crucial memory accesses and thus creating an incomplete access pattern. Nevertheless, these patterns are still useful but might not be accurate. Like any other performance analysis tool, MODA generate traces in giga or tera bytes. It is virtually impossible to record all memory traces. In order to overcome this problem, MODA implements temporal sampling. This process is made possible with the help of code duplication through binary rewriting, which is explained next.

4.2 Region Duplication and Temporal Sampling

MODA framework provides temporal sampling feature by providing a duplicate region of the code. When the compiler reaches the linker stage, the framework calculates the size of the binary and appends an empty space of equal size at the end of the original binary doubling its size. Next, the framework copies an entire binary into this new space where it modifies the control field to disable user defined exception. This is done using binary rewriting as explained in section 4.3.

Temporal Sampling feature is achieved by using the completed instruction count to switch between the original and the duplicate version of the code. It starts with the original binary where it instruments memory accesses included in N consecutively executed instructions. Then it switches to the duplicate code where M instructions are executed and no memory accesses are instrumented during this time. M and N are predefined numbers determined by the users that monitoring kernel uses to make the required switch. The switching between different version of the code also requires updating different special registers provided by Cray XMT along with the target registers which is described in detail in chapter 6.
Table 4.1: Symbols and their meaning

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ReadRate}_{RS}</td>
<td>Rate at which reconstruction streams read from the trace buffer</td>
</tr>
<tr>
<td>\text{FillRate}_{MK}</td>
<td>Rate at which application streams write to the trace buffer</td>
</tr>
<tr>
<td>\text{FillRate}_{ii}</td>
<td>Maximum fill rate governed by maximum instruction issue</td>
</tr>
<tr>
<td>\text{FillRate}_{c}</td>
<td>Constrained fill rate</td>
</tr>
<tr>
<td>\text{Fmax}</td>
<td>Maximum frequency of an architecture</td>
</tr>
<tr>
<td>\text{IIR}</td>
<td>Instruction Issue Rate</td>
</tr>
<tr>
<td>\text{MOF}</td>
<td>Maximum memory operations that can be in flight at any given time for a given stream</td>
</tr>
<tr>
<td>\text{LSD}</td>
<td>Maximum memory instruction i.e. LOAD/STORE density</td>
</tr>
<tr>
<td>\text{LMA}</td>
<td>Local memory access Cycles</td>
</tr>
<tr>
<td>\text{RLS}</td>
<td>Conservative ratio of remove vs local references</td>
</tr>
<tr>
<td>\text{X}</td>
<td>Intermediate Variable</td>
</tr>
<tr>
<td>\text{TOS}</td>
<td>Trace overhead per sample</td>
</tr>
<tr>
<td>\text{TNS}</td>
<td>Trace overhead per non-sample</td>
</tr>
<tr>
<td>\text{TSR}</td>
<td>Average trace sampling rate</td>
</tr>
<tr>
<td>\text{ReadRate}</td>
<td>readdrate</td>
</tr>
</tbody>
</table>

Finding good M and N is very important to provide the temporal sampling without jeopardizing the memory access pattern. In order to investigate M and N, we came up with equations 4.1 to 4.5 (Contribution 4). Here are some of the symbols and their meaning as used these equations.

Equation 4.1 states that the rate at which application streams write in the trace buffer \text{FillRate}_{MK} should not exceed the rate at with reconstruction streams read \text{ReadRate}_{RS}. In order to achieve this monitoring kernel uses full/empty bits to synchronize between reads and write.

$$ \text{ReadRate}_{RS} > \text{FillRate}_{MK} $$

(4.1)

Equation 4.2 calculates the maximum instruction issues fill rate \text{FillRate}_{ii} by calculating the product of maximum frequency of the architecture \text{Fmax}, instruction issue rate \text{IIR}, maximum memory operations that can be in flight at any give time.
Table 4.2: Values used for table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOS</td>
<td>1/45</td>
</tr>
<tr>
<td>TNS</td>
<td>1/2</td>
</tr>
<tr>
<td>IIR</td>
<td>1/215</td>
</tr>
<tr>
<td>MOF</td>
<td>3</td>
</tr>
<tr>
<td>LSD</td>
<td>1/2</td>
</tr>
<tr>
<td>LMA</td>
<td>100</td>
</tr>
<tr>
<td>RLS</td>
<td>1/5</td>
</tr>
<tr>
<td>X</td>
<td>1 MHz (plugging above numbers in 4.4)</td>
</tr>
</tbody>
</table>

