MEMORY STATE FLOW ANALYSIS AND ITS APPLICATION

by

Xiaomi An

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

Winter 2011

MEMORY STATE FLOW ANALYSIS AND ITS APPLICATION

by

Xiaomi An

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: _____
Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: _____
Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGEMENTS

First of all, I wish to thank my advisor Prof. Guang R. Gao, who gave me the chance to study in CAPSL and always supported me in my research. The experiences in CAPSL broadened my views in the whole field of computer science and I believe this will benefit me for my whole life.

I particularly thank Dr. Vugranam Sreedhar, who is my mentor and has been giving me selfless help both mentally and technically all along.

I thank all CAPSL members, including Juergen, Joseph, Sunil, Chen Chen, Xiaoxuan, JushuaSu, Tom, Stephane, Josh L, Aaron, Lucas, Mark, Robert, Daniel, Elkin, Cloudia, Gan, Xu, Handong, etc. Because of you, I had a happy time in the big family of CAPSL.

During my thesis writing, Joseph helped me to understand XMT architecture and its programming model; Juergen always encouraged me when I felt frustrated and tired; Twin Josh made a great effort to improve my English writing; and Mark helped me to acquire an account on Cray XMT supercomputer which is very important for my research. I won't be able to finish my thesis without your help.

I thank Peggy Gao, although we only met two times, her sincereness and enthusiasm impressed me and encouraged me to have a good attitude toward life.

I also thank my dear friends, Yuanqu, Xiaoting, Qunhui, etc. Your accompany helped me get through those tough days.

Finally, I thank my spouse Xiaoning, who always supported my work and gave me great patience.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The Cray XMT supercomputer system is the third generation of the Cray MTA supercomputer architecture. It is a scalable massively multithreaded platform which is based on the Cray XT infrastructure and uses the Cray massively parallel processing (MPP) system design. The XMT system uses Threadstorm processors which have a global shared memory.

A very interesting feature of XMT is that it provides a fine-grain data-centric synchronization for managing concurrency by extending each memory word with tag bits. A tag bit can be in one of two states: full or empty. Cray XMT supports a number of synchronized memory operations to read from and write to tagged memory. A synchronized read/write operation can be suspended if a corresponding memory cell is not in an expected state. For example, a synchronized memory operation can be suspended if it reads from an empty memory address or writes to a full memory address. If in a parallel program a suspended operation can never get to an expected memory state, the operation will be suspended forever. We say that the program/operation is "deadlocked".

Although the synchronized operations provide extreme flexibility to implement fine-grain parallelization of both regular and irregular applications, it is very easy to get programs into deadlock. And due to the lack of parallel program debugging tools and runtime support to detect such program deadlocks, it is hard to locate this kind of synchronization errors. So it is important to develop an effective static analysis algorithm which can detect them as early as possible. The typestate

analysis is a static program verification technique which can identify illegal operations performed on some objects due to the wrong object state. However, almost all the previous works on typestate analysis only deal with sequential programs and cannot work for concurrent programs.

In this thesis we present a new analysis technique, called memory state flow analysis to determine whether a given parallel program with synchronized read/write operations will deadlock. We extend the classical static single assignment (SSA) form with memory state information (called MSSA form) and use that to compute the memory state of each program variable at certain program points. Based on the MSSA form, we perform memory state verification to determine whether a synchronized read or write operation will ever be deadlocked. Our approach can also handle pointer variables and array sections.

The main contributions of our work are: (1) a framework to do memory state analysis is built, which catches the main features related to the memory state of parallel programs. (2) a memory state verification method to detect a potential program deadlock is developed. Thus, the essentially exponential-complex problem is decoupled and solved using dataflow analysis with simple heuristics, and most program deadlocks problems can be caught by careful heuristics design.

We have implemented our analysis using the Open64 compiler, and we present some preliminary results. The preliminary results show that our memory state flow analysis can detect most of the program deadlock problems. For example we can detect program deadlocks due to unbalanced synchronized operation types, defective operation order, etc.

We also include a survey of related work, such as static program verification for concurrent systems, program representation and dataflow analysis for parallel programs, typestate analysis, etc. A conclusion and a survey of future work are also presented.

Chapter 1

INTRODUCTION

In this chapter, we firstly give a brief introduction to the Cray XMT supercomputer's fine-grain programming model. Such a model requires the programmer to focus on data and the dependencies between data. Next, we explain the synchronization problems that can occur using fine-grain synchronization. Lastly, we illustrate our main contributions to detect these problems using static dataflow analysis based on static single assignment (SSA) form.

1.1 Data Centric Programming Model on Cray XMT

Computer architects and designers have been exploring the massively multicore and supercomputer architecture areas with the hope of improving execution time of large-scale scientific and server applications for many years. Over the past few years, the underlying programming model for such architectures has also been evolving. Parallel programming models such as OpenMP[41] and Pthreads have been two popular choices for these architectures. Both of them focus on providing thread-centric parallelism. In general, a thread is a unit work a processor can do. In thread-centric parallelism, multiple units of work or threads can be run in parallel. As such, a programmer needs to focus on coordination among multiple threads when developing software based on these programming models.

The Cray XMT[1] supercomputer is massively multithreaded machine with a global shared memory, which is especially tailored for developing large-scale irregular applications. These irregular applications are usually organized around pointer

based data structures such as trees and graphs with random access patterns to their memory. Cray XMT architecture supports a data-centric parallel programming model where programmers focus data dependences when developing applications.

In more detail, the hardware fine-grain synchronization model provided by the XMT is a simple bit extension of memory to indicate the state of data. Every word in the memory is extended with a *tag bit*. The tag bit can be either *full* or *empty*. [1] This allows for *synchronized* read and write operations to be supported. For example, a synchronized read operation can read from a memory address if the tag bit is *full*. Once it is read, the tag bit state can change to *empty* (known as `readfe` operation) or leave as *full* (known as `readff` operation). Similarly, a synchronized write operation can write to a memory address if the tag bit is *empty*. Once it is written, the tag bit state is changed to *full* (known as `writeef` operation).

1.2 Fine-Grain Synchronization Problem

A synchronized operation may be suspended when the expected tag bit is not satisfied. For example, the `readfe` operation will be suspended when it reads from memory word whose tag bit state is *empty* until a `writeef` operation changes its state to *full*. Similarly the `writeef` operation will be suspended when it writes to memory word whose tag bit state is *full* until a `readfe` operation changes its state to *empty*. One difficulty with data-centric parallel programming model is determining whether synchronized memory operations respect the underlying data dependencies of the program. In other words, if programmers are not cautious in using synchronized operations, certain synchronized operations will be suspended for ever; we call this kind of synchronization problem a "deadlock".

To further illustrate the problem, consider the following code. In the codes below, `readfe` is a synchronized read operation, `writeef` is a synchronized write

[1] Cray XMT has two bits with each word and the second bit is used for tracking other kinds of states.

operation, and `purge` is a synchronized operation that will reset the tag bit state of a memory address to *empty*. In this classical example of consumer/producer problem, thread 1 places a 10 in x. And threads 2 and 3 both compete to consume the resource. Only one readfe operation will complete, and consume the full bit leaving it empty for the other thread. Thus, either thread 2 or thread 3 will be stuck depending on which thread starts first, since the there is only one `writeef` operation which "produces" a *full* memory state while there are two `readfe` operations which "consume" the *full* memory state.

```
int x ; // shared variable
purge(&x) ;
cobegin { // creates parallel sections
  section { // thread1
      writeef(&x, 10) ;
  }
  section { // thread2
    int y // local variable
    y = readfe(&x) ;
  }
  section { // thread3
    int z // local variable
    z = readfe(&x) ;
  }
}
```

Furthermore, the presence of pointer, array elements, and concurrency makes the analysis even more complex for a programmer. Now let us consider the following example. In the code below, `readff` is a synchronized read operation. The writing of array a is conditionally dependent on the values of another array c. Hence,

the `readff` operation for each read of element of a may be deadlocked unless each corresponding element of c is greater than 0.

```
for (i = 0; i < RANK; i++)
  purge(a[i]);

// conditionally initialize the arrays
for (j = 0; j < RANK; j++) {
  if(c[j] > 0)
    writeef(&a[j], 1.0);
}
// parallel forall loop
forall (i = 0; i < RANK; i++) {
  ... = readff(&a[i]);
}
```

To highlight the impact of aliasing on the precision of memory state analysis, consider the following snippet of code: Aliasing here indicates one memory location can have multiple names. For instance f is an alias for x and also and alias for y.

```
s1: int x=0, y=0 ; // x and y initialized
s2: purge(&x) ; purge(&y) ; // x[empty], y[empty]
s3: writeef(&x, 10); // x is full
s4: f = &x ; // f->x[full], y[empty]
s5: while(?) { // f->x[bottom] f->y[bottom]
s6:   z = readff(f) ; // f->x[bottom], f->y[bottom]
s7:   if(?) {
s8:     z=readfe(*f) ; // f->x[bottom], f->y[bottom]
s9:     writeef(&y, 10) ; // f->x[empty], f->y[full]
```

```
s10     f = &y; // f->x[bottom], f->y[full]
s11:   } // f->x[bottom], f->y[bottom]
s12: } // f->x[bottom], f->y[bottom]
```

One approach for dealing with aliases is to first perform alias analysis and then perform memory state analysis. The result of a two-phase analysis is shown above in the comment section. At statement s8, since `f` can point to either `x` or `y`, and `y` can be either *full* or *empty*, the memory state of `*f` is made `bottom`. After carefully examining the statements at s6 and s8, we can see that `*f` should never be empty. We will show how to obtain a more precise result using our memory state analysis that is based on static single assignment (SSA) form.

1.3 Contributions

In this thesis we describe a new memory state flow analysis (MSFA) to determine the memory state that a variable can be at each program point. We then use the result of MSFA to determine whether a program may be "deadlocked". A simple parallel programming model is used that consists of two basic kinds of parallel constructs: `cobegin/coend` for parallel sections and `forall` for parallel loops. We will use an extended static single assignment form to precisely compute MSFA for program sections where sequential semantics are respected.

The main contributions of our work are: (1) a framework to do memory state analysis is designed, which take in consideration the main features related to the memory state of variables in parallel programs; (2) a memory state verification method based on simple heuristics to detect the potential program deadlock is developed; (3) our analysis has been implemented in the Open64 compiler and primary experiments are performed which show that a wide types of program deadlock problems can be detected by our framework.

As far as we know, this work is the first to perform memory state analysis for synchronized memory operation using data flow analysis method. Our approach is good because it essentially takes an exponentially complex problem, decouples and solves it with dataflow analysis and simple heuristics. For the most part, many program deadlock problems can be caught by careful heuristic design.

1.4 Synopsis

In Chapter 2, we will give a survey about the high performance architectures which support fine-grained synchronization based on *full/empty* bits implementation in hardware. In Chapter 3, we will introduce our memory state flow analysis. The main techniques in both program representation and data flow analysis method will be illustrated. In Chapter 4, we will show how we implement our analysis based on Open64 compiler as well as the empirical results. We will also introduce some real applications which have been successfully ported to XMT system and show how the synchronized operations are used in their parallelization. In Chapter 5, we will illustrate how to apply our MSFA for parallel programs with single-assignment semantics. Chapter 6 includes a survey of related work, such as static program verification techniques for concurrent systems, program representation and dataflow analysis for parallel programs, typestate analysis, etc. In Chapter 7, we will draw a conclusion of our work and give a survey of future work.

Chapter 2

BACKGROUND

In this chapter, a survey is provided on high performance architectures that provide full/empty bits based fine-grain synchronization in hardware. The programming model for each architecture is also provided to better understand the architectures strengths and weaknesses when programming. The Cray XMT, Tera MTA, and Cyclops SSB systems are focused on although there are many other architectures with hardware support for fine-grain synchronization. Finally, a brief introduction to the static single assignment form is given.

2.1 Full/Empty Bits based Fine-Grain Synchronization and Data-centric Program Model

The granularity of parallelism is highly dependent on the performance of synchronization mechanisms. Fine-grain synchronization with a data-centric programming model is attracting more and more attention in the field of high performance computing these days due to limited parallelism of coarse-grain synchronization. Thus, a very fine-grain model is needed to extract as much parallelism as possible. In such a model, each memory word can be lock and unlocked. This allows for point to point synchronization with granularity at the single memory word level using read and write operations.

However, the overhead of fine-grain synchronization is higher than coarse-grain synchronization due to increase in synchronization operations. To achieve good performance, hardware support is needed to implement fast synchronization. For

example, full/empty bits have been supported on many architectures. Applications ported to these architectures have achieved very good performance and scalability due to the abundant parallelism extracted.

2.1.1 Full/Empty Bits

Full/Empty bits are a kind of hardware bits which are supported by hardware as tags to mark the status of memory(e.g. full or empty) to support word-level fine-grain synchronization. Many high performance architectures have such support, e.g. HEP, Tera, MDP, Alewife, M-Machine and Cray XMT. Based on the full/empty bits, a producer-consumer style synchronization can be implemented, e.g. a synchronized read waits for full state (wait for memory state to be produced) and will reset the bit to empty after the read finishes (consume the memory state); and a synchronized write waits for empty state (wait for memory state to be consumed) and will reset the bit to full after the write finishes (produce the memory state).

Usually, in the whole memory system each word has their own associated full/empty bits like in Tera MTA and Cray XMT; while on Cyclops, a synchronization state buffer(SSB) is implemented to cache the synchronization data. We will introduce the Tera MTA, Cray XMT and Cyclops SSB architectures and the programming model supported by them in the following sections.

2.2 Tera MTA System and Programming Model

The Tera MTA ("Multi-Threaded Architecture") is a revolutionary commercial supercomputer. Compared with other parallel architectures, the Tera MTA can effectively use high amounts of parallelism on a single processor. By running multiple threads on a single processor, it can tolerate memory latency and keep the processor saturated. It can also achieve performance benefit from running on

multiple processors, when the computation is sufficiently large[59]. The MTA architecture is applicable to a wide spectrum of problems, for example, applications that do not vectorize well due to too much scalar computation or conditional statements.

2.2.1 Tera MTA System Overview

The Tera MTA system has the following features[60]:

- Shared memory

  Each processor has uniform access to every location of a shared memory. There is no memory hierarchy consideration, such as a data cache.

- High bandwidth Network

  The high-bandwidth network avoids unexpected memory-traffic bottlenecks. Each processor can issue a memory operation at each clock cycle without risk of possible network or memory congestion.

