Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2012-08

M.Sc. Thesis

Compile Time Analysis for Hardware Transactional Memory Architectures

Anupam Chahar, B.Tech.

Abstract

Transactional Memory is a parallel programming paradigm in which tasks are executed, in forms of transactions, concurrently by different resources in a system and resolve conflicts between them at run-time. Conflicts, caused by data dependencies, result in aborts and restarts of transactions, thus, degrading the performance of the system. In case these data dependencies are known at compile time, then the transactions can be scheduled in a way that conflicts are avoided, thereby, reducing the number of aborts and improving significantly the system's performance. This thesis presents the Compiler insights to Transactional memory (CiT) tool, an architecture independent static analyzer for parallel programs, which detects all potential data dependencies between parallel sections of a program. It provides feedback about load-store instructions in a transaction, dependencies inside of a loop and branches, and severals warnings related to system calls which can can affect the performance. The efficiency of the tool was tested on an application including different types of induced data dependencies, as well as several applications in the STAMP benchmark suit. In the first experiment, a 20% performance improvement was observed when the two versions of the application were executed on the TMFv2 HTM simulator.

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Compile Time Analysis for Hardware Transactional Memory Architectures

A GCC Plugin Support for Hardware Transactional Memory

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Embedded Systems

by

Anupam Chahar, B.Tech.
born in Agra, India

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Compile Time Analysis for Hardware Transactional Memory Architectures"** by **Anupam Chahar, B.Tech.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: August 2012

Chairman: ____________________
prof.dr.ir. A.J. van der Veen

Advisors: ____________________
dr.ir. T.G.R.M. van Leuken

____
ir. Sumeet S. Kumar

Committee Members: ____________________
dr. K.L.M. Bertels

____

Abstract

Transactional Memory is a parallel programming paradigm in which tasks are executed, in forms of transactions, concurrently by different resources in a system and resolve conflicts between them at run-time. Conflicts, caused by data dependencies, result in aborts and restarts of transactions, thus, degrading the performance of the system. In case these data dependencies are known at compile time, then the transactions can be scheduled in a way that conflicts are avoided, thereby, reducing the number of aborts and improving significantly the system's performance. This thesis presents the Compiler insights to Transactional memory (CiT) tool, an architecture independent static analyzer for parallel programs, which detects all potential data dependencies between parallel sections of a program. It provides feedback about load-store instructions in a transaction, dependencies inside of a loop and branches, and severals warnings related to system calls which can can affect the performance. The efficiency of the tool was tested on an application including different types of induced data dependencies, as well as several applications in the STAMP benchmark suit. In the first experiment, a 20% performance improvement was observed when the two versions of the application were executed on the TMFv2 HTM simulator.

Acknowledgments

First of all, I would like to thank my advisor dr.ir. T.G.R.M. van Leuken for allowing me to do this thesis under his supervision. Thanks for giving me the freedom of choosing the direction in the field which was unknown to me. Your inputs in every meeting was really important and helpful.

I would like to thank my mentor ir. Sumeet S. Kumar for his support from the first day to the last month. Thanks for keeping me motivated all throughout the duration of the thesis. Thanks for answering all my questions so patiently. Thanks for all long hour discussions we had with your great inputs. It would not have been possible without you.

Thanks Tasos for keeping the environment in the workspace really cheerful. Also, helping and guiding me whenever I required.

I would also like to thank to Kuchbhi group. Pavan, Kaushal and Rakshith, you guys rock. Thanks for sharing your experience and, guiding and helping me during my masters. Rakshith J, Nadeem, Nakul, Karthik for being such lovely housemates. I had an awesome time with you guys which I can't forget. Vignesh, thanks for being such a lovely friend and companion since the starting of masters in embedded system. Thanks for nice discussions we had about studies, careers and space and also for guidance, suggestions, help and support in the end.

A special thanks to Chesta, Nadeem, Harshad and Vignesh, for their full support and help in the toughest phase of my life. You all are amazing. Ankita, special thanks for your all support from such a long distance, which always has encouraged and motivated me throughout my masters.

In the last but not the least, a very special thanks to my family which has been supporting and encouraging me since my childhood. You always believed me without any single penny of doubt. With your motivation only I have reached so far. Once again thanks for your infinite love.