MOF and the maximum memory instructions density LSD.

\[
FillRate_{ii} = F_{max} \times IIR \times MOF \times LSD
\]  

(4.2)

However the constrained fill rate \( FillRate_c \) is determined by local fetch latency LMA and the conservative ratio of remote vs local references RLS that is shown in equation 4.3.

\[
FillRate_c = F_{max} \times LMA \times RLS
\]  

(4.3)

The FillRate that can be achieved by the tool is therefore the minimum of two fill rate as shown in 4.4 in terms of \( X \).

\[
X = \min[FillRate_{ii}, FillRate_c]
\]  

(4.4)

The fill rate for the monitoring kernel \( FillRate_{MK} \) is therefore the function of variable \( X \), trace overhead per sample \( TOS \), trace overhead per non-sample \( TNS \) and average trace sampling rate \( TSR \) as shown is 4.5.

\[
FillRate_{MK} = \frac{X \times TOS + TNS}{(TSR \times TNS + (1 - TSR) \times TOS)}
\]  

(4.5)

Using values from table 4.2 in the equations (4.1 - 4.5), we came up with the table as shown in table 4.3 When TSR=0, the application slows down by 2x because of
Table 4.3: Overhead given an average tracing rate

<table>
<thead>
<tr>
<th>TSR</th>
<th>FillRate</th>
<th>Overhead Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>1/100</td>
<td>0.411</td>
<td>~2.43</td>
</tr>
<tr>
<td>1/25</td>
<td>0.268</td>
<td>~3.72</td>
</tr>
<tr>
<td>1/10</td>
<td>0.158</td>
<td>~6.3</td>
</tr>
<tr>
<td>1</td>
<td>0.022</td>
<td>45</td>
</tr>
</tbody>
</table>

the overhead incurred by the indirection created during instrumentation of memory addresses. When each and every memory address is instrumented and traced, this overhead increases to 45x. Therefore it is very important to pick a right sampling ratio in order to get enough traces that can reflect the behavior of an application without jeopardizing the performance. The table shows that if 1 out of every 10-memory reference is traced, the overhead is 6.3x. Similarly, the overhead drops as we reduce the number of traced memory accesses as shown in the table, i.e. the overhead of tracing 1 out of every 25 and 1 out of every 100 memory accesses are 3.72x and 2.43x respectively.

4.3 Binary Rewriting

MODA framework uses binary rewriting to hijack the XMT runtime system to generate traces and to provide temporal sampling (see section 4.2). Once the original binary for an application is created, the framework starts making changes in the binary to add the required functionality for trace collection.

First of all, the framework makes a copy of the entire binary in the newly created and appended space at the end. This new copy, which is the twin version of the binary, is then modified to disable user defined exception. This is achieved by modifying the compiler and forcing it to generate codes with memory operations that have trap control fields. By default, the generated code has user define exceptions enabled. In the duplicate version, however, these trap bits are disabled which helps
the framework to switch between different versions of the code to provide temporal
temporal sampling feature as described in section 4.2.

Second of all, the framework goes through the symbol table and finds out the as-
signed addresses for runtime primary trap handler and the monitoring kernel. Then, it modifies the original binary at predefined location in runtime primary trap han-
der and the monitoring kernel to enable the jump from one to the other in order to hijack the XMT runtime system.

The Instrumentation Phase is followed by the Monitoring Phase where memory accesses are monitored and traces are generated. The following chapter describes the Monitoring Phase.
Chapter 5

MONITORING PHASE

The Monitoring Phase is the second phase of the MODA framework. This phase is responsible for initializing the framework structures and generating traces during the execution of an application. The generated traces are later analyzed to study the behavior of an application and to pinpoint the source of performance degradation. Since this phase runs in parallel with an application code, it is very important for this phase to run with minimal overhead.

As discussed in chapter 4, some of the memory references that we want to monitor are registered in the Instrumentation Phase. However, heap-allocated variables are registered during runtime in the Monitoring Phase. During the execution process, when the application streams try to access the registered memory references, an exception is raised and the control is given to the primary trap handler (PTH). First few lines of PTH are modified during the Instrumentation Phase to redirect the control to the monitoring kernel. Inside, the kernel, information such as memory address, time stamp, processor identifier, stream identifier and program counter are recorded. In order to minimize the space and latency overhead, this information is recorded in compressed format as show in figure 5.4 in a local memory space. The recorded traces are moved to the permanent storage by the collection streams. The monitoring kernel overview is shown in figure 5.1.