- Multithreading and lookahead

  The Multithreading and lookahead are supported which provide latency tolerance. At each cycle, a processor issues one instruction from any of the many ready threads. Each memory operation is associated with a lookahead number which tells how many more instructions it may execute from the same thread prior to the completion of the memory operation. Thus, even greater latency tolerance can be allowed.

- Memory state bits

  Four state bits are associated with each memory word: a forwarding bit, a full-empty bit, and two data-trap bits. The full-empty bit and one data-trap bit can be used for lightweight synchronization.

- Wide instructions

  Every instruction consists of up to three operations that can include one memory operation and a pair of arithmetic operations.

The Tera MTA comes with a sophisticated compiler which can automatically parallelize sequential code by decomposing it into threads. Two other tools, Traceview and Canal, are also provided which allow the programmer to profile performance and to understand the program implementation. The Traceview analyzes the execution trace and illustrates how well the executable uses the available hardware. The Canal can provide an annotated version of the source file which can help programmer get to know how the program has been decomposed into threads and targeted to the hardware.

2.2.2 Programming on Tera MTA

The Compiler is responsible for exploiting parallelism existing in the programs. Pragmas which give hint of the available parallelism inside the loops can be utilized by the compiler to perform more aggressive parallelization. For example, the codes below give an example of parallel implementation of a sort algorithm:

```
for (i = 0; i < key_value_bound; i++)
  count[i] = 0;
for (i = 0; i < nkeys; i++)
  count[key[i]]++;

start[0] = 0;
for (i = 1; i < key_value_bound; i++)
  start[i] = start[i - 1] + count[i - 1];

start$ = (sync int *) start;
```

```
#pragma tera assert parallel
for (i = 0; i < nkeys; i++)
  rank[i] = start$[key[i]]++;
```

Barriers will be inserted by the compiler to separate the four main loops. The first three loops can be automatically parallelized by the compiler. However we had to add an assert to indicate that the last loop can be parallelized.

The variable start$ is declared as a pointer to sync int, and accesses to the sync variable should be carried out atomically, so that no other iteration can possibly cause a race condition. The atomic update can be performed using a fetch-and-add operation provided by the Tera MTA system.

Other synchronized read and write primitives are also provided, like purge, readff, readfe, writeef, etc[30].

2.3 Cray XMT System and Programming Model

The Cray XMT supercomputer system is a scalable, massively multithreaded platform with a globally shared memory architecture, which is based on the Cray XT infrastructure and uses the Cray massively parallel processing (MPP) system design. The XMT system has the following features[1]:

- Performs large-scale data analysis.
- Uses Cray Threadstorm processors. Each processor is directly connected to a dedicated Cray SeaStar2 interconnect chip, resulting in a high-bandwidth, low-latency network characteristic.
- Scales from 16 to 512 processors providing over half a million threads, using 4 terabytes of system memory.

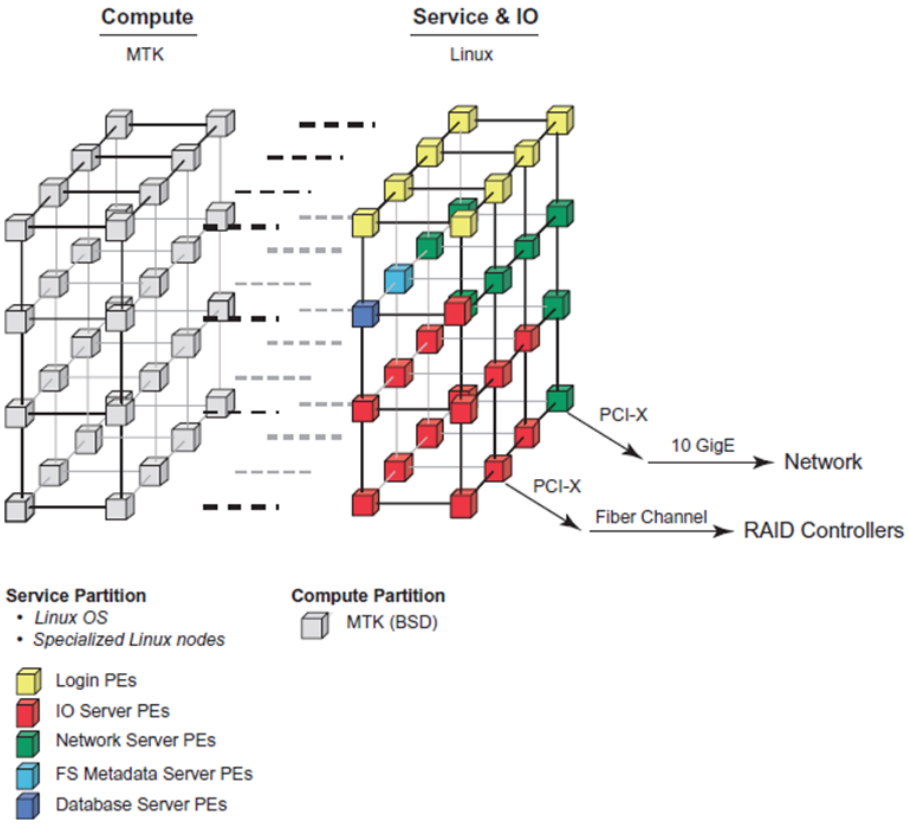


Figure 2.1: Cray XMT Hardware System Architecture

- Contains separately dedicated compute, service, and I/O nodes. Service nodes have AMD Opteron processors and can be configured for I/O, login, network, or system functions. Compute nodes have Threadstorm processors.

- Runs the Cray Linux Environment (CLE) operating system which distributes a multithreaded kernel (MTK) to the compute blades and standard Linux on the service and I/O blades. This enables the compute nodes to focus on the application without being hampered by system administrative functions.

These features enable an efficient support of fine-grain parallelism.

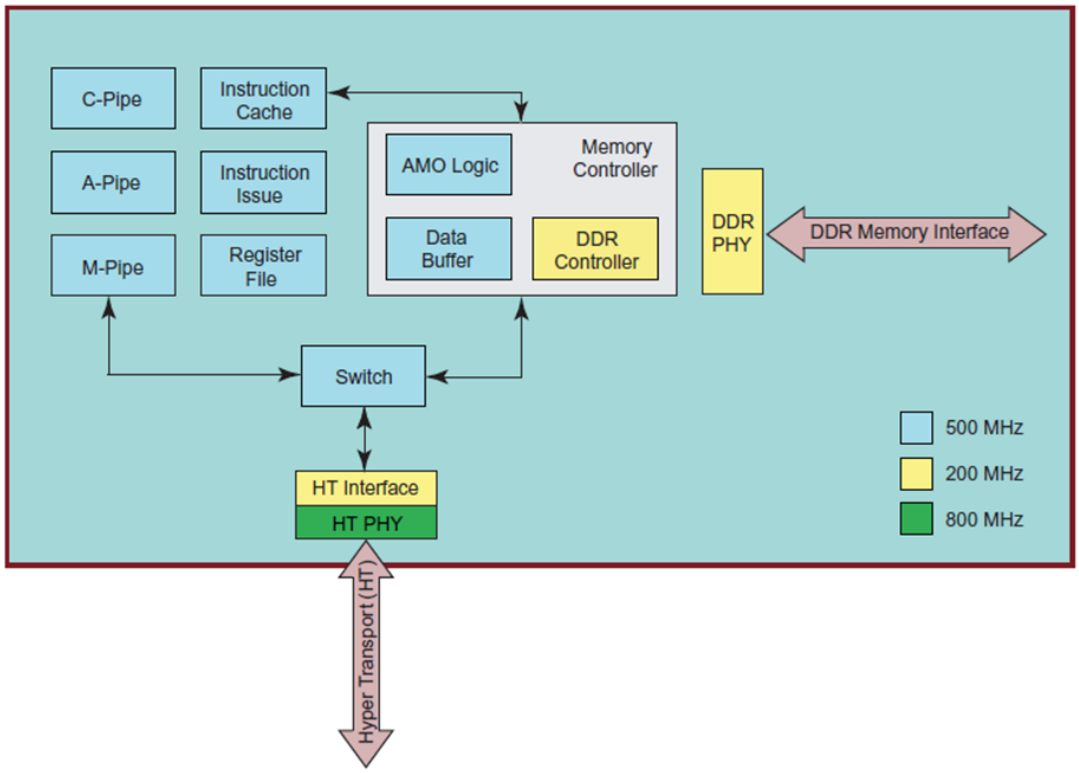


Figure 2.2: Threadstorm Processor Architecture

2.3.1 Hardware and Software Overview

The Cray XMT system includes both compute nodes and service nodes. Each node is a logical grouping of a processor, memory, and a data routing resource. The Compute nodes run application programs, each of which consists of a Threadstorm processor, DIMM memory, and a Cray SeaStar2 chip. Service nodes handle support functions such as user login, I/O, and network management. Each service node contains an Opteron processor, DIMM memory, and a SeaStar2 chip. Figure 2.1 shows a conceptual view of 3-D torus network topology for compute and service nodes[1]. We can see that the services nodes are classified, according to their functions, into login nodes, I/O service nodes,network service nodes, data base service nodes, etc.

Figure 2.2 shows the Threadstorm processor architecture which is included in each compute node. Threadstorm processor is a multithreaded processor that can support as much as 128 streams with 31 general purpose 64-bit registers, 8 target registers. A stream is the hardware unit used to execute a single thread. There are three functional units in the Threadstorm processor: a M unit which issues memory operation, an A unit which executes arithmetic operation, and a C unit which executes control or simple arithmetic operation. The Threadstorm ISA is a kind of large instruction word (LIW) where each instruction can specify up to three operations, one for each functional unit. Besides the instruction execution logic, it also includes DDR memory controller, data cache, HyperTransport(HT) interface, and a switch which connect the components[1].

The Cray XMT software system is optimized for applications that have fine-grain synchronization requirements, large processor counts, and significant communication requirements. The software stack is shown in the Figure 2.3. The stack on the left applies to the service nodes, and the stack on the right applies to compute nodes. We can see that the compute nodes and service nodes run different operating systems: the compute nodes run the MTK operating system and the service nodes run the SUSE LINUX operating system. The MTK OS is monolithic running as a single instance across all the compute nodes on the XMT system; this is different from other Cray systems where the operating system runs independently on each compute node. Because there is a single instance, MTK provides better support for fast context switching and fine-grain parallelism. The user environment is similar to the environment on a typical Linux workstation, including a development environment that provides compilers, libraries, parallel programming models, debuggers, and performance measurement tools[1].

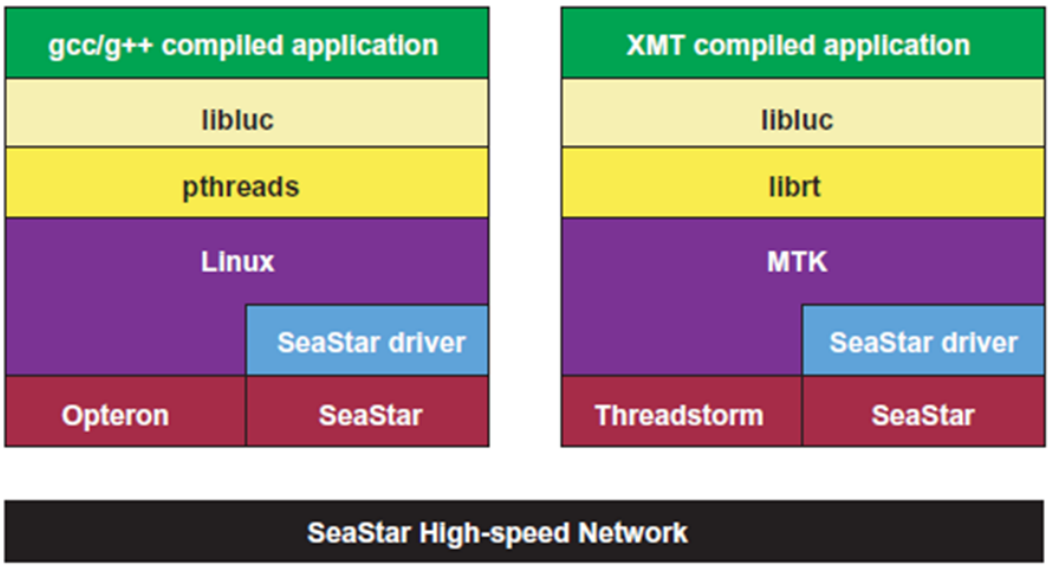


Figure 2.3: Cray XMT Software Stack

2.3.2 Fine-Grain Parallelism and Generic functions

The Cray XMT provides rich and efficient hardware level concurrency and synchronization, e.g. the single-cycle context switching enables processors to switch among multiple threads without involving the operation system, and the lightweight synchronization operations can access the shared memory by using full/empty bit without invoking the operating system. These features enable a fine-grain data-centric parallel programming model. The XMT represents one multithreading extreme, similar to pure data-flow machines. Its peculiar fine-grain thread management techniques make it an ideal candidate to process applications with irregular memory access patterns. Examples of applications with these features include

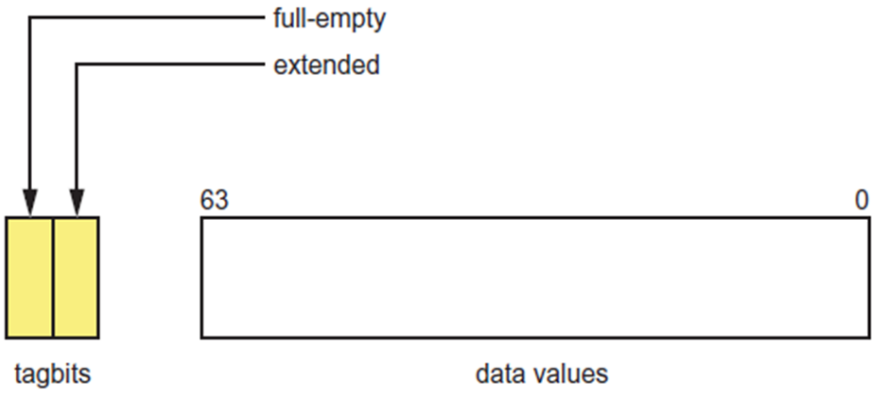


Figure 2.4: Data Word with Tag Bits

Graph Analysis, social network, and triadic analysis. For these applications, irregular memory accesses cannot be predicted statically at programming or compile time, therefore the Cray XMT is well-suited to for these types of applications.