Anupam Chahar, B.Tech.
Delft, The Netherlands
August 2012

Contents

List of Figures

List of Tables

Introduction 1

1.1 Motivation

Multiprocessing is becoming prominent in this era due to constraints in uniprocessors. Consequently, parallel programming is being more emphasized in order to exploit multiprocessors. Hence, programmers are required to parallelize sequential software. Over the years, there were tools developed [9] which can help programmers to find the potential parallelism. However, the problem for the programmer is not limited to the parallelization of a sequential software. In the parallelized code, the programmer needs to consider synchronization of shared data between parallel sections of the software. Although, synchronization can be achieved using conventional lock based system, it is expensive in terms of memory usage and can cause problems like deadlock, convoying and priority inversion. Moreover, the programmer needs to be aware of concurrent accesses of the shared data. To relieve the programmer from these constraints, researchers have explored ways to ease the programmability of software. Transactional Memory [16] is one of the solutions which is promising in reducing the developer's effort in parallel programming. In transactional memory, all data dependencies due to shared data between parallel sections are resolved at run-time, thereby, making the programmer relaxed about synchronization of shared data.

TMFab [18] and TMFv2 [23] are hardware transaction memory based multiprocessor system on chip architectures. In these architectures, a hardware scheduler is responsible for scheduling transactions on different processors. After the complete execution, each transaction validates all its speculative writes to other transactions. If there is a conflict due to shared data in between transactions then one transaction has to abort and restart, and other commits to the memory. Due to abort, the transaction has to be executed again, which results in degradation in performance. If conflicts are known at compile time, transactions can be scheduled in order to reduce the number of aborts and restarts by which there can be a significant improvement in performance as mentioned in [23].

In this thesis, we propose the *Compiler insights to Transaction memory* (CiT), a tool for statically analyzing transactions to find the potential data dependencies between them. Feedback about loops, load-store instructions and system calls is given to the programmer about the transaction in order to aid him to write an optimized code. The CiT tool is the plug-in to GCC and gets instantiated during compilation. The CiT tool works on internal representation of the language in the compiler which is independent to any architecture.

1.2 Thesis Goals

The thesis describes the development of a software tool chain in order to provide details of data dependencies between parallel tasks. The main objectives of the thesis are:

- To develop a software tool which can support scheduler of TMFab and TMFv2 architecture in bringing the best performance out of the architecture.
- To help programmer, to write an optimized code for transactions.

1.3 Contribution

In this thesis, we have developed an architecture independent static analyzer for parallel transactions. The main contributions of the thesis are:

- An architecture independent static analyzer for parallel transactions is developed through which, in a given a parallelized program, all static dependencies can be detected and make the programmer aware of possible dynamic dependencies.
- All potential data dependencies are detected in the test application with different scenarios and few STAMP benchmarks.
- Provides feedback about load-store instructions in a transaction, dependencies under loop and warnings related to system calls which can affect the performance.
- Improvement in the simulation time of TMFv2 using the tool.
- Several parallel applications including STAMP benchmarks are successfully ported on these architectures.

1.4 Thesis Organisation

The 2nd chapter provides the overview of the multi-core systems, techniques used to provide synchronization among parallel processes and issues related to them. Transactional Memory is also introduced as a solution to the issues in the multi-core programming. The existing Transactional Memory architectures (TMFab and TMFv2) is discussed. Also, it introduces to the algorithm analysis and methodologies for the same, in the context of data dependencies. Static program analysis and various methodology used inside compiler are discussed. Further, Compilation procedure is mentioned which is followed by precise details of GCC architecture. Moreover, GCC internal representations (GENERIC and GIMPLE) is briefly described. Plugin, new feature of GCC, is explained with its benefits. In the 3rd chapter, a conceptual overview of CiT tool is explained. First, the necessity and features of CiT is mentioned. Also, what is data dependency in transaction and their causes, are illustrated. It is followed by the approach used and conceptual detail of it. Further, each feedback is briefly explained. In the 4th chapter, the implementation of CiT is discussed in detail. First, CiT plugin placement in the GCC architecture is explained. Thereafter, CiT architecture is briefly explained.

Intra and Inter procedural analysis is illustrated in detail. It is followed by explanation of algorithm to detect potential data dependencies. In the end, process of extracting feedback information is discussed. The 5th chapter describes the validation of the CiT tool by testing it on test application and real applications. Moreover, application in TMFv2 with existing scheduler is discussed, which is supported by a experimental result. The 6th chapter summarizes the goal, motivation and the methodology of the CiT tool. Further, intuitive areas related to the CiT which can be explored in the future are described.