In addition to trace collection, the Monitoring Phase is also responsible for providing temporal sampling feature and handing the control back to the Cray XMT runtime
Figure 5.1: A High Level Overview of the Monitoring Kernel Framework

system once all the traces are recorded. The implementation of monitoring kernel is rather complicated and is explained in chapter 6. Here, we’ll talk about the collection framework, different memory hierarchy and their access time and the message format MODA framework uses.

5.1 Collection Scheme Overview

The collection overview can be seen in figure 5.2. As explained before, the framework allocates the nearby local memory for each processor in the system to store memory traces in order to minimize the collection overhead. These low latency local buffers include trace buffer and the double buffered collection buffer. Trace buffers are used by application stream for trace recording during runtime whereas collection buffers are used for temporarily storing traces until they are moved to disk.

During the execution when the registered memory is accessed, it raises a user defined exception. Since multiple streams could be running in parallel, multiple traps can occur in parallel. As mentioned before, the framework takes the control over from XMT runtime and uses application streams to record memory traces. Multiple
streams running in parallel records compressed memory traces in its respective trace buffers (Contribution 1). Since the size of the trace buffer is limited, recording is done in round-robin fashion i.e., when the end of the buffer is hit, it starts recording from the beginning of the trace buffer again. In order to make sure no traces are over written during this process, MODA framework uses a full/empty bits to ensure no trace loss. This also means that the application can slow down significantly if the traces are not read fast enough. To solve this problem, MODA framework provides a temporal sampling feature as described in section 4.2

As the traces are recorded, they are read in parallel by helper streams called reconstruction streams. It is very important to have a balance between application and the reconstruction stream. Too many reconstruction streams would mean fewer application streams which can slow down the application a lot and too few will get easily overwhelmed by too many application streams. Reconstruction streams read traces from trace buffer and move them to one of the collection buffers. When the
collection buffer hits the predefined threshold, they are moved to a permanent location in a disk using protocol provided by low weight user communication (LUC) library. During this process, the second trace buffer is used for trace collection. This helps continue flow of data with minimal disruption to the application.

5.2 Monitor Kernel and Memory Operation Interception

The process of trace collection starts with the memory operation interception when application streams try to access the registered memory. This is done with the help of user defined trap bit, which is raised high during registration process. As these memory accesses are intercepted during runtime, monitor kernel is responsible for re-executing these memory operations to completion before it passes the control back to the XMT runtime system.

When a memory exception happens, XMT saves the address and the data for the incomplete memory operation in a special kind of buffer which are designed solely for this purpose. Monitor kernel uses information from these buffers to record traces information and to re-execute the required memory operations. It is very important that monitor kernel make no unnecessary changes in any of the operations, memory and registers to preserve the correctness and the behavior of an application.

5.3 Trace Memory Hierarchy and Rates

Cray XMT has a distributed shared memory. This means that any processor can access the memory located anywhere in the system. However, number of cycles it takes to access memory at different nodes varies. Local memory accesses are approximately 100 cycles and farthest remote memory accesses are 1000 cycles. Nodes in between can take anywhere between 100 to 1000 cycles. Figure 5.3 shows the number of cycles taken to access local memory and the remote memory in the network.
Since accessing remote memory can be up to 10 times more expensive than accessing local memory, MODA framework uses local memory for trace collection before they are moved to permanent storage. This is very important to reduce the overhead of the framework each time a user defined exception is raised.

**Figure 5.4:** MODA Trace Format

5.4 Locality Exploiting Message Format

Every time a user defined exception is raised, MODA framework records memory address, time stamp, processor identifier, stream identifier and program counter.
Recording all of them in its original format would mean storing five different trace messages. This can significantly jeopardize the memory space as well as performance since these are all stored in memory. To solve this issue, monitor kernel uses compressed message format that takes advantage of locality of access. When memory closer to each other are accessed one after another, there are only few bits differences in memory address, timestamp and PC. In such case, monitor kernel tries to store minimal number of bits from which it can recreate all the information later (Contribution 2).

The idea here is to divide memory address, timestamp and PC into two halves. Upper bits are then combined to form upper bits message. Similarly lower bits are combined to form lower bits message. Upper message consist of 16 bits of timestamp, 10 bits of upper address bits and 22 upper PC bits. Lower message consist of 13 bits of timestamp, 38 lower address bits and 10 lower PC bits as shown in figure 5.4. Inside the monitor kernel, upper bits are recorded only if they are different from the previously recorded upper bits. Lower bits are always recorded. In order to make the sorting easier when these messages are recreated and sorted during post processing, every time upper bits are recorded, monitor kernel also records 64 bits of timestamp. So, in the worst case three messages are recorded and only one message is recorded in most cases.