The Cray XMT compiler provides a number of generic functions which perform read and write, purge, touch, and int_fetch_add operations on variables. Generic functions frequently affect, or have behavior that is dependent upon, the full-empty state of the variable. Below gives a parallel dataflow algorithm of three-point wavefront stencil which used the generic functions to perform synchronized memory access operations like `purge`, `readff` and `writeef`. the semantics of these generic functions will be illustrated later in this section .

```
for (i = 0; i < RANK; i++)
  for (j = 0; j < RANK; j++)
    purge(a[i, j]);
```

```
for (j = 0; j < RANK; j++) {
  a[0,j] = 1.0; a[j,0] = 1.0;
}

#pragma mta assert parallel
#pragma mta interleave schedule
for (i = 1; i < RANK; i++) {
  for (j = 1; j < RANK; j++) {
    double N = readff(a[i-1] + j);
    double W = readff(a[i ] + j - 1);
    double NW = readff(a[i-1] + j - 1);
    double V = (N + W + NW) / 3.0;
    writeef(a[i, j], V);
  }
}
```

Generic write functions write new values to variables, depending upon the full-empty state of the variable. The following generic write functions are supported on XMT[3]:

- `writeef`(&v, value)

  Writes value in variable v when v is in an empty state and sets v to a full state. If v is in a full state, the write operation is blocked until v changes to an empty state.

- `writeff`(&v, value)

  Writes value in variable v when v is in a full state and leaves v in a full state. If v is in an empty state, the write is blocked until v changes to a full state.

- `writexf`(&v, value)

  Writes value in variable v and sets v to a full state.

- `int_fetch_add`(&v, i)

  Atomically adds integer i to the value at address v, stores the sum at v, and returns the original value from v (setting v to a full state).

- `purge`(&v)

  Writes 0, using the appropriate data type, to variable v and sets v to an empty state.

Generic read functions return the value of a variable, depending upon the full-empty state of the variable. The following generic read functions are supported on XMT[3]:

- `readfe`(&v)

  Returns the value of variable v when v is in a full state and sets v to an empty state.If v is in an empty state, the read operation is blocked until v changes to a full state.

- `readff`(&v)

  Returns the value of variable v when v is in a full state and leaves v in a full state. If v is in an empty state, the read operation is blocked until v changes to a full state.

- `readxx`(&v)

  Returns the value of variable v but does not interact with the full-empty memory state.

- `touch`(&v)

  The touch function returns the value of future variable v, where v is associated with a future statement that has been spawned, but whose body may

or may not have already begun execution. If the future body that writes v has not begun executing, the thread calling touch executes the future body. If the future body associated with v is currently being executed or has finished executing, touch(&v) acts like a `readff`(&v) function.

However, it is very easy to have synchronization errors if the generic functions are not used properly. And any deadlocks that can occur at runtime cannot be detected by the current Cray compiler.

2.3.3 Implicit and Explicit Parallelism

There are two kinds of programming models supported by the Cray XMT: implicit parallelism and explicit parallelism. The implicit parallelism is expressed as a loop using the same loop constructs that are part of most every standard programming language. The compiler must be able to understand these loop constructs, exploit, and implement the parallelism existing in them. Usually the number of iterations contained in a loop must be determined before the loop begins.

Cray XMT compiler can auto-parallelize the following three types of loops[3]:

- Loops with independent iterations
- Linear recurrences
- Reductions

An example of linear recurrence loop is the following:

```
for (i = 1; i < n; i++) {
  x[i] = x[i - 1] + m;
}
```

The compiler can identify that the above loop is equivalent to the codes below and parallelize it.

```
for (i = 1; i < n; i++) {
  x[i] = x[0] + i * m;
}
```

The compiler may also rely on directives to parallelize some loops. For example, the three-point wavefront stencil uses pragmas to help compiler identify parallel loop.

However, for some cases, loop parallelism cannot be used, such as searching a linked data structure or implementing a recursive algorithm in parallel. In such cases, we can use explicit parallelism. On Cray XMT, the explicit parallelism is implemented by futures. Future is a kind of construct which explicitly indicates which sections of code may execute concurrently with other sections.

The codes below show how a future structure is used to parallelize the binary search tree algorithm[3].

```
int search_tree(Tree *root, unsigned target) {
  int sum = 0;
  if (root) {
    future int left$;
    future left$(root, target) {
      return search_tree(root->llink, target);
    }
    sum = root->data == target;
    sum += search_tree(root->rlink, target);
    sum += touch(&left$);
  }
  return sum;
}
```

Figure 2.5: Data Word with Tag Bits

2.4 Cyclops SSB and Programming Model

The Cyclops SSB provides fine-grain synchronization using a novel design without the hardware expense of extending each word. The design of SSB is motivated by the a simple observation: at any instance during the parallel execution only a small fraction of memory locations are actively participating in synchronization. Based on this, a fine-grain synchronization can be implemented that records and manages the states of frequently synchronized data by hardware with only a modest cost. The SSB design has been implemented in the context of the IBM Cyclops-64 architecture[58].

2.4.1 SSB Design and Implementation

SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of frequently synchronized data units to support and accelerate word-level fine-grain synchronization.

The SSB includes a few entries, figure 2.5 shows the structure of a SSB entry. Each SSB entry consists of four parts: (1) address field that is used to determine a unique location in a memory bank, (2) thread identifier, (3) an 8-bit counter, and (4) a 4-bit field that supports up-to 16 different synchronization modes. The address bits are used as a key to search the buffer and locate the entry of the synchronized location. The remaining three fields forms the synchronization state for that memory location.

The number of SSB entries was decided by the following equations, 1) $E_b \leq M_b$ and 2) $E_b \geq S_b$, where E_b represents the number of SSB entries, M_b represents the

number of memory banks, and S_b represents the average amount of synchronizations at a memory bank.

2.4.2 SSB Programming Model

The SSB can be used to implement both mutual exclusion and satisfy read-after-write data dependencies between a large number of threads. In the case of mutual exclusion, SSB allows each memory word to be individually locked with minimal overhead. It supports various types of locks: `read lock` (shared lock), `write lock` (exclusive lock), and `recursive lock`. For data synchronization that enforces the `read-after-write` dependencies between threads, several modes of data synchronization are supported: two `single-writer-single-reader` modes, and one `single-writer-multiple-reader` mode. Thus fine-grained synchronization can be supplied to help exploit fine-grained parallelism in applications.

Below gives the detailed description of the lock/unlock operations supported by SSB:

- (RT, Value) = swlock_l(MemAddr)
  Acquire write lock for memory location MemAddr, load the content. stores the content of the memory location into Value and store the return value(success or failure) into RT;

- (RT, Value) = srlock_l(MemAddr)
  Acquire read lock for memory location MemAddr, load the content. stores the content of the memory location into Value and store the return value(success or failure) into RT;

- sunlock(MemAddr)
  Release the lock for memory location MemAddr;

- RT = sunlock_r(MemAddr)

  Release the lock for memory location MemAddr, and store the return value(success or failure) into RT.

Below gives the descriptions of Single-Writer-Single-Reader (SWSR) Data Synchronization operations:

- RT = sswrsr_w1(MemAddr, Value)

  SWSR synchronized write mode 1; write the data in Value into memory location MemAddr and store the return value (success or failure) into RT;

- (RT, Value) = sswrsr_r1(MemAddr)

  SWSR synchronized read mode 1; read from memory location MemAddr, stores the content into Value and store the return value (success or failure) into RT;

- RT = sswrsr_w2(MemAddr, Value)

  SWSR synchronized write mode 2; the semantics are similar to SWSR synchronized write mode 1;

- (RT, Value) = sswrsr_r2(MemAddr)

  SWSR synchronized read mode 2; the semantics are similar to SWSR synchronized read mode 1;

The sswsr_w1 and sswsr_r1 support a busy-wait approach which coordinates with the software and the sswsr_w2 and sswsr_r2 operations support a blocking strategy with the support of instruction-level sleep/wakeup of the underlying multi-core architecture.

Examples in the following codes show how to use the SWSR to enforce the data dependence between read and write.

```
(a) Writer

tmp = ...  // tmp is a local variable
// write tmp to shared variable data, and mark it as available
while(sswsr_w1(&data, tmp) != SUCCESS)
;

(b) Reader

// read from shared variable data to local variable tmp, if available
while(1){  // a busy-wait loop
(RT, tmp) = sswsr_r1(&data);
if(RT == SUCCESS)
break;
}
```

2.5 Static Single Assignment Form

SSA form is a type of intermediate representation in which every variable is assigned exactly once. In this section, we will give a brief introduction of SSA construction algorithm based on program control flow graph.

A control flow graph (CFG) of a program is a rooted directed graph G = (N, E, r, t), where N is the set of nodes representing basic blocks in the program, E is the set of edges representing the flow of control from one node to another node, $r \in N$ is a distinguished root node with no incoming edges, and $t \in N$ is a distinguished terminal node with no outgoing edges. We assume that all nodes in N are reachable from the start node s.

If $x \rightarrow y \in E$, then x is called the source node and y is called the destination node of the edge; and sometimes we will say that y is a successor of x, and x is a

predecessor of y. The set of all successors of a node x is denoted by Succ(x) and the set of all predecessors of x is denoted by Pred(x). A path P of length n is a sequence of edges P : $(x_0 \rightarrow x_1 \rightarrow x_n)$, where each $x_i \rightarrow x_{i+1} \in E$. We will use the notation P : x $\rightsquigarrow$ y to represent a path of length zero or more. A node x $\in$ N dominates a node y, denoted as x dom y, if and only if all paths from the start node s to y always pass through x. If x$\neq$y then x is said to strictly dominate y. If x strictly dominates y and x is the closest node to y then x is said to immediately dominate y, and is denoted as x = idom(y). For each node in the dominator tree we associate a level number that is the depth of the node from the start of the tree. We write x : level to indicate the level number of a node x (the level number of the start node is 0). Similar to dominance relation we also use the dual post-dominance relation; a node y postdominates another node x, denoted as y pdom x, if and only if all paths from the node x to end node always pass through y. Similar to dominance relation, one can dually define strict postdominance relation, immediate post-dominace relation, and postdominator tree.

The dominance frontier DF(x) of a node x is the set of all z such that x dominates a predecessor of z, but does not strictly dominating z. The DF(X) of a set of nodes X is defined as DF(X) = $\bigcup_{x \in X}$DF(x). The iterated dominance frontier IDF(X) for a set of node X is defined as a limit of the increasing sequence:

$$IDF_1(X) = DF(X)$$
$$IDF_{i+1}(X) = DF(X \cup IDF_i(X))$$

A Sparse Evaluation Graph (SEG) is a projection of a control flow graph (CFG) for a specific data flow problem. A SEG is constructed from a CFG by identifying the initial set of nodes $N_a \subseteq N_c$ that affects the data flow problem (that is, whose transfer function is a non-identity function), computing $N_\phi = IDF(N_a)$, and inserting a sparse edge $n_s \rightarrow m_s$ between any two nodes in $N_s = N_a \cup N_\phi$ if there is a path P_c from n_s to m_s in the original CFG and P_c does not contain any other

node in N_s (other than n_s and m_s at the end points of the path). For separable data flow problems, such as reaching definition or live variable analysis, more than one SEG is constructed, one for each different variable. SSA form is a special kind of SEG for representing def-use chain and in which variables are renamed and explicit ϕ-functions are introduced to ensure each variable has exactly only definition.

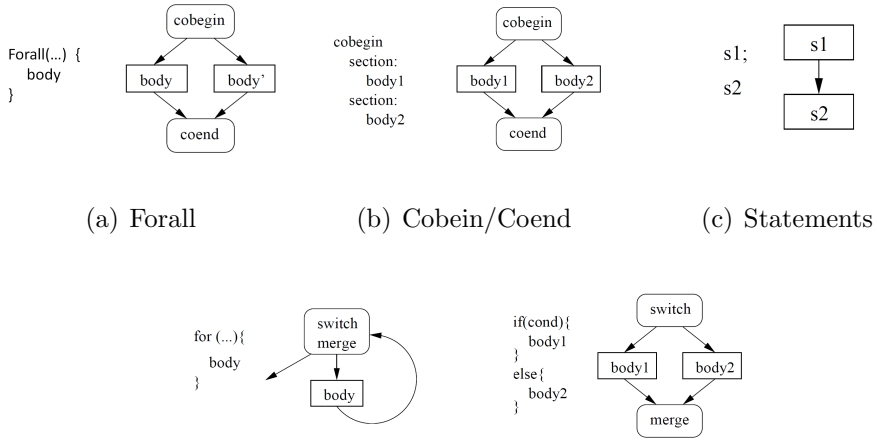
Chapter 3

MEMORY STATE FLOW ANALYSIS

Memory state flow analysis (MSFA) is a static data flow analysis to determine the memory (tag bit) state for each synchronized variable at each program points. In this chapter we firstly introduce the parallel control flow graph and the augmented SSA (ASSA) form we used to represent the parallel programs. We then present a memory state SSA (MSSA) form, which is an extension of classical SSA form augmented with memory state information, concurrency information, and array regions. Lastly, we will show how to build the MSSA form and how to perform memory state verification Based on it.

3.1 Parallel Control Flow Graph

We assume a simple parallel language which is made up of nested parallel regions. A program begins execution as a single thread called the `parent` thread(sometimes also called as the `master` thread). When a parallel region is encountered, the `parent(master)` thread generates a team of threads (`child` threads) to execute the enclosed code sections. When they complete, they synchronize and terminate, leaving only the parent thread to proceed. Parallelism is expressed using two parallel constructs: `forall` and `cobegin/coend`. When control reaches a `forall` construct, all iterations of the loop body are started and proceed concurrently, each by one thread. A `cobegin/coend` parallel region consists of a set of sections. Each section in a `cobegin/coend` region is executed concurrently with



(a) Forall (b) Cobein/Coend (c) Statements

(d) Sequential for (e) Conditional switch

Figure 3.1: PCFG Construction

other sections in the region. To simplify the presentation, we restrict the class of programs that support only structured control flows for `forall` and `cobegin/coend` one cannot jump in and out of these parallel constructs arbitrarily. We will also assume a structured program and does not contain gotos, breaks, and continue statements.

For synchronization we will follow Cray XMT synchronization reads and writes with full/empty tag bits. We will use the following synchronization read and write operations defined by Cray XMT: `readfe`, `readff`, `writeef`, `writexf`, `writeff`, `touch`,`int_fetch_add` and `purge`[3].