Background 2

This chapter provides the overview of the multi-core systems, techniques used to provide synchronization among parallel processes and issues related to them. Transactional Memory is also introduced as a solution to the issues in the multi-core programming. The existing Transactional Memory architectures (TMFab and TMFv2) is discussed. Also, it introduces to the algorithm analysis and methodologies for the same, in the context of data dependencies. Static program analysis and various methodology used inside compiler are discussed. Further, Compilation procedure is mentioned that is followed by precise details of GCC architecture. Moreover, GCC internal representations (GENERIC and GIMPLE) is briefly described. Plug-in, new feature of GCC, is explained with its benefits.

2.1 Multicore System

In this modern technology era, parallelism has critical role for optimized use of resources and saving time. Parallel computing is becoming important because of speed and instruction level parallelism limits in uni-processor [15]. Therefore, multi-core architectures are being explored in modern systems.

A Multi-core system can have number of processors ranging from two to thousands. System can have memory that is distributed, shared and hybrid. Applications on these systems require communication and synchronization between parallel sections. This is achieved through two well-known techniques:

Message passing : In this technique, each process can communicate and send/receive the data through message to/from other processes;

Shared memory : In shared memory systems, the same memory space is shared by all processes. Intercommunication is done through shared variables in this memory.

Message-passing has relatively simple hardware configuration, but it makes programming more complex as the programmer must explicitly write his program as communication process. Also, the programmer has to keep the data packet's size small as the communication latency increases with the size of the packet. On the other hand, shared memory has complex hardware, but it makes the programmer with an illusion of a single memory common to all processes. All parallel tasks can work with a common dataset resident memory. In order to provide a coherent view of memory to all parallel tasks, there are mechanisms, like snoopy protocol over a shared-bus [11] or directory based coherence on interconnects [3], inside hardware. Despite several advantages over the message passing, in shared memory, programmers are required to have synchronization in the shared data. Thus, hardware rules like memory consistency models [4] have been devised and software routines have been developed around these rules to provide necessary synchronization.

Locking is one of the techniques to provide the synchronization. In conventional lock-based systems, any task before accessing the shared data needs to get ownership of accessing rights through lock. Subsequently, the task has to release the lock after accessing the shared data that allows other task to access that. This process of acquiring and releasing a lock makes programming complex as the programmer needs to know the exact data which is being shared. Further, locks are expensive as they need extra memory space and time to initialize and destroy. Moreover, lock based programming model is always prone to various problems:

1. *Priority Inversion*: occurs when lower priority process is holding a lock needed by higher priority process.
2. *Convoying*: occurs when a process holding a lock no longer exists or descheduled, other processes which need it won't be able to proceed.
3. *Deadlock*: can occur if processes attempt to lock same set of objects in different order.

To resolve these problems with lock-based system methodology, concept of transactional memory is introduced [16].

2.2 Transactional Memory

It is an architecture intended to make lock-free synchronization as efficient as conventional techniques based on mutual exclusion. In this model, the critical section is marked as a transaction. A *transaction* is finite sequence of machine instructions, executed by a single processor, satisfying following properties:

- *Serializability*: Transactions appear to execute serially, meaning that the steps of one transaction never to be interleaved with steps of another. Committed transactions are never observed by different processors to execute in different orders.
- *Atomicity*: Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making its changes visible to other transactions (effectively) instantaneously, or it aborts, causing its changes to be discarded.

Concept of transaction can be implemented in the software and hardware.

Software Transactional Memory In software transactional memory (STM)[28][13] [8], the architecture is build either as a library or at compiler level [17]. For a transaction, STM allocates two data structure at run-time: Read set: every memory read is stored in it; Write set: every memory written is stored in it. Once the transaction is complete, it try to commit to the memory by verifying read and then performing write. If the read set is mismatched then the transaction aborts and restarts. As the above mentioned process is done through the software, it is slower than hardware transactional memory. However, it does not required any specific hardware.

Hardware transactional Memory In hardware transaction memory architectures [16] [12] [10], the programmer just specify the critical section as a transaction and hardware resolves all conflicts at run-time. A transaction after complete execution need to *validate* all its speculative writes with others transactions on several processors to find conflicts. Once validated, it is allowed to commit writes to the shared memory. There are existing hardware implementations of the transaction memory like TCC [12], TMFab [18] and TMFv2 [23].

2.2.1 TMFab and TMFv2

TMFab [18]is a stacked-die transactional many-core processor architecture with goals of having scalability, processor independent architecture and the ease of multi-programming. It is composed of three main components: Scheduler, Processing Element (PE) and L2 Data Cache. Each processing element has its own L1 instruction cache and transactional data cache. A Speculative Write Buffer (SWB) resides in L1 data cache to hold all speculative writes. L2 data cache is shared among all the processing element. A hardware scheduler is responsible for scheduling transactions to various processing elements. For communication between all tiles, network-on-chip based interconnect architecture is used in order to provide scalability.