When traces are moved from trace buffer to collection buffer, traces from different streams for each processor are combined before moving them to a permanent storage. In such case in order to differentiate messages from different streams a tag format that contains processor identifier and stream identifier is used. All these messages use upper 3 bits as a marker to identify the different kind of messages to help with the reconstruction of messages during post processing.
Chapter 6

IMPLEMENTATION: CRAY XMT

This chapter explains the implementation details involved during the Instrumentation and the Monitoring Phase. Registering and unregistering memory references, creating reconstruction streams, allocating local memory spaces, initiating exception and the intricacy involved in restoring the state of an execution back to normal are all architectural specific details. To provide clear understanding about MODA implementation, this chapter introduces the Cray XMT architecture and the implementation of the different MODA features.

6.1 Introduction to the Cray XMT

Cray XMT is a highly parallel system with 128 processors connected to each other with high speed Cray XT network forming a 3D torus. Each physical processor can supports 128 instruction streams at a time. Instruction streams are programmed like a Wide-instruction RISC processor and can hold up to three operations at a time.

Memory in Cray XMT is distributed equally across the network. Each processor has 8GB of memory that is local to the memory subsystem. It can also access remote or global memory that is in a different memory subsystem. Such globally accessible memory makes Cray XMT shared distributed memory system. The hardware provides a mechanism for fine grain synchronization between processors and the streams [11, 12].
Cray XMT does not have a cache and locality is non-existent. All memory allocations are randomized by the hardware, making it a perfect architecture for irregular applications. It exhibits large amount of concurrency and is an ideal fit for developing performance analysis tool at large enough scale. This way application can be tested at small scale with enough parallelism where analysis is easier before it roll out into production.

6.2 Instrumentation Phase

The Instrumentation Phase is responsible for registration/deregistration of memory references of interest, creating reconstruction streams for trace movement and allocation of local memory space for each processor in the system to store memory traces. Registering of the static and global variables are done using ELF reader during the Instrumentation Phase. For dynamic variables, registration is done with the help of wrapper functions, which call normal malloc function and register them right after allocation. During registration process, the user-defined trap bit for the memory address is set high. Doing so raises a user defined exception, every time that variable is accessed. The process of registration is little more than setting the user-defined trap bit high. When the trap bit is set high, XMT system assumes the address is a forwarded address, which means that the original data in that memory address has to be saved somewhere else, and has to be pointed to this new location. In addition to that data, state information attached to that memory address has to be moved to the new location as well. For example, if memory address A is registered, the trap bit for A is set high and it’s data is replaced by an address of new memory location B which now saves A’s original data and state information. Deregistration is a reverse process in which data and states are reset to original state.

The programming model in Cray XMT provides an API that let the user decide on
number of teams and streams to be used during runtime. Number of teams is equivalent to number of processors as there can be only one team per processor. For each team created in the system, there can be up to 128 streams running concurrently. In order to make sure that there are dedicated streams that can read and move the traces as they are recorded, the framework creates 16 reconstruction streams for each team. These streams loop in the buffer space dedicated for storing traces and move data from trace buffer to collection buffer and finally to the hard disk.

Local memory for each processors are created using mmap during initialization. In our current implementation, the framework creates 256 MB of memory for each processor. These are shared by all streams within a single team. Each stream gets 8 KB of local memory that it can use as trace buffer. As a note, these allocated memories are contiguous which enable monitor kernel to calculate the correct location using the offset from the fixed starting memory address. Since there are 128 streams, in total 1 MB of memory space in used for trace buffers and the remaining 255 MB is used as collection buffers which are divided into two equal halves to provide double buffering feature as explain in chapter 4.

6.2.1 ELF Reader and Compiler Modifications for Sampling

MODA framework uses ELF reader to read the elf format object files. It traverses through different sections and uses symbol table entry to find out the assigned address for the symbol being searched. Using this technique the framework creates static dictionary, which is a file containing global and static variables and their assigned addresses. This information is used for registration process. In addition to this, ELF reader and symbol tables are used to find assigned addresses for primary trap handler, monitoring kernel and the dummy space created to make a twin version of the code. Addresses for primary trap handler and monitoring kernel are used to insert the trampoline codes during binary rewriting. The address for dummy spaces is used to create a twin version of the code used for temporal sampling.
The framework provides the feature of temporal sampling because of the tremendous amount of data the tool produces. This feature is made possible with the help of some modifications done on Cray XMT compiler. By default code generated for memory operations can be with or without the control fields. With control field, one can control if the memory address is a forwarding address or if it has synchronization construct or if any of the traps are enabled or disabled. With modified compiler, the generated memory operations always have control fields attached to it, which enable us to have control over how memory is accessed. The disadvantage of this approach is that it can increase the code size to some extent. However, comparing the original and the modified code showed minimal difference in code size.