The set of statements in a parallel program and the control flow among them form a graph, called the parallel control flow graph(`PCFG`). Since we assume a structured parallel program, the corresponding PCFG is a structured graph. Figure 3.1 illustrates how to construct `PCFG` for each program construct. We insert control flow edges from a `cobegin` node to the first statement node of each of the parallel sections of the corresponding parallel region, and we also insert control flow edges

from each of the last statement node in the corresponding parallel section to the `coend` node. For `forall` we clone the body of the `forall` loop once and treat the loop body and its clone as two parallel sections of a `cobegin/coend` parallel region. Therefore we represent `forall` as in `cobegin/coend` parallel region. We interpret the body and its clone as being two different iterations of the parallel loop, and for our analysis purpose we do not care what those two iterations are, except when dealing with arrays which we will discuss later in the paper.

3.2 Concurrency and Exclusion

3.2.1 Concurrency Relation

Two statements s1 and s2 in a program P is said to be concurrent if there exists an execution of P such that there are two threads T1 and T2 which can execute s1 and s2 either simultaneously or mutually exclusively. Recall that at the end of `cobegin/coend` and `forall` statements all threads merge, and a barrier is typically needed to merge different threads. The barrier semantics require that either all threads in a team execute a barrier or none of them executes the barrier. The set of barriers in a program divide the parallel region into a set of phases, and each phase in the parallel region will be executed by all members of the team that belong to the same parallel region. Notice that if two statements belong to the different phases, they cannot be concurrent.

Let s and t be any two statements in a PCFG. We say that s and t are concurrent, denoted as Conc(s, t) if: (1) s and t belong to two different parallel sections of a parallel region, (2) s and t are in the same phase of the parallel region, and

The first condition is straightforward since if the two statements s and t are in the same parallel section, they cannot be concurrent. The second condition is needed because two phases of a parallel region cannot be executed concurrently. It is important to remember that our concurrency relation obtained using the above

proposition is conservative: Conc(s, t) implies that there exists an execution of the program so that two threads can execute s and t concurrently or mutually exclusively. It is possible, due to resource constraint or scheduling constraints, that s and t can be ordered. The Conc(s, t) is not transitive but is symmetric. Sometimes we will use the notation Conc(s) to denote the set of all statements (nodes) that are concurrent with s. Note that r $\in$ Conc(s) if and only if Conc(r, s).

3.2.2 Exclusion Relation

Two statements s1 and s2 in a program P is said to be exclusive to each other if they cannot co-exist in any program path. A simple example is two branches of an if statement.

Let s and t be any two statements in a PCFG. We say that s and t are exclusive, denoted as Excl(s, t) if: (1) s and t belong to the same section of a parallel region, and (2) there is no path from s to t or from t to s.

The first condition is sufficient but not necessary, however, since the exclusion relation will only be used within one parallel region, we limit s and t belong to the same section here. The second condition guaranteed that s and t will not co-exist in one program path. We use Excl(s, t) to represent s and t are exclusive to each other. Similar to concurrency relation, exclusion relation is also symmetric but not transitive. We use Excl(s) to denote the set of statements that are exclusive to s. And r $\in$ Excl(s) if and only if Excl(r, s).

3.3 Augmented SSA Form

SSA form[47] is an intermediate representation where every variable is assigned only once. At control flow merge points ϕ-functions are added to ensure that every use of a variable has exactly one definition. A ϕ-function is of the form $x_n = \phi(x_0, ..., x_{n-1})$, where $x_i (i = 0, ..., n)$ are set of variables with static single assignment. A SSA graph is a graph representation of SSA form and contains (1) a

set of SSA nodes representing definitions and uses of variables, including ϕ-nodes, and (2) a set of SSA edges that connect the definition of a variable to all uses of the variable.

For our PCFG we insert π-function at the end of the `cobegin` node and insert ψ-function at the beginning of `coend` node. The π and ψ functions serve to connect sequential and parallel regions. To represent the memory state of different array elements, we extend the traditional SSA form to handle array regions. Thus, the memory state of each array element can be defined at each program point. We will use the term Augmented SSA (ASSA) form that combines traditional SSA form, concurrent SSA form, and array region SSA form.

3.4 MSSA Form

During runtime the memory state of a variable can be either full or empty. The goal of MSFA is to determine whether a memory operation will be suspended forever due to inconsistency or errors in the memory state. To determine whether a memory operation will be suspended forever we model the flow of memory state using a finite state model. Our memory state model (MSM) consists of four states: Full (F), Empty (E), Full Wait (FW), and Empty Wait (EW).

Figure 3.2 illustrates the MSM. The state transition edges are annotated with different synchronization operations supported on Cray XMT (the semantics of these are described in the Cray XMT Programming Environment Users Guide[3]). Since readxx does not influence memory state transition and int_fetch_add is equivalent to writeff in memeory state transition, we ignore them in the MSM and in the following analysis.

To compute the memory state of a variable at each program point, we first annotate each variable v with a state variable s and s can be in one of the four states described above. We denote a state annotated variable as $v : s$ where the variable v is annotated with memory state s. We denote a state transition for a variable

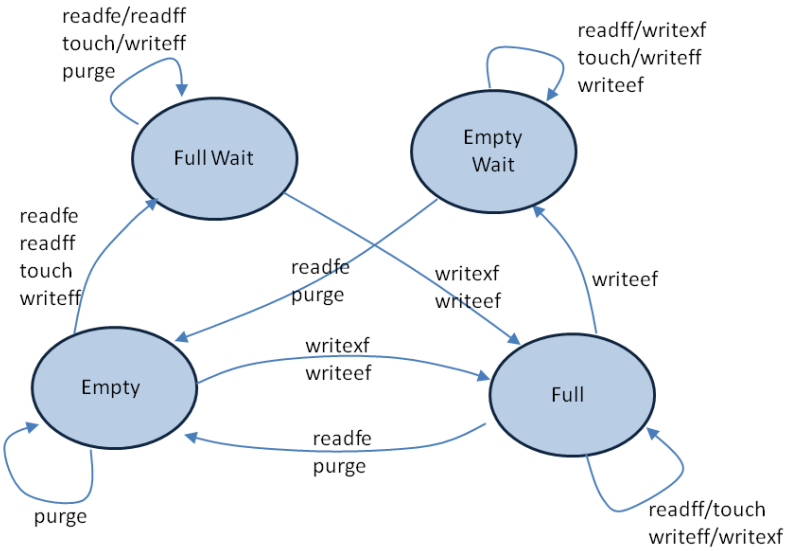


Figure 3.2: Memory State Model

v as $v : s1 \underrightarrow{o} s2$, where o is one of the synchronization operations that changes the state of v from $s1$ to $s2$. Assume that a variable v is in E state, and we perform a `readfe` operation on v, then we have v: E $\underrightarrow{readfe}$ FW. Here the state FW denotes the fact that the `readfe` operation is waiting for a write operation (`writeef` or `writexf`) to change the state of v to F. In the following analysis, we call operations with FW/EW state as suspended operation and we call operations that can change FW/EW state to F/E state as waited operation. We use Wop(o) to denote the set of waited operations for suspended operation o.

We perform MSFA on a MSSA form, where memory state information is associated for each node in ASSA form. We will propagate memory state information on the MSSA form using the lattice shown in Figure 3.3. Besides, we identify suspended operations and collect waited operation set for them as part of the memory state information.

Our MSFA consists of two steps: (1) constructing the MSSA form and (2) inferring whether any memory operation will be suspended forever.

3.5 Construction of MSSA form

At each node in the MSSA form we associate two memory state cells, MCelli and MCello to store memory state information. MCelli stores the input memory state value of a node and MCello stores the output memory state value of a node. For suspended operations, MCelli also stores their corresponding waited operations.

Each memory state cell is initialized with $\top$ lattice value. We then propagate the memory state over the MSSA form. For each synchronized operation that define a memory state s, we use the MSM shown in Figure 3.2 to compute state information. We then use the lattice model shown in Figure 3.3 to merge memory state information at ϕ-nodes and ψ-nodes. However, $\perp$ state may be generated at some program merge points, which means the memory state is unknown statically. To handle the unknown memory state, we extend the original MSM, shown in Figure 3.4. We call the new MSM as extended memory state model (EMSM), and use it in our MSSA form construction. The EMSM is consistent with the lattice and the memory state verification based on it will generate conservative results.

Below gives the MSSA Construction Algorithm. We use dst(e) to denote the destination node of an SSA edge e, and use src(e) to denote the source node of e. An SSA edge is said to be root SSA edge if src(e) has no incoming edge. The algorithm for memory state construction uses a worklist of SSA edge, MSSAWorklist. Note

$$
\begin{array}{cc}
L = \{e, f, e_w, f_w, \top, \perp\}, a \in L & \\
\top \wedge a = a & \perp \wedge a = \top \\
e \wedge f_w = f_w & f \wedge e_w = e_w \\
e \wedge f = \perp & e_w \wedge f_w = \perp \\
e \wedge e_w = \perp & f \wedge f_w = \perp
\end{array}
$$

Figure 3.3: Memory State lattice. e is the Empty state, f is the Full state, f_w is the Full Wait state, and e_w is the Empty Wait state.

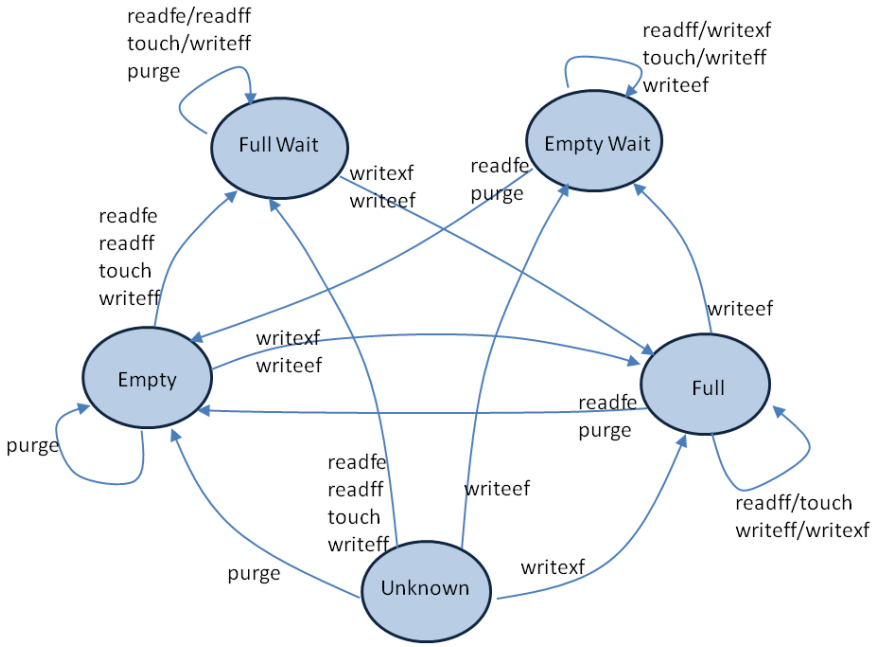


Figure 3.4: Extended Memory State Model

that for a π-node, which marks the start of parallel regions, we consider the π node as a fake synchronized operation, generating the same memory state as its input state. This memory state will be the input memory state shared for all sections of the parallel region.

`MSSA Construction Algorithm`

1. Initialize memory state in all memory cells to $\top$.
2. Initialize the worklist MSSAWorklist with root SSA edges.
3. Do the following steps until MSSAWorklist is empty.
   (a) Take an MSSA edge e from the MSSAWorklist and calculate the output memory state of dst(e),

(b) If the dst(e) is a ϕ-node, the output memory state lattice value for the ϕ-node is computed using the memory state lattice shown in Figure 3.3.

(c) If the dst(e) is a π-node, the output memory state of the π is same as its input memory state.

(d) If the dst(e) is a ψ-node, handle it in the same way we handle ϕ-node.

(e) If the dst(e) is an synchronized operation, say o, then the value of the output memory state cell is evaluated according to EMSM shown in Figure 3.4.

If the evaluated output memory state is E/F, then that is the final output memory state of o.

If the evaluated memory state is FW/EW, mark o as suspended operation and calculate Wop(o). Wop(o) include all waited operations for o existing in Conc(o). We then finish the state transition by following corresponding edges in EMSM (i.e. FW/EW $\xrightarrow{Wop(o)}$ F/E, and F/E $\xrightarrow{o}$...), and get the final output memory state of o.

Besides, If src(e) is a π-node, we mark o as suspended too, and calculate Wop(o), considering π-node as a fake sync operation and a candidate for Wop(o).

(f) Store output memory state into MCello(dst(e)); if the output memory state changes the value of MCello, then add all the outgoing SSA edges of dst(e) into MSSAWorklist.

The first two steps of the above algorithm are straightforward. In step four we propagate the memory state until a fixed point is reached. Consider the following code as an example.

```
int x ; // shared variable
purge(&x) ;
```

```
cobegin { // creates parallel sections
  section { // thread1
    if (p) {
        writeef(&x, 10) ;
    } else {
        writeef(&x, 20) ;
    }
    writeef(&x, 30) ;
  }
  section { // thread2
    int y ; // local variable
    y = readfe(&x) ;
  }
  section { // thread3
    int z ; // local variable
    z = readfe(&x) ;
  }
}
```

The MSSA form constructed for the above codes is shown in Figure 3.5. In the figure, we use MSin(o) to denote the input memory state of synchronized operation o and Msout(o) denotes the output memory state of o. The dashed lines represent SSA edge and the solid lines represent control flow edges.