During the process of binary rewriting, the control fields are modified in the duplicate region of the code to disable user defined trap. When a code executes code from duplicate region during runtime, there is no monitoring of memory accesses. This enables the framework to do temporal sampling.

### 6.2.2 Scripts and Wrappers: An End to End Solution

Phases in MODA framework can be divided into front-end and back-end. Front-end includes the Instrumentation and the Monitoring Phases, whereas back-end includes the Analysis and the Visualization Phases. Front-end and back-end can be run as separate entity, however it is very important to merge them into a single unit to make the software user friendly. For this, scripts that wrap all phases of MODA have been developed. The scripts are written in Perl and provide end-to-end solution. The script set path for compiler, java and MODA. It then makes sure all the protocols needed for data storage and transfer are in place. Once everything is confirmed, the script goes through different phases of MODA and provides the visualized output if everything is successful. In case of failure, it provides an error file that points to what went wrong.
6.3 Monitoring Phase

The Monitoring Phase is the core of MODA framework. This phase runs in parallel with the application and records memory traces. Since the performance of monitoring kernel directly affects the performance of the tool, the entire code for the kernel has been written in assembly (*Contribution 3*).

When a user-defined exception is raised, the framework hands the control over to the monitoring kernel. The kernel starts by saving values from eight general-purpose registers (GPR) into eight special registers provided by Cray XMT for context switching. Throughout the kernel, these eight registers are used to provide temporal sampling trampolines, to create traces, re-execute interrupted memory operations and to check for exception that can rise inside the kernel.

In order to provide temporal sampling feature, the kernel checks if the exception initiated is an instruction count exception. These kind of exceptions are raised when the instruction count decrements from one to zero. As described in 4.2, MODA framework uses the count for number of instructions executed to switch between original and the duplicate version of the code. If the raised exception is not an instruction count exception, the kernel restores all registers back to its original state and hands the control back to the primary trap handler. However, if the raised exception is an instruction count exception, the kernel checks where in the code the exception was raised. If the exception was raised in original code, it modifies the target registers accordingly, so it can jump back to the duplicate version of the code and vice versa. As for note, there can be multiple kinds of exception being raised concurrently. In such case, the kernel still has to take care of additional exceptions if exist, before jumping back to the application.

The kernel uses memory address, program counter and time stamp to create memory traces in a compressed format. Every time an exception is raised, information
such as memory address and data for failed memory operations are stored in a predetermined space in the memory by XMT runtime. This feature helps the runtime to find out which memory operations failed and the data associated with them. This information can be used by runtime to re-execute the failed operations. The monitoring kernel reads from these locations and combines them with time stamp and program counter, which is read from a special register to create memory traces. For note, Cray XMT allows up to eight different memory operations per processor in flight at any time. This means that in the worst case, there can be eight interrupted memory operations per processor. All interrupted memory addresses and data are stored in memory as we mentioned before. The monitoring kernel goes in a loop to create traces for every interrupted memory operations.

Once memory traces are recorded, monitoring kernel is also responsible for re-executing all interrupted memory operations to completion. This is done by using the address and data information saved in the memory. In order to guarantee that the operation is successful, the kernel checks the status to confirm its success. In case of failure, the kernel updates registers accordingly to let the runtime primary trap handler know that those memory operations need to be taken care of. In case the memory operation is successful and is a store operation, all necessary steps of storing are taken care of automatically. However, if the successful memory operation is a load, the kernel is responsible to find out which register is supposed to get the loaded value and save data in respective registers. This is done using jump table once the register number is known.

When all interrupted memory operations are taken care of, the monitoring kernel restores each register back to its original state. It then loads the address and hands the control back to the runtime primary trap handler. All the steps performed in monitoring kernel are shown in figure 6.1.
Figure 6.1: Monitoring Kernel Steps
6.3.1 Primary Trap Handler and the MODA Trampoline

In Cray XMT runtime, every time an exception is raised, a control is handed over to the runtime primary trap handler. The job of a primary trap handler is to check what kind of exceptions are raised and take care of them accordingly. In this process, all exceptions are re-executed to completion and the control is handed back to the application.

Hijacking of the runtime system is done using binary rewriting and creating a trampolines. First few lines of runtime primary trap handler are overwritten to include an address of monitoring kernel. This enables the framework to hijack the runtime and record traces.

6.3.1.1 Synchronous Local Buffers and Asynchronous Local Buffers

Monitoring kernel stores memory traces in the trace buffer. Since the size of trace buffer is on 8 KB, it can fill up very quickly. Once the end of the trace buffer is hit, the kernel stores memory traces at the beginning of the buffer. In other words, the kernel goes through all location in the buffer in a round robin fashion. In order to make sure, no memory traces are overwritten, monitoring kernel used full/empty bits to store and load from these buffers. This helps to guarantee that no traces are lost.

Asynchronous local buffers are used to collect traces for all streams in a given processor. In order to identify the processor and stream, a header is attached before moving the data from synchronous local buffer into this buffer. The header format is shown in figure 5.4. The collected traces are later dumped to the disk when they hit the predetermined threshold.
6.3.1.2 The XMT Message Format

MODA framework uses trace format as explained in section 5.4 to record memory references. Traces are created by simply bit-shifting and bit-masking memory address, timing and PC information. Upper bits from a previous exception are saved in the memory so that they can be compared with the current one to see if there are any changes. In case the upper bits vary from the previous recorded one, they are updated in the memory and are also stored in the trace buffer. However, if the upper bits match the previous one, only lower bits are recorded.

Upper and Lower bits have 3 bits marker that are used to indentify different kind of messages. Every time an upper bits message is recorded, it is followed by 64 bits time stamp. This information is useful while combining messages from different processors during post processing. It is followed by lower bits message. If multiple user-defined exceptions are raised at the same time, multiple lower bits are recorded. In such cases, since program counter and timestamp are same for all messages, only addresses are recorded for multiple trace messages.

6.3.1.3 Twin Region Selection and Modification

During the execution of the application, twin region of the code is used to provide temporal sampling. This region is created by making a copy of the entire binary. Modifications done on compiler enable the framework to modify control field of memory operations in twin section of the code to disable user defined exception.

6.4 Runtime Localization

Cray XMT uses a special section in the memory to store runtime information during the execution of the primary trap handler. This special memory also includes a linklist of stack, which are used by different streams running in parallel. By default this memory is global which means accessing it can take up to 1000 cycles. In order
to reduce the unnecessary cycles, a linklist of local stack are created. This helped MODA framework cut the overhead to half its original overhead.
Chapter 7

RESULTS

In order to showcase MODA, we used two examples: Matrix Multiply and Breadth-First-Search. For each example, we showed memory access counts for memory subsystem, variable, and over time. With the collective information provided by these charts, we show how MODA can guide a user to possible optimization opportunities.

Hardware Testbed

Current version of MODA framework has been implemented on Cray XMT supercomputer. The system used for testing our framework has 128 processors, each equipped with 128 hardware streams capable of running in Simultaneous multi-threaded fashion at 500 MHz frequency. The processors are connected in a 3-D torus with Cray XT network. Each processor has 8 GB of local memory, which are accessible throughout the system forming a terabyte of distributed shared memory. Each memory bank has 512 MB of space, which means that there are 2x14 memory banks associated with each processor and 2x21 total memory banks in the entire system. The system doesn’t have data cache and locality is non-existent.

Software Testbed

On the software side, we chose two applications in order to show MODA capabilities. They are naive implementations of matrix multiply and the breadth-first-search...
(BFS) from the Graph500 benchmark suite. These two applications are very different in terms of their memory access behavior. Matrix multiply is a well-known application that exhibits regular access pattern. On the other hand breadth-first-search is known for its irregular nature. These applications are compiled using XMT C compiler version 6.5.0 with all default parallelization features enabled. Next section presents the overhead of the tool, especially the overhead incurred by the monitoring kernel during tracing.

**MODA Overhead**

In XMT, local memory accesses take 100 cycles in average and remote accesses take around 800 cycles in average. However, as explained in chapter 4 section 4.2, during registration MODA framework creates a indirection for all instrumented memory references. In such case, as shown in table 4.3, if no memory references are traced, MODA still exhibits 2x overhead. This means that memory references that are instrumented during registration but not traced will take around 200 cycles if it’s local and around 1600 cycles if it’s remote. However, if every instrumented memory references are traced, the overhead at current state is 45x. The overhead, however can be reduced with code compaction in monitoring kernel.