The MSSAWorklist initially contains the root SSA edge starting from o1. The output memory state cell of o3, o4 and o5 contains memory state value F, and the output memory state cell of o1, o2, o6 and o7 contains E. Note that we take the π-node as a fake synchronized operation and denote it as o2. We calculated Wop(o3) and Wop(o4) since the source of their SSA edges is π node. We calculated

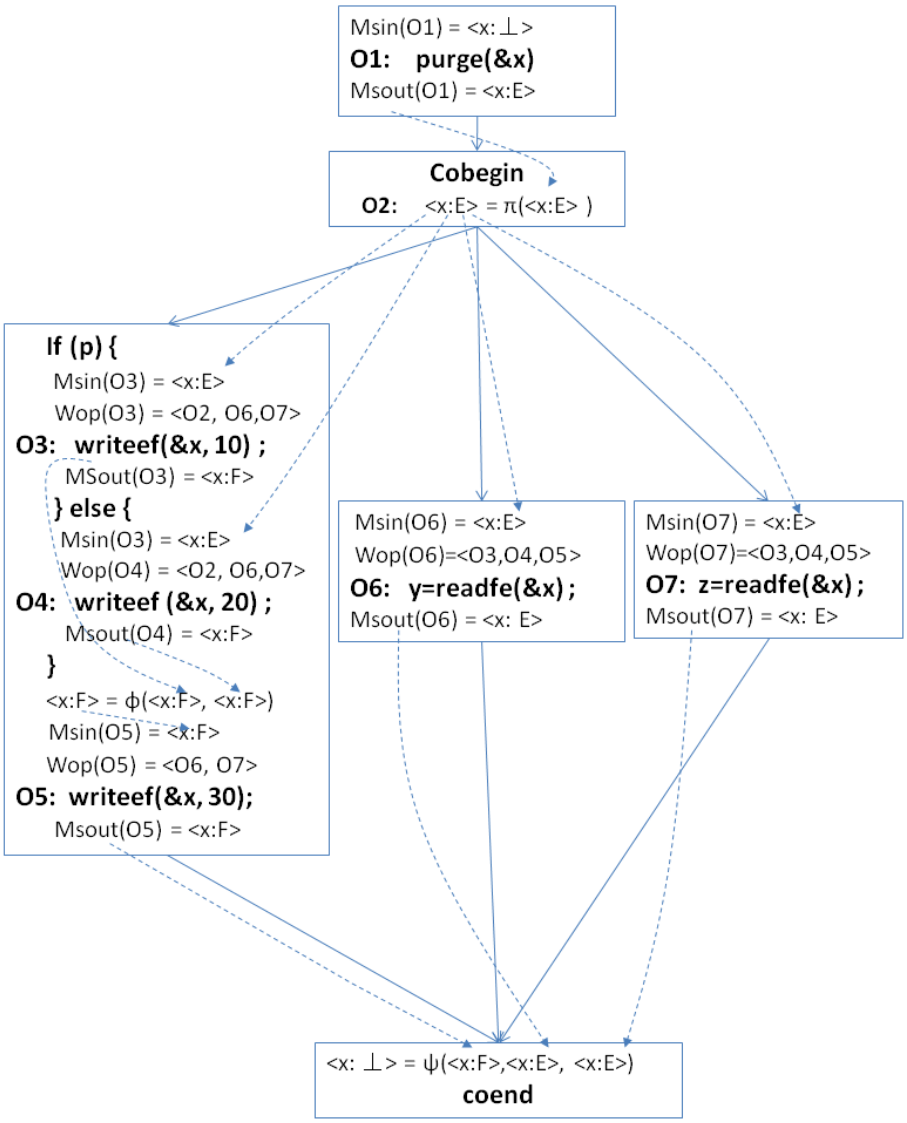


Figure 3.5: MSSA example

Wop(o5), since o5 may be suspended waiting for E state. Similarly, we calculated Wop(o6) and Wop(o7) because o6 and o7 may be suspended waiting for F state.

3.6 Memory State Verification

Given the MSSA form, a naive way to do memory state verification is to check whether the waited operation set of each suspended operation is empty. If it is empty, the program will deadlock. However, due to the uncertainty caused by the static analysis and the concurrency of multithreaded program, the naive method is not able to catch many program deadlock issues.

For example, in Figure 3.6(a), the `writeef` will be included in waited opeartion set of both the two `readfe` operations in the MSSA form, but the F state generated by `writeef` can only enable one `readfe` operation, leaving the other suspended forever. in Figure 3.6(b), the `writeef` may not be executed due to the condition, thus the `readfe` may deadlock. We call this kind of program deadlock problem "count problem". That is, the number of waited operations is not enough to generate the memory state waited by all the suspended operations, so that the program will deadlock.

Another kind of deadlock problem is called "order problem". In Figure 3.7(a), all of the four synchronized operations will deadlock, since the two `readfe`'s will wait for F state forever that can only be generated by `writeef`'s, which are dominated by the `readfe`'s themselves. Example shown in Figure 3.7(b) is similar to (a), except that the operations access different synchronization variables. In both (a) and (b), the number of waited operations is enough, but they show in a defective order in program, so that the suspended operations will deadlock.

In this paper, we apply a quantitative and a ordering analysis based on the MSSA form. The intuition is: The MSSA form of any correct parallel program (i.e. none of synchronized operations will deadlock), must satisfy the following two conditions: 1) quantitative condition; 2) ordering condition.

(a) Straight line codes

```
int a;
purge(&a);
cobegin {
  section {
    int v = foo();
    writeef(&a, v);
  }
  section {
    int u = bar1(readfe(&a)) + bar2(readfe(&a));
  }
}
```

(b) Conditional statement

```
int a;
purge(&a);
cobegin {
  section {
    int v = foo();
    if (p)
      writeef(&a, v);
  }
  section {
    int u = bar(readfe(&a));
  }
}
```

Figure 3.6: Deadlock Examples: Count Problem

Quantitative condition For any combination of k (k≥1) non-exclusive suspended operations which access the same memory location and are suspended by the same memory state (either FW or EW), the union of their corresponding non-exclusive, non-self-consumed waited operations form a set, say W, $|W| \geq k$ must be satisfied.

Ordering condition For any combination of k (k≥1) non-exclusive suspended operations which are suspended by the same memory state(either FW or EW), the union of their corresponding waited operations form a set,say W, there must exist an operation o, $o \in W$ and o is not dominated by any of the k suspended operations.

A self-consumed operation is an operation which is needed to enable(i.e. generate the memory state required by) another operation in its own section. The non-self-consumed operations can be easily identified on our MSSA form. For an waited operation o1, follow its outgoing SSA edge e: if dst(e) is a non-suspended operation o2 and MSout(o1) is the required memory state of o2, then o1 is self-consumed; if des(e) is a ϕ-node, and MSout(o1) == MSout(ϕ), follow outgoing SSA edges of the ϕ-node and check recursively. If no non-suspended operation can be found to consume MSout(o1), then o1 is a non-self-consumed operation.

A non-exclusive operation set is an operation set where no two operations are exclusive to each other. For example, in Figure 3.5, o3 and o4 are exclusive to each other. Although o3 and o4 are both included in Wop(o6), only one of them can be executed to produce a F state. We will use the exclusion relation to identify the non-exclusive operations to avoid over-counting of the number of suspended or waited operations.

Based on the above observations: we developed the heuristics to do quantitative and ordering verification for synchronized operation groups in the program. Our heuristics can detect the potential synchronization errors existing in Figure 3.6(a) and Figure 3.7(a),(b). A simple extension of our heuristic which can identify conditional statement will be able to detect the synchronization error existing in Figure 3.6(b) too.

Figure 3.8 shows the heuristic to do quantitative verification for synchronization variable v with MSSA graph G. Function Non_Self_Consumed_Set(Wop(o)) filters and returns the set of non-self-consumed operations contained in Wop(o). Function Union_Non_Exclusive(waitset, Nscset(o)) unions waitset and Nscset(o) by adding elements in Nscset(o) into waitset, and guarantees that only elements that keep the output set non-exclusive will be unioned. In function Choose_susp_for_Count(susp_list_FW, suspset, waitset), an item will be taken from

susp_list_FW which is not exclusive to any of the items in suspset, and the waited operations of the item should have been partially included in waitset. If no such item was found in susp_list_FW, return NULL.

Figure 3.9 gives the heuristic to do ordering verification given MSSA graph G. In function Choose_susp_for_Ordering(susp_list_FW, suspset, waitset), an item will be taken from susp_list_FW which is not exclusive to any of the items in suspset, and the item should dominate at least one of the elements already contained in waitset if waitset is non-empty. If no such item was found in susp_list_FW, return NULL. Function Dom(suspset, waitset) return true if each element of waitset is dominated by some element contained in suspset.

3.7 Array Region Memory State Verification

Synchronized operations can also apply to array elements; to verify the correctness of these operations, we use a triple of three array regions to represent the memory state of an array at each program point, which includes full region, empty region, and unknown region. The full region include the array elements whose memory state is full, the empty region include array elements whose memory state is empty, and the unknown region include array elements whose memory state is unknown. We use list of convex regions to represent array region. Set operations like union, intersect, difference, etc can be implemented on convex regions while maintaining good precision[48].

Array accesses usually happen inside loops, to reduce the complexity of analysis and get a better balance between efficiency and precision, we treat array access inside loops in following way:

- For `forall` loops, ignore the possible interleaved execution among different iterations and apply sequential semantics on array accesses within the loop. Thus, `forall` loop can be handled as sequential loop, and memory verification

will be straightforward after propagating memory state and can be handled as a forward dataflow problem.

- For loops inside parallel section of `cobegin/coend` construct, ignore the possible interleaved execution between loop iterations and other concurrent parallel sections. Thus, we can treat array accesses inside loops as a whole, summarize the array region accessed by the synchronized operation inside the loop and consider the operation as an aggregate operation accessing the array region, and use set operations on array regions to perform memory state verification.

To summarize the array accesses inside loops, we extended our MSSA form by adding another η-node at the end of each loop which contains synchronized operations accessing array elements.

Figure 3.10 gives an example, where loops exist inside each parallel section. We inserted η-node after each loop, summarize the array region accessed synchronized operation inside loops. We denote the aggregated operation using $AOi :< o, Ra >$, where o represents the corresponding synchronization operation inside the loop, Ra represents the array region accessed by o. Please note that the accessed the region of the aggregated π-node is a triple of three array regions, i.e. full region, empty region and unknown region.

We also insert π, *psi* nodes, propagate memory state according to the EMSM, identify suspended operations and calculate the waited operation set for them. For each suspended operation, the waited memory state and the suspended array region are also calculated in the MSSA form. We denote the suspended information for AOi using $Suspended(AOi) :< s, Rs >$, where s represents the waited memory state (either FW or EW), and Rs represents the suspended array region.

To do memory verification based on the array MSSA form, we use the following method:

For each suspended aggregate operation AO_i, with $Wop(AO_i) =< AO_{i_1}, AO_{i_2}, ..., AOi_n >$, if $\bigcup(Ra(AO_j), j = i_1, ..., i_n) \supseteq Rs(AO_i)$, then we will draw a conclusion that AO_i will not deadlock. $Ra(AO_i)$ represent the array region accessed by the aggregated operation AO_i, and $Rs(AO_i)$ represent the array region suspended on the aggregated operation AO_i

We can see that since $Ra(AO3) \not\supseteq Rs(AO4)$, AO4 will deadlock.

Although the current memory verification method is simple, it can be easily extended based on the array region MSSA form to catch more synchronization problems caused by array access.

3.8 Discussion

In our analysis, we assumed that the input programs are structured programs nested with parallel regions which can be either `forall` loop or `cobegin/coend` construct. The synchronized operations can be included in both sequential region and parallel region of the input program. After analysis, a warning will be outputted if a synchronization error is detected to be existing in the input program.

However, our analysis cannot detect all the potential synchronization errors. For example, due to the limitation of static analysis, some conditional statements make it impossible to give precise result. Due to the randomness in execution order of concurrent programs, a potential program deadlock may only appear in "some" execution path, our analysis cannot detect such problems either. Conservative analysis can report more problems while may cause more false positive at the same time. However, more careful designed heuristics can give more precise result and cause less false positive.

(a) Access same synchronization variable

```
int a;
purge(&a);
cobegin {
  section {
    int v = readfe(&a);
    writeef(&a, foo(v));
  }
  section {
    int u = readfe(&a);
    writeef(&a, bar(u));
  }
}
```

(b) Access different synchronization variables

```
int a, b;
purge(&a);
purge(&b);
cobegin {
  section {
    int v = readfe(&a);
    writeef(&b, foo(v));
  }
  section {
    int u = readfe(&b);
    writeef(&a, bar(u));
  }
}
```

Figure 3.7: Deadlock Examples: Order Problem

```
Procedure Verify_Count(VAR: v, MSSA_Graph G)
  WORKLIST susp_list_EW;
  WORKLIST susp_list_FW;
  SET suspset;
  SET waitset;
  for each suspended operation o accessing v in G {
    Nscset(o) = Non_Self_Consumed_Set(Wop(o));
    if (Nscset(o) is empty) {
      report o is potential deadlocked operation
      exit
    } else {
      if (o is suspended due to FW)
        add o into susp_list_FW
      else
        add o into susp_list_EW
    }
  }
  while(susp_list_FW is not empty) {
    suspset.clear()
    waitset.clear()
    o = Choose_susp_for_Count(susp_list_FW,
          suspset, waitset)
    while (o) {
      suspset = Union(suspset, o)
      waitset = Union_Non_Exclusive(waitset,
          Nscset(o))
      if (size(waitset) < size(suspset)) {
        report potential deadlock
        exit
      }
      o = Choose_susp_for_Count(susp_list_FW,
          suspset, waitset)
    }
  }
  while (susp_list_EW is not empty) {
    ... // similar as above
  }
```

Figure 3.8: Heuristic: Quantitative Verification

```
Procedure Verify_Order(VAR: v, MSSA_Graph G)
  WORKLIST susp_list_EW;
  WORKLIST susp_list_FW;
  SET suspset;
  SET waitset;
  for each suspended operation o in G {
    if (o is suspended due to FW)
      add o into susp_list_FW
    else
      add o into susp_list_EW
  }
  while(susp_list_FW is not empty) {
    suspset.clear()
    waitset.clear()
    o = Choose_susp_for_Ordering(sop_list_FW,
          suspset, waitset)
    while (o) {
      suspset = Union(suspset, o)
      waitset = Union(wait, Wop(o))
      if (Dom(suspset, waitset)) {
        report potential deadlock
        exit;
      }
      o = Choose_susp_for_Ordering(susp_list_FW,
          suspset, waitset)
    }
  }
  while (susp_list_EW is not empty) {
    ... // similar as above
  }
```

Figure 3.9: Heuristic: Ordering Verification

int sum = 0;
int a[100];
$MSin(AO1) =< E : \emptyset, F : \emptyset, U : [0 : 99] >$
for (i=0; i<100; i++)
  purge(&a[i]);
$AO1 :< purge : Ra = [0 : 99] >$
$MSout(AO1) =< E : [0 : 99], F : \emptyset, U : \emptyset >$
cobegin {
  $< E : [0 : 99], F : \emptyset, U : \emptyset >= \pi(MSout(AO1))$
  $AO2 :< \pi, Ra =< E : [0 : 99], F : \emptyset, U : \emptyset >>$
  section{ // Thread1
    int i;
    $MSin(AO3) =< E : [0 : 99], F : \emptyset, U : \emptyset >$
    $Suspended(AO3) :< EW : Rs = [0 : 49] >$
    $Wop(AO3) =< AO2 >$
    for (i=0; i<50; i++) {
      writeef(&a[i], foo(i));
    }
    $AO3 :< writeef : Ra = [0 : 49] >$
    $MSout(AO3) =< E : [50 : 99], F : [0 : 49], U : \emptyset >$
  }
  section{ // Thread2
    int v;
    $MSin(AO4) =< E : [0 : 99], F : \emptyset, U : \emptyset >$
    $Suspended(AO4) :< FW : Rs = [0 : 99] >$
    $Wop(AO4) =< AO3 >$
    for (i=0; i<100; i++) {
      v = readff(&a[i]);
      sum += v;
    }
    $AO4 :< readff : Ra = [0 : 99] >$
    $MSout(AO4) =< E : \emptyset, F : [0 : 99], U : \emptyset >$
  }
  $< E : \emptyset, F : [0 : 49], U : [50 : 99] >$
    $= \psi(MSout(AO3), MSout(AO4))$
}

Figure 3.10: Array MSSA Form

Chapter 4

IMPLEMENTATION AND EMPIRICAL RESULTS

In this chapter, we will show how we implement our analysis based on Open64 compiler. We will also introduce some real applications which have been successfully ported to XMT system and show how the synchronized operations are used in their parallelization. We will then give the empirical results which show the kinds of synchronization errors that can be detected by our analysis.