**Case Studies: Matrix Multiply**

The first example we chose for our experiments is matrix-multiply. This example was chosen not only to show MODA’s capabilities, but also to validate MODA to ensure its correctness. Since matrix multiplication is a regular application and memory access pattern is easy to predict, this was also our first step towards tool validation. The matrix size chosen is 128 by 128 and they were run with processors ranging from 1 to 32. This also verifies the capability of MODA to run in parallel. Showing the memory access counts peaks in different subsystem can help us understand where the contention is. Having information about time frame at which those peaks occurred
Figure 7.1: Variable’s access count for Matrix Multiply and symbols or variables responsible for those contentions can help programmer to optimize an application with some simple tweaks.

Figure 7.2(a) shows the memory access counts (Y-axis) for different memory subsystem (X-axis). The distribution of memory accesses in this case is uniform. This is because XMT randomizes its memory allocation to avoid contention in the network. In such case, when a memory is allocated it attempts to make uniform distribution all over the memory subsystem as shown in chapter 3 figure 3.3. In addition, in matrix multiply the accesses to these matrices are done in contiguous and uniform fashion. The results show complete agreement with the expected results.

Figure 7.3(a) shows the memory access count over time in second. First peak in the chart is created due to the memory accesses during initialization phase when matrix A, B and C are given certain values. As the computation start, the access counts are uniform as expected. The final peak represents the accesses caused by
Figure 7.2: Memory Access Counts in Memory Subsystem for Matrix-Multiplication
Figure 7.3: Memory Access Counts over time for Matrix-Multiplication
the verification code to ensure the correctness of the matrix multiply code.

Figure 7.1 shows the variable access count, represented by DYN_A, DYN_B and DYN_C. Access counts for variable DYN_A and DYN_B are equal in number since there are equal number of read accesses to those matrices. On the other hand, access to DYN_C is greatly reduced. This is because of the optimization done to restore the value in a register temporarily until the result is ready to be stored in the memory. Figures 7.3(b), 7.3(c), 7.3(d), 7.3(e), 7.3(f) show memory access counts over time for processors 2, 4, 8, 16 and 32. These access counts over time domain show the pyramid shape, as the number of processors is incremented. This behavior is because of the increased parallelism as we add the number of processors. The access count starts slowly but increases rapidly as more data is available for computation. When most of the computation approaches completion, access count gets reduced because of the data starvation, as resources stay idle.

Similarly, figures 7.2(b), 7.2(c), 7.2(d), 7.2(e) and 7.2(f) show memory access counts for memory subsystem for processors 2, 4, 8, 16 and 32. The behavior is similar to the one shown in figure 7.2(a). This is again because of the scrambling done by the XMT system.

The memory access counts for different variables stay the same for any number of processors since the accesses to these references don’t change with the increased parallelism.

**Case Studies: Graph 500**

The second example we chose for our experimentation is a famous breadth-first-search (BFS) from Graph 500 benchmark suite. In this algorithm, the search starts from a root node and it traverses through different children nodes in the graph until it finds the search element. BFS is well known for its irregular nature, inhomogeneous sparsity and connectivity, which makes it difficult to predict. In such case, the
Figure 7.4: Memory Access Counts for Breadth-First-Search optimization can be very difficult. Although XMT is designed for irregular applications like BFS, there can still be contentions causing bottlenecks that can degrade the performance.

Figure 7.4(a) shows the memory access count across the memory subsystem in XMT. The accesses in the case of BFS are highly irregular. Multiple peaks can be seen at different memory banks i.e. 14, 43, 95 and 101. This result is very different from the result seen in the case of matrix multiply. Here, the access pattern are irregular and it’s very difficult to predict the access behavior.

Figure 7.5 shows the memory access count for different variables used in BFS code. The access counts for variables vary significantly. From the figure we can see variables `prng_state_store` and `xadj` have much higher access count than the others. This is because these variables are pointers to the block of memory that is accessed multiple times inside the kernel. In such case, it is much efficient to save these addresses in the registers so we don’t have to access the memory all the time. However, the compiler was not able to do so causing contention at these locations. Just with a simple modification in the code, where these variables are stored in the registers temporarily, we were able to reduce the number of accesses for `xadj` variable from 290,000 to 4000. This optimization also reduced bank contention in bank 43 where
Figure 7.5: Access Count per Variables/32P

this variable was residing.
Finally, figure 7.4(b) shows the memory access count over time. It starts with some high peak where graph is generated and initialized. It is then followed by graph randomization where we see a huge plateau. As the BFS traversal and verification code get executed, memory access counts shows randomness in terms of accesses. There are multiple small peaks and lows. The randomness gets denser with increased number of processors. This is because of the synchronization overhead between multiple parallel streams. In addition, the compiler was not able to take advantage of optimization opportunities like loop collapse which hurt the performance.
Chapter 8