4.1 Implementation

Our memory state flow analysis was implemented in Open64 compiler[49]. Figure 4.1 shows the infrastructure of Open64 compiler and indicates the phase where our MSFA is implemented. Open64 compiler applies different program transformations and optimizations while lowering IR from very high level IR to very low level IR. The higher level IR keeps more program structures and semantics while the lower level IR is subject to more kinds of optimizations. There are totally three main phases in Open64 compiler: frontend, middle end and backend. All the program analysis and transformations in the middle end use WHIRL, which is a tree intermediate representation. There are four sub-phases in the middle end in Open64 compiler: Very High Optimizer(VHO), Inter-procedural Analysis and Optimization(IPA), Loop Nest Optimization(LNO), and Global(Scalar) Optimization(WOPT). Our MSFA was implemented between VHO and the IPA. The reason

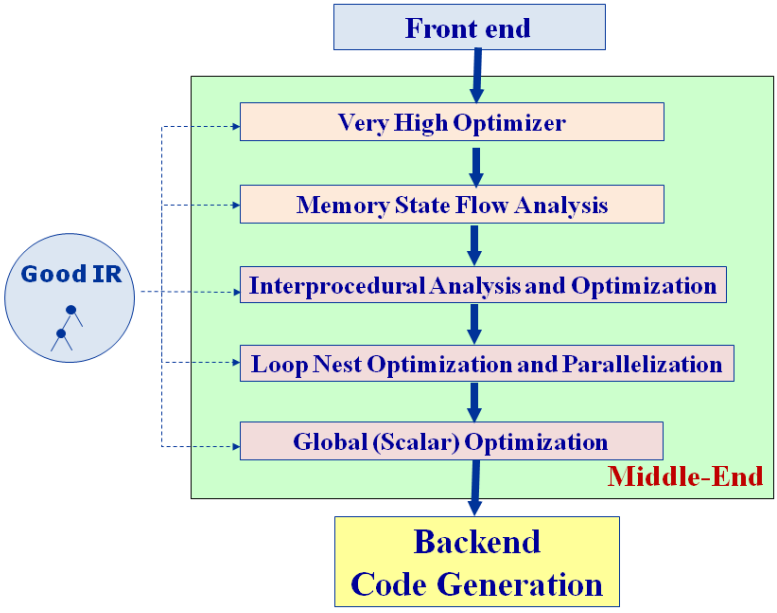


Figure 4.1: MSFA Implementation Phase

that we choose to implement our MSFA there is because, at this stage, all the structural representation of the program is still well maintained and it is easy to identify parallel constructs.

Figure 4.2 shows the flow diagram of the MSFA implementation. Firstly we build the PCFG, then construct MSSA form which includes: 1) build dominator tree; 2) inserting ϕ-nodes, π-nodes and ψ-nodes; 3) add SSA edges between definition and their uses; 4) perform concurrent analysis and calculate concurrent operation set for each synchronized operation; 5) perform exclusion analysis and calculate exclusive operation set for each synchronized operation; 6) Propagate memory state information, mark suspended synchronized operation, and calculate waited operation set for each of them. Finally we perform memory state verification based on the MSSA form. To handle array region, we also need to insert η-nodes, and perform array region analysis while propagating memory state information.

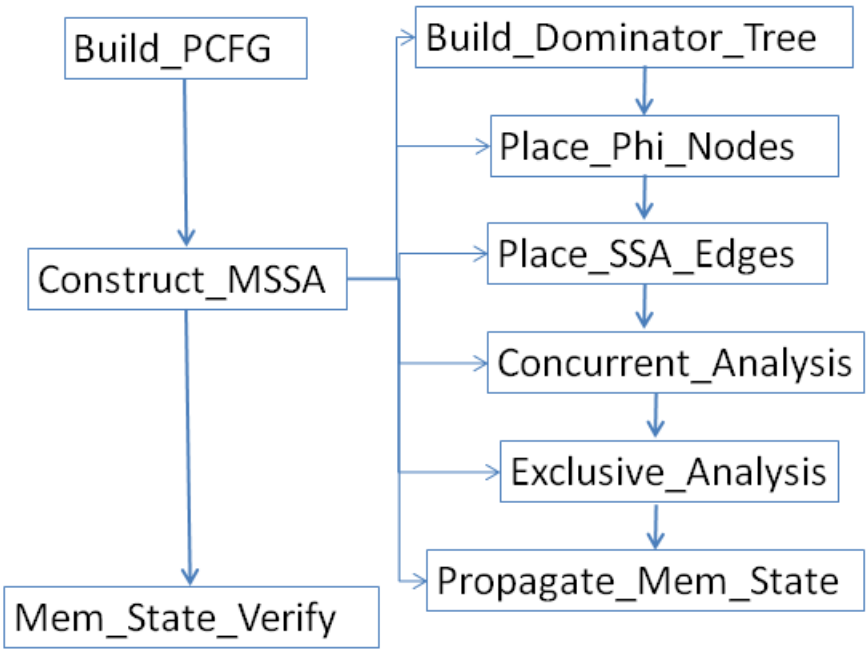


Figure 4.2: MSFA Implementation Flow Diagram

To identify the generic functions which supplied the synchronized read and write operations, we enhanced the compiler frontend to identify them as intrinsics. We build the parallel control flow graph, using region (which is a kind of compiler internal representation) to represent parallel sections. The region representation is very efficient, since the parent region and its children form a tree, with which we can easily identify whether two sections represent concurrent threads within the same parallel region.

We use bit-vector to represent the set of synchronized operations, which makes our implementation very efficient. We utilize the ARA(array region analysis) of Open64 to implement our array regions, which is based on convex region representation.

4.2 Application Introduction

We will introduce several important applications which have been ported to XMT by Prof. David Bader's group[15].

4.2.1 GraphCT

GraphCT[16] is a parallel toolkit which is capable of applying complex analysis tools to massive graphs. GraphCT supplies multithreaded implementations of known algorithms for the Cray XMT, taking advantage of the large shared memory and fine-grained synchronization of the Cray XMT. Besides, GraphCT can also run sequentially on POSIX-like platforms.

It provides an optimized library of functions including clustering coefficients, connected components, betweenness centrality, graph diameter, and distribution statistics. Functions are provided to read and write large data files in parallel using the shared memory. GraphCT uses a single common graph representation for all analysis kernels, and a straightforward API makes it easy to extend the library by implementing your own custom routines.

GraphCT makes no assumptions about the type or structure being analyzed, so that the user can choose a data representation according to the structure of the graph and the analysis to be performed. The graph is stored in compressed-sparse row (CSR) format, a common representation for sparse matrices. The number of vertices and edges is known when ingesting the data, so the size of the allocated graph is fixed.

The combination of GraphCT and the Cray XMT's massive multithreading permits exploration of graph data sets previously considered too massive.

4.2.2 STINGER

STINGER(Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation)[17, 18] is an extensible representation for streaming, dynamic networks and graphs. Linking blocks of edges in a hybrid data structure, STINGER is able to accommodate massive streams of edge insertions and deletions while simultaneously supporting fast, parallel access to neighborhood information. STINGER supports directed and undirected graphs, weighted and unweighted edges, edge and vertex types, time stamps, and can be extended with additional meta data. STINGER can be built on shared memory platforms including the Cray XMT and commodity hardware using OpenMP + atomic intrinsics.

Figure 4.3 shows the diagram of the STINGER data structure. This data structure provides a compromise between list- and array-based graph representations to support both efficient updates and efficient analysis.

STINGER takes the efficient element of CSR, stores end vertices in arrays, and loosens other requirements. STINGER also borrows from the list structure and stores edge end vertices as a list of arrays. Its linked array structure permits simple multithreaded traversal. STINGER is a compromise that permits dynamic updates while supporting a wide variety of analytical algorithms on a single copy.

The basis operations of STINGER include: read-only queries of vertices and edges, insertion and deletion both vertices and edges, aging off entities by time stamp, and checkpoint/restart.

4.2.3 SSCA2

The intent of the Scalable Synthetic Compact Applications 2 (SSCA2) benchmark[19, 20] is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a weighted, directed graph. In addition to a kernel to construct the graph from the input tuple list, there will be three additional computational kernels to operate

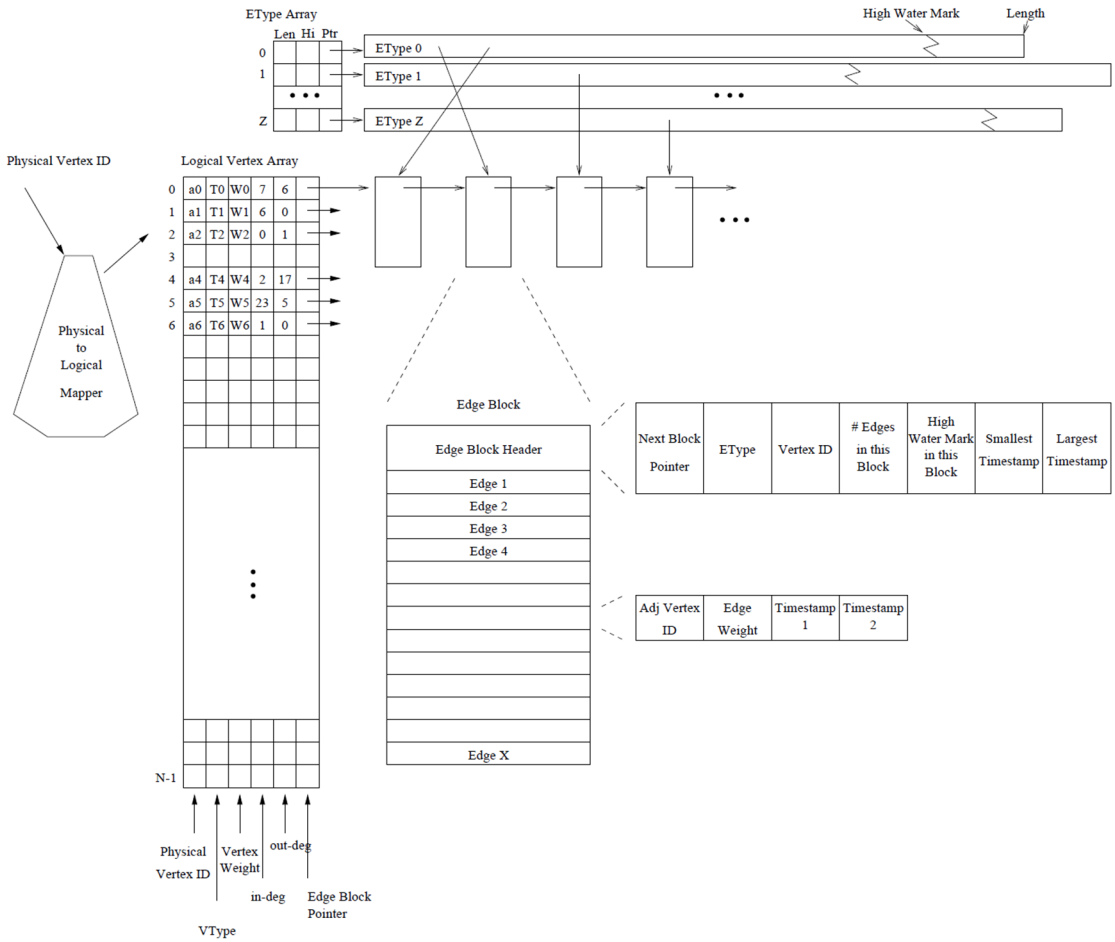


Figure 4.3: A diagram of the STINGER data structure

on the graph. Each of the kernels will require irregular access to the graphs data structure, and it is possible that no single data layout will be optimal for all four computational kernels.

The first kernel constructs the graph in a format usable by all subsequent kernels. No subsequent modifications are permitted to benefit specific kernels. The second kernel extracts edges by weight from the graph representation and forms a list of the selected edges. The third kernel extracts a series of subgraphs formed by following paths of specified length from a start set of initial vertices. The set of initial vertices are determined by kernel 2. The fourth computational kernel computes a centrality metric that identifies vertices of key importance along shortest paths of the graph.

4.3 Implementing Parallel Applications By Synchronized Operations

The synchronized read/write operations can be used to implement multiple synchronization primitives, such as lock, barrier, critical section, etc. These primitives are used prevalently in traditional thread-centric parallel programming model like openmp and pthread. Thus, the parallel applications written in traditional thread-centric parallel programming model can be easily ported to XMT with the help of synchronized read/write operations.

Besides, the tagged memory with full/empty bits support in XMT can be taken as extremely fine-grain lock, which enables us to parallelize irregular applications which are hard to parallelize using thread-centric programming model.

Below gives a few examples which illustrate how the synchronized operations are used to implement efficient parallel algorithms. All the examples come from real applications already ported to XMT.