COMPARISON

Discussion about different types of tools and some of the closely related tools that do memory analysis have already been introduced in our background section (see chapter 2). Here we compare our tool and contributions to couple of existing tools. The first one is METRIC, a performance analysis tool that can do memory analysis and memory hierarchy simulation. The second is the framework called PIN, which lets the programmer build custom tool. One of the possible custom options is to build a tool that can collect memory traces.

METRIC [45] is a analysis tool that uses partial memory traces to do memory hierarchy simulation. It uses binary rewriting for instrumentation. Unlike MODA, METRIC doesn’t claim to do parallel trace collection. It uses SEQUITUR [48] compression algorithm, which shows benefits on nested loop to compress its traces. Binary rewriting scheme used for instrumentation also allows METRIC to only sample traces as user require during runtime. In contrast, MODA uses binary rewriting to hijack the runtime system and short-circuit paths to collect traces and to provide sampling. Moreover, MODA’s monitoring kernel is written entirely in assembly to provide maximum performance and our analysis goes beyond virtual address space.

PIN [43] is a framework that uses binary rewriting and allows user to produce custom tools. Using PIN framework, user can collect memory traces. However, synchronization mechanisms are required to keep track of processor and thread
identifiers, which makes the collection sequential. One important feature that PIN framework provides is its ability to attach and detach a process as desired. Using this feature, user can sample their traces. Compression algorithm can be included to compress the traces as well. The framework binary rewrites on the fly to make such features possible. The collected memory traces from PIN can be used for simulation and have been used for multiple memory hierarchy studies [37]. However, they don’t provide glimpse on contention in the network or the memory subsystem. In contrast MODA framework is designed to collect memory traces and provide analysis that can pinpoint the cause of bottleneck.

As stated previously, most of the existing analysis tools are control centric. In contrast, MODA provides a memory centric approach of analysis that can provide better insight on memory usage and contentions that can be very useful in this multicore era.
Chapter 9

CONCLUSION AND FUTURE WORK

The current version of MODA framework is capable of generating memory traces that can reveal the memory access pattern with minimal perturbation in application behavior. The Instrumentation Phase gets the framework ready for trace collection. The Monitoring Phase exploits different optimizing opportunities such as local memory usage, compressed trace format, sampling and usage of synchronization construct to provide high-speed parallel trace collection. In the post-processing phase, the framework uses efficient and parallel encoding/decoding scheme. Encoded traces are used by the GUI interface to present visual introspection of the analysis. In the current state, the tool provides end-to-end solution for users to do correct analysis. In addition, there are multiple future directions that can be used for tool enhancement as well as architectural research. They are:

1. Feature that can point section of code responsible for bottlenecks. This way user can focus on optimizing the section responsible for performance degradation.
2. Creating partial MODA traces to allow user to analyze partial traces instead to having to trace the entire application.
3. Dynamic MODA that will give the user more freedom to select partial section to trace.
4. Additional features on GUI interface to provide the user the flexibility to zoom in and out of the histogram charts, stack histogram to figure out the variables involved, and automatic detection of inconsistencies.
5. Further enhancement of Monitoring kernel with code compaction.

6. Porting of MODA to other architectures. Implementation for Tilera [59, 58, 57] is currently on the way. Implementation presented on this thesis will serve as a basis for future implementation.

7. Information provided by MODA about memory contention can be used for study to perform thread coalescing, thread splitting and data movement to dynamically reduce contention in memory as well as the network.

8. MODA can be used for further study to perform thread and data aggregation to perform collective percolation among threads to provide mutual benefit to the neighboring threads. Information about memory accesses in the physical address space can help the thread groups to perform altruistic data movement.

Summarizing, this thesis presents MODA’s capabilities and it’s implementation details. In the current state, MODA can trace memory references that analyze and pinpoint contention issues down to the memory bank level. The kind of resource centric performance analysis tool can be very useful in an era where multicore is advancing rapidly. In this thesis, we focused on the Monitoring Phase of MODA framework, which is the core of MODA design. We showed different issues with monitoring kernel and the approach we took to make this framework possible.
BIBLIOGRAPHY