4.3.1 Lock

In the codes below, the `readfe` and `writeef` implement lock and unlock to guarantee the mutual exclusive access to the same memory location.

```
void stinger_int64_swap (int64_t *x, int64_t *y) {
     int64_t vx, vy, t;
     vx = readfe (x);
     vy = readfe (y);
     writeef (x, vy);
     writeef (y, vx);
}
```

4.3.2 Critical Section

In the example below, the pair of `readfe` and `writeef` implement a critical section, it will have the equal semantics if we add omp critical pragma and replace the `readfe` and `writeef` with normal read/write operations.

```
static struct stinger_eb **ebpool = NULL;
static void init_ebpool (void) {
  // critical section
  {
  struct stinger_eb **new_ebpool;
  new_ebpool = readfe (&ebpool);
  if (new_ebpool) {
       writeef (&ebpool, new_ebpool);
       return;
  }

  new_ebpool = xcalloc (INIT_N_EBPOOL_PTRS, sizeof (*ebpool));
```

```
  new_ebpool[0] = xmalloc (EBPOOL_SIZE * sizeof (struct stinger_eb));
  which_ebpool = 0;
  cur_ebpool_tail = 0;
  writeef (&ebpool, new_ebpool);
  }
}
```

4.3.3 Barrier

The `purge`, `writeef` and `readff` in the following codes implement a barrier, where thread2 will be blocked until `writeef` in thread1 is finished and the two threads get synchronized.

```
int search_tree(Tree *root, unsigned target) {
  int sum = 0;
  if (root) {
    int left;
    purge(&left);
    cobegin{
      section { // thread1
        int v = search_tree(root->llink, target);
        writeef(&left, v);
      }
      section { //thread2
        sum = root->data == target;
        sum += search_tree(root->rlink, target);
        sum += readff(&left);
      }
    }
```

```
  }
  return sum;
}
```

4.3.4 Atomic Operation

The codes below use int_fetch_add to implement atomic operation to the shared variable numMarkedEdges inside the `forall` loop.

```
void getStartLists(int weight, int *weight,
       int *numMarkedEdges, int *markedEdges) {
  int i;
  *numMarkedEdges = 0;

  forall (i = 0; i < NE; i++) {
    if (weight[i] == maxWeight) {
      int k = int_fetch_add(numMarkedEdges, 1);
      markedEdges[k] = i;
    }
  }
}
```

4.3.5 Fine-Grain Lock

The `readfe` and `writeef` operations in the following example guaranteed the mutual exclusive accessing to the each array element or array D. The fine-grain lock implementation helped to achieve extremely fine-grain parallelism.

```
void floydWarshallFE(int m, int w[MAX][MAX], int d[MAX][MAX]){
  int i, k;
  forall(i=0; i<m; i++){
```

```
    int j;
    for(j=0; j<m; j++)
      d[i][j]=w[i][j];
  }
  forall(k=0; k<m; k++){
    int i;
    forall(i=0; i<m; i++){
      int j;
      forall(j=0; j<m; j++){
        int oldV, newV;
        oldV=readfe(&d[i][j]);
        newV=MIN(oldV,d[i][k]+d[k][j]);
        writeef(&d[i][j],newV);
      }
    }
  }
}
```

4.4 Empirical Results

In this section, we show how our MSFA can be used to identify several kinds of program deadlock problems.

4.4.1 Count problem: Straightline Codes

Figure 4.4 shows two examples where the program is deadlocked due to not enough waited operation count.

The deadlock in case1 is detected by quantitative verification which combines suspended operations (i.e. `readfe`) belonging to different parallel sections; while the

deadlock in case2 is detected by quantitative verification which combines suspended operations within the same section.

4.4.2 Count Problem: Conditional Statements

In Figure 4.5 (a), both `readfe` and `writeef` appear in the conditional statements, so that there is no deadlock in the program. Since our MSFA can identify that the two `readfe`'s are exclusive to each other and the so do the two `writeef`'s, it can avoid the over-counting in quantitative verification so as to avoid the false-positive.

In Figure 4.5 (b), since `writeef` appear in the conditional statements, one of the `readfe` will deadlock. This can be detected by our analysis.

4.4.3 Count Problem: More Synchronization Types

Besides `readfe` and `writeef`, there are other types of operations which do not change the memory state after execution, e.g. `readff`. Figure 4.6 gives two examples where the `readff` operations can be suspended and cause program deadlock. Our MSFA can also identify these problems by quantitative verification.

4.4.4 Order Problem: access same memory location

Figure 4.7 shows an example where all the synchronized operations will deadlock due to defective operation order. Our MSFA will detect this using ordering verification. After forming combination composed by the first `readfe`'s in both sections, we can see the all of their waited operation(i.e. `writeef`'s) are dominated by themselves.

4.4.5 Order Problem: access different memory locations

Figure 4.8 shows an example where the defective operation order happen among accesses to different memory locations. It can be detected by the same technique as those happening within accesses to the same memory location.

(a) Case1

```
int a;
purge(&a);
cobegin {
  section {
    int v1 = readfe(&a);
  }
  section {
    int v2 = readfe(&a);
  }
  section {
    int v3 = readfe(&a);
  }
  section {
    writeef(&a, foo());
  }
  section {
    writeef(&a, bar());
  }
}
```

(b) Case2

```
int a;
purge(&a);
cobegin {
  section {
    int v1 = readfe(&a);
    int v2 = readfe(&a);
    int v3 = readfe(&a);
  }
  section {
    writeef(&a, foo());
    writeef(&a, bar());
  }
}
```

Figure 4.4: Count problem: Straight line codes

(a) Case1

```
int a;
purge(&a);
cobegin {
  section {
    int v;
    if (p)
      v = readfe(&a);
    else
      v = readfe(&a);
  }
  section {
    if (p)
      writeef(&a, foo());
    else
      writeef(&a, bar());
  }
}
```

(b) Case2

```
int a;
purge(&a);
cobegin {
  section {
    int v1, v2;
    v1 = readfe(&a);
    v2 = readfe(&a);
  }
  section {
    if (p)
      writeef(&a, foo());
    else
      writeef(&a, bar());
  }
}
```

Figure 4.5: Count problem: Conditional Statements

(a) Case1

```
int a;
purge(&a);
cobegin {
  section {
    int v1 = readff(&a);
  }
  section {
    int v2 = readfe(&a);
  }
  section {
    writeef(&a, foo());
  }
}
```

(b) Case2

```
int a;
purge(&a);
cobegin {
  section {
    int v1 = readfe(&a);
    int v2 = readff(&a);
  }
  section {
    writeef(&a, foo());
  }
}
```

Figure 4.6: Count problem: More Synchronization Types

```
sync int a;
int sum;
purge(&a);
cobegin {
  section {
    int v = readfe(&a);
    writeef(&a, foo1());
    writeef(&a, bar1());
    v = readfe(&a);
  }
  section {
    int u = readfe(&a);
    writeef(&a, foo2());
    writeef(&a, bar2());
    u = readfe(&a);
  }
}
```

Figure 4.7: Order Problem: access same memory location

```
sync int a;
sync int b;
purge(&a);
purge(&b);
cobegin {
  section {
    int v = readfe(&a);
    writeef(&b, foo());
  }
  section {
    int u = readfe(&b);
    writeef(&a, bar());
  };
}
```

Figure 4.8: Order Problem: access different memory locations

Chapter 5

MEMORY STATE FLOW ANALYSIS FOR SINGLE ASSIGNED DATA STRUCTURE

Single-assignment is a primary feature of functional languages to avoid any possible side-effect and achieve parallelism. A single assigned variable can only be written once and read multiple times, so that a producer-consumer type of fine-grained synchronization can be achieved. For example, I-structure is a data structure to support parallel computing in data flow model based systems. The components of an I-structure object can only be assigned once, but can be read for many times. In I-structure, runtime check is needed to guarantee the write-once feature.

In this chapter, we will discuss how to use our memory state flow analysis (MSFA) to statically detect whether a program may be "deadlocked", when the synchronized variables are claimed to have the single-assignment feature.

5.1 Language Model

We assume the same parallel language as chapter 3, which includes two kinds of parallel constructs: `forall` loop and `cobegin/coend` construct.

To guarantee the single assignment attribute of synchronized variables, we restrict the usage of generic functions, and make the following assumptions:

- The initial memory state of all the synchronized variable is `empty`;

- The synchronized variable(including array element) can only be written by `writeef` operation when its memory state is `empty` and after that its memory state is changed to `full`;

- The synchronized variable(including array element) can be only read by `readff` operation when its memory state is `full` and its memory state is left as `full` after that;

5.2 Memory State Flow Analysis

Based on the language model, we build a new memory state model(MSM), shown in Figure 5.1, which is a simplified version of the MSM shown in Figure 3.2. The new MSM also consists of four states: Full (F), Empty (E), Full Wait (FW), and Empty Wait (EW). Here the state FW denotes that a `readff` operation is waiting for a `writeef` operation to change memory state from E to F. The state EW denotes that a `writeef` operation is waiting when the memory state is F; however, since no operation can change the memory state from F to E, it will be suspended forever. The state EW actually indicates the existence of multiple write operations to the same memory location, which is a violation of single-assignment rule.

We use the same memory state lattice as that shown in Figure 3.3. We will construct MSSA form based on the new MSM and the lattice in similar way as stated in section 3.5. We associate memory state information for each node in the SSA graph, propagate memory state of each synchronized variable, identify suspended operations, and then calculate their corresponding waited operations. Based on the MSSA form, we then perform memory state verification to infer whether an suspended synchronized operation will deadlock. The program deadlock problems are also classified into count problem and order problem, and they can be detected by the quantitative verification and ordering verification respectively, using heuristics shown in section Figure 3.8 and Figure 3.9.

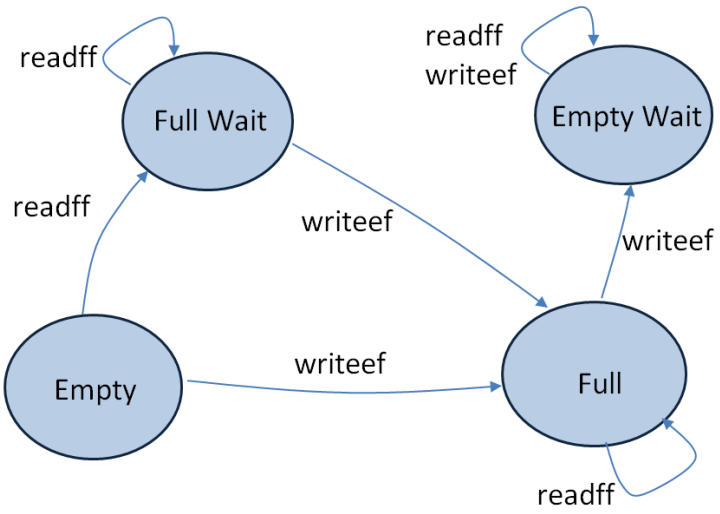


Figure 5.1: Memory State Model

For example, for the following codes, the `writeef` operation in thread2 will be deadlocked.

```
int a;
cobegin {
  section { // thread1
    int v = foo();
    writeef(&a, v);
    v = readff(&a);
  }
  section { // thread2
    int u = readff(&a);
    writeef(&a, bar(u));
  }
}
```

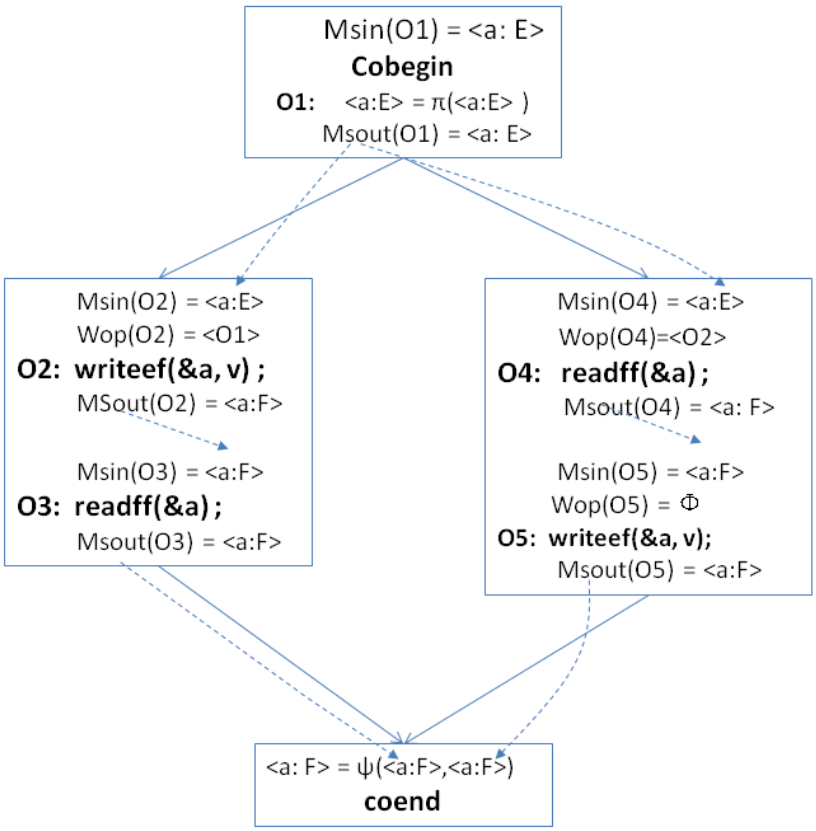


Figure 5.2: MSSA Form

Figure 5.2 shows the MSSA form of the above example. We can see that O5 (i.e. the `writeef in thread2`) is identified as a suspended operation, but its waited operation set is empty. This can be detected by our quantitative verification and the potential program deadlock can then be reported.

Chapter 6

RELATED WORK

In this chapter, we illustrate the related work in four fold, including 1) static concurrent system verification, 2) program representation and dataflow analysis techniques for concurrent systems, 3) typestate analysis, 4) I-structure and M-structure.

`Static Program verification for Concurrent Systems`

Concurrent Programs are usually hard to write and debug because of the indeterminism caused by their inherent concurrency. Program bugs can be detected either dynamically or statically. Static analysis tools which can identify program bugs automatically are of great value, since they can consider different execution paths exhaustively while incurring no runtime overhead. Static tools for finding concurrency bugs can be classified into the following types: 1) Type systems, e.g. rccjava[50] and Java atomicity types. Both of them extended the language type system with atomicity-related properties like thread-local, shared, "protected by lock", etc. 2) Program analysis tools, e.g. Warlock[51] and RacerX[52]. They use inter-procedural analysis, track the program behaviors and look for inconsistencies. 3) Model checking tools, e.g. Java Pathfinder[53] and Bandera[54]. Exhaustively testing are performed by these tools, but usually on a simplified program model.

Our work performs program memory state analysis using SSA based dataflow analysis method.

`Program Representation and Dataflow Analysis for Concurrent Systems`

There have a lot of works which perform dataflow analysis for concurrent systems based on Parallel flow graph(PFG). For example, in [8], dataflow equations are developed for explicit parallel programs, and global data flow analysis can be applied on a parallel flow graph which is built to handle parallel sections. A reach-def analysis for parallel programs is given which considered synchronization between threads. In [10], bit-vector analysis for parallel programs is presented, which can be used in multiple program optimizations, such as code motion, partial dead-code elimination,etc. Sarkar and Simons proposed a parallel program graph(PPG)[13] that subsumes program dependence graphs(PDG) and conventional control flow graph. A reaching definition analysis on PPG was developed for deterministic parallel parallel programs.

Traditional SSA form for sequential programs is also extended to represent parallel programs. E.g. a parallel static single assignment form(PSSA) was proposed by Srinivasan et al[11, 12]. PSSA was developed for PCF Parallel Fortran parallel sections construct with copy-in/copy-out semantics. Each thread receives its own copy of the shared variables at a fork point and can modify only its own local copy. However, PSSA form cannot handle parallel programs with truly shared memory semantics where the result of a parallel execution depends on particular interleaving of statements in parallel programs.

Lee and Padua proposed a CSSA[55] form based on concurrent control flow graph(CCFG) for parallel programs with cobegin/coend and parallel for construct and the post/wait synchronization mechanism. Based on that,several optimizations, like constant propagation, dead code elimination, common subexpression elimination can be extended to apply on parallel programs. And sequential consistency can be guaranteed.

Our work is also based on SSA form with memory state information

embedded, and we handle both scalar and array regions.

Typestate Analysis

Typestate analysis[21, 22, 23, 27] has been given attention as an important technique for static program verification. In this model, objects of a given type may exist in one of finite states, the operations allowed on the object depend on the state of it. And the operations may also change the object state. The goal of typestate verification is to statically determine whether the execution of a given program may cause an illegal operation performed on a object according to the state of the object. For example, whether an object is used before it is initialized, or whether a file is used after it is closed.

Research about typestate was usually disjoint from research about concurrency, while [25] tried to combine these two kinds of analysis to detect data race and atomicity violation via type state guided static analysis. Our work is another case to combine the typestate analysis and concurrent analysis to detect possible synchronized errors(program deadlock) existing in parallel programs using memory state flow analysis.

I-structure and M-structure

Both I-structure and M-structure are a nonfunctional feature introduced into a functional language. An I-structure is a data structure proposed to facilitate parallel computing[61] on dataflow model based systems. The components of an I-structure object can only be single-assigned, but can be read many times; and runtime check is used to guarantee write-once feature. An I-structure element can be in one of three states: empty, full, and deferred. Producer-consumer type of fine-grain data synchronization can be achieved by interacting with the state of an I-structure when accessing it. Unlike I-structure, which regards the redefinition

of an element as an error, the M-structure is a fully mutable data structure such that an element can be redefined repeatedly[62]. The M-structure provides implicit synchronization by using take and put operations, which guarantee the necessary serialization while avoiding loss of parallelism.

Chapter 7

CONCLUSION AND FUTURE WORK

Cray XMT provides a data-centric synchronization model where every word in the memory is extended with *tag bits* so that synchronized read and write operations are efficiently supported by hardware, and extreme fine-grain parallelism can be achieved. The synchronized read/write operations give programmers tremendous flexibility to implement parallel algorithms and achieve high performance even for irregular applications which are traditionally hard to parallelize. On the other hand, they also bring problems since it is very easy for the programmer to generate synchronization errors and introduce deadlocks into programs.

In this work, we developed MSFA(memory state flow analysis) which includes two phases. In the first phase, a MSSA form is constructed where the memory state information is associated to an ASSA(augmented SSA) form, and all the operations which may be suspended are identified. In the second phase, we apply a memory state verification on both operation count and operation order. We implemented our analysis in Open64 compiler and the experiment results show that our analysis is effective to detect many potential program deadlock problems.

Our future work will focus on improving our algorithm to deal with more synchronization problems, some of which have already been illustrated in the previous chapters. And we will also try to use our MSSA form to exploit synchronization optimizations which may enhance parallel program performance.

APPENDIX
SOURCE CODE ACQUISITION, AND USAGE

Source Code Acquisition

The version of the Open64 compiler which we used for implementation is 4.2.3, which can be downloaded from http://www.open64.net/download/open64-4x-releases.html

The source codes of our implementation exist on capsl server atlantic, the directory is: xan@atlantic:/fastlane/user/xan/workspace/open64-4.2.3-0

Below gives the list of the new source files created:

osprey/be/be/mssa_main.cxx

osprey/be/be/mssa_main.h

osprey/be/be/mssa_bb.cxx

osprey/be/be/mssa_bb.h

osprey/be/be/mssa_cfg.cxx

osprey/be/be/mssa_cfg.h

osprey/be/be/mssa_dom.cxx

osprey/be/be/omp_lower.cxx

osprey/be/be/Makefile.gsetup

Below gives the list of the modified source files:

osprey/be/be/driver.cxx

osprey/be/region/region_util.cxx

osprey/be/region/region_util.h

osprey/common/com/intrn_entry.def

osprey/common/com/wn_core.h

osprey/common/com/wn_pragmas.cxx

osprey/common/com/wn_pragmas.h

osprey/common/com/config_opt.h

osprey/common/com/config_opt.cxx

osprey/common/com/config.h

osprey/common/util/bitset.c

osprey/common/util/bitset.h

osprey/wgen/ wgen_expr.cxx

libspin/gspin-tree.c

libspin/gspin-tree.h

osprey-gcc-4.2.0/gcc/tree.c

osprey-gcc-4.2.0/gcc/builtins.def

osprey-gcc-4.2.0/gcc/builtin-types.def

MSFA Usage

To invoke the MSFA, just use the following command:

opencc -mp -keep -Wb,-trLOW your_programname.c

The analysis information will be stored in a trace file named your_programname.t, and error information will be printed on the screen if it is detected.

BIBLIOGRAPHY

[1] Cray Inc. Cray XMT. System Overview. 2009.

[2] Cray Inc. Optimizing Loop-Level Parallelism in Cray XMT Applications, 2009.

[3] Cray Inc. Cray XMT Programming Environment User's Guide. March 2009.

[4] Feo, John, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. Proceedings of the 2nd Conference on Computing Frontiers (May 2005): 28C34.

[5] George Chin Jr., Andres Marquez, et al. Implementing and Evaluating Multithreaded Triad Census Algorithms on the Cray XMT. IPDPS '09. 2009.

[6] Jace A. Mogill and David J. Haglin. A comparison of Shared Memory Parallel Programming Models. CUG2010. 2010.

[7] Yuan Zhang and Evelyn Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. PpoPP'07. san Jose, CA. March 2007.

[8] Dirk Grunwald and Harini Srinivasan. Data Flow Equations for Explicitly Parallel Programs. PPoPP'93, May 1993.

[9] Natthew Huntbach. A concurrent Programming Model using Single-assignment, Single-writer, Multiple-reader Variables.

[10] Jens Knoop, Bernhard Steffen and Jurgen Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. ACM TOPLAS, Vol. 18, No.3, May 1996.

[11] Harini Srinivasan. Optimizing explicitly parallel programs. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, July 1994.

[12] Harini Srinivasan, James Hook and Michael Wolfe. Static single assignment for explicitly parallel programs. POPL'93. Jan 1993.

[13] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. LCPC'93. August 1993.

[14] Jeanne Ferrante, Karl J. Ottentein and Joe J. Warren. The program dependence graph and its use in optimization. ACM TOPLAS, 9(3)319-349, July 1987.

[15] http://www.cc.gatech.edu/ bader/code.html

[16] David Ediger, Karl Jiang et al. Massive Social Network Analysis: Mining Twitter for Social Good. ICPP 2010.

[17] David Ediger, Karl Jiang et al. Massive Streaming Data Analytics: A Case Study with Clustering Coefficients. MTAAP 2010.

[18] David A. Bader, Jonathan Berry et al. STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. Georgia Institute of Technology, Tech. Rep., 2009.

[19] David A. Bader, John Feo, et al. HPCS Scalable Synthetic Compact Applications ♯2 Graph Analysis. (SSCA♯2 v2.2 Specification). September 2007.

[20] David A. Bader, Kamesh Madduri et al. Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems. CTWatch Quarterly, 2(4B):41–51, 2006.

[21] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering, 12(1):157-171, 1986.

[22] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001.

[23] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP:Path-sensitive program verification in polynomial time. In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 57-68, 2002.

[24] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In Proceedings of the 10th International Static Analysis Symposium, 2003.

[25] Yue Yang, Anna Gringauze, Dinghao Wu, and Henning Rohd. Microsoft Research TechReport. Detecting Data Race and Atomicity Violation via Typestate-Guided Static Analysis. MSR-TR-2008-108,2008.

[26] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In the ACM Symposium on Principles of Programming Languages, 2002.

[27] Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. ICSE'10, May 2010, Cape Town, South Africa.

[28] Dean M. Tullsen. Jack L. Lo, et al. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. Proceedings of the 5th International Symposium on High Performance Computer Architecture, January 1999.

[29] Diego Novillo, Ronald C. Unrau and Jonathan Schaeffer. Analysis and Optimization of Explicitly Parallel Programs. Technical Report TR 98-11 University of Alberta. August 1998.

[30] Jason Riedy and Rich Vuduc. Microbenchmarking the Tera MTA. Tech-report, Berkeley. May 21, 1999.

[31] Douglas C. Schmidt and Tim Harrison. Double-Checked Locking - An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. 1997.

[32] JeanLouis Colaco, Bruno Pagano and Marc Pouzet. A Conservative Extension of Synchronous Dataflow with State Machines. EMSOFT05, September 19C22, 2005, Jersey City, New Jersey, USA.

[33] Yuan Zhang, Vugranam C. Sreedhar and Weirong Zhu. Optimized Lock Assignment and Allocation: A Method for Exploiting Concurrency among Critical Sections. PPoPP07 March 14C17, 2007, San Jose, California, USA.

[34] David Mizell and Kristyn Maschhoff. Early experiences with large-scale Cray XMT systems. IPDPS '09 Proceedings of the 2009 IEEE International Symposium on ParallelδDistributed Processing. 2009.

[35] Jaeyong Shim, Dongsoo Han, and Hongsoog Kim. Communication Deadlock Detection of Inter-organizational Workflow Definition. S. Bhalla (Ed.): DNIS 2002, LNCS 2544, pp. 43C57, 2002.

[36] Stephen P. Masticola and Barbara G. Ryder. A model of Ada programs for static deadlock detection in polynomial times. PADD '91, Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging. 1991.

[37] Shivali Agarwal, Rajkishore Barik and Dan Bonachea et al. Deadlock-Free Scheduling of X10 Computations with Bounded Resources. SPAA07, June 9C11, 2007, San Diego, California, USA.

[38] Shivali Agarwal, Rajkishore Barik and Vivek Sarkar. May-Happen-in-Parallel Analysis of X10 Programs. PPoPP07, March 14C17, 2007, San Jose, California, USA. 2007.

[39] John Thornley. A parallel Programming Model with Sequential Semantics. PHD Theis, California Institute of Technology. 1996,

[40] Rajiv Gupta. Generalized dominators and post-dominators. POPL'92, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1992.

[41] Ruud van der Pas. An Introduction Into OpenMP. IWOMP 2005.

[42] George Chin, Andres Marquez. Sutanay Choudhury and Kristyn Maschhoff.Implementing and evaluating multithreaded triad census algorithms on the Cray XMT. IPDPS'09, Proceedings of the 2009 IEEE International Symposium on ParallelδDistributed Processing.

[43] Diego Novillo, Ron Unrau and Jonathan Schaeffer. Concurrent SSA Form in the Presence ofMutual Exclusion. 1998 International Conference on Parallel Processing(ICPP'98), Minneapolis,Minnesota, August 1998.

[44] Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng and Uzi Vishkin.Evaluating the XMT Parallel Programming Model. HIPS'01, Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments.2001.

[45] Rob Farber. Experimental comparison of emulated lock-free vs. fine-grain locked data structures on the Cray XMT. Parallel δ Distributed Processing, Workshops and Phd Forum (IPDPSW). 2010.

[46] Jong-Deok Choi, Jong-Deok Choi and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. POPL'91, Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.1991.

[47] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, et al. Efficiently computing static single assignment form and the control dependence graph. Transactions on Programming Languages and Systems (TOPLAS),Oct 1991.

[48] Saman P Amarasinghe. Parallelizing Compiler Techniques Based on Linear Inequalities. Phd thesis, Stanford University. 1997.

[49] http://www.open64.net/

[50] MARTIN ABADI, CORMAC FLANAGAN and STEPHEN N. FREUND. Types for Safe Locking: Static Race Detection for Java. ACM Transactions on Programming Languages and Systems, Vol. 28, No. 2, March 2006, Pages 207C255.

[51] N. Sterling. Warlock - a static data race analysis tool. USENIX Winter Technical Conference, pages 97-106, 1993.

[52] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. SOSP '03, Proceedings of the nineteenth ACM symposium on Operating systems principles.

[53] http://babelfish.arc.nasa.gov/trac/jpf

[54] http://bandera.projects.cis.ksu.edu/

[55] Jaejin Lee, David A. Padua, Samuel P. Midkiff: Basic Compiler Algorithms for Parallel Programs. PPOPP 1999: 1-12

[56] Matthew B. Dwyer.Data Flow Analysis Frameworks for Concurrent Programs. Technical Report. University of Massachusetts. 1995.

[57] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. ISCA '07, Proceedings of the 34th annual international symposium on Computer architecture. 2007.

[58] http://en.wikipedia.org/wiki/Cyclops

[59] Allan Snavely, Larry Carter, Jay Boisseau, et al. Multi-processor Performance on the Tera MTA. Supercomputing'98, Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM). 1998.

[60] Gail Alverson, Preston Briggs, Susan Coatney, et al. Tera Hardware-Software Cooperation. Supercomputing'97, Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM). 1997.

[61] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. ACM Trans. Program. Lang. Syst., 11(4):598C632, 1989.

[62] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In in Proc. of Conf. on 1991 Functional Programming Languages and Computer Architectures, pages 538C568, 1991.