In the TMFab, all transactions are scheduled on different processors by a hardware scheduler. A transaction starts execution immediately once it is scheduled. TMFab uses lazy version management, which means that the transaction's speculative writes are held in a local-write buffer, i.e. Speculative Write Buffer(SWB), instead of updating to the cache. Once the transaction is completely executed, all addresses in SWB has to be validated with other transactions being executed. Conflict detection scheme checks whether there are conflicting addresses between any transactions or not. If there is a conflict, then contention management scheme handles the conflict by aborting the lower priority transaction. When a transaction aborts, data cache and speculative write buffer get refreshed and the transaction restarts.

In terms of performance, TMFab has best case speed of 3.44x over single core execution while maintaining the ease of programmability. However, the performance of TMFab is restricted by its validation scheme as it is not scalable. According to the scheme, once a transaction is complete it sends broadcast message on the network with addresses it needs to validate. The increase in number of processing elements will induce more congestion in the network and overhead of validation process at every processing elements. Further, in TMFab only one transaction is allowed to validate at a time. Consequently, all other transactions has to wait to get network cleared of any other validation process.

TMFv2 [23] is based on the TMFab architecture. The main goal of this architecture is to reduce the overhead of validation and traffic congestion in the network. It is architecturally different from TMFab in terms of its validation scheme and banked memory.

In TMFv2, a transaction after completion validate all addresses it has written over at the shared memory, instead of validating them with other transactions. Thus, it unicast on the shared memory instead of broadcast to all processing elements. This avoids significant amount of traffic in the interconnect. Also, other processing elements can continue their execution without any interruption unlike in TMFab. Further, TMFv2 allows more than one transaction to validate in parallel.

The speedup of 2.5x was achieved in the validation process and 2.7x reduction in the memory latency. Although it reduces the overhead of validation but it introduces the concept of hazards. In the architecture, a write to the memory is at the granularity of a byte but the conflict detection scheme checks at the level of word. A *hazard* is introduced when the conflict detection scheme detects the conflict when bytes are written at two different location on the same word by two different transactions respectively. A hazard detection scheme is introduced to resolve this issue. Every validated cache line goes to the hazard detection module which compares each byte of old and new cache line. If there is no conflict then the transaction is allowed to commit. Due to hazard, every cache line has to be re-validated which cause an overhead.

As it is mentioned in [23], that a software toolchain is needed, to track the static and dynamic dependencies at compile-time. Using this information, a new scheduling scheme could be devised, which orders transactions in a way that aborts are avoided, or stalls transactions before speculatively accessing conflicting addresses in the shared L2D.

2.3 Related Work

There are techniques and tools available for the static and dynamic analysis to detect data dependencies. Some of them discover data dependencies within the loop, thereby exposing opportunities for data level parallelism. For example, many static analyzers are using techniques mentioned in [27] for finding the data dependency within loop. On the other hand, profiling tool Embla [9] allows user to discover data dependency in a sequential program, thereby, exposing opportunities for task level parallelism. Similarly, tools like QUAD [2], a dynamic analyzer, focus on discovering data dependencies between functions. Embla and QUAD use Valgrind framework[25] and Pin [19] runtime binary instrumentation system respectively. At present, QUAD does not explicitly show data dependencies between functions when they are executing in parallel. However, there is a paper [20] which detects data dependencies between transactions, but it is meant for software transaction memory. Since, there is no relevant information about the implementation part, it is not possible to transform the paper for hardware transactional memory architectures. We believe, this thesis is the first compile time methodology for the data dependency analysis for parallel transactions programs with a feedback about the transaction for hardware transaction memory architectures.

In order to track static and dynamic dependencies at compile time, it is required to analyze the algorithm. Therefore, in the next section brief introduction about algorithm analysis is mentioned.

2.4 Algorithm Analysis

Many times it is necessary to know how a newly developed algorithm by a programmer will behave on machine. To know the behavior of the algorithm at run-time, an algorithm is analyzed. There are two methods through which an algorithm is analyzed: static analysis and dynamic analysis. The goal of both methods is common i.e. to extract relevant information from algorithm which affects the behavior of the program but in different ways.

Dynamic analysis is done at run-time. It records the behavior of algorithm during execution. This is achieved by inserting instrumented code inside the algorithm. Through this an algorithm is monitored step by step. Dynamic analysis is also called Profiling. Static analysis, on the other hand, is done at compile time. Thus, algorithm need not to be run on the machine or completely compiled. These analysis techniques can be used to detect data dependencies in parallel programs.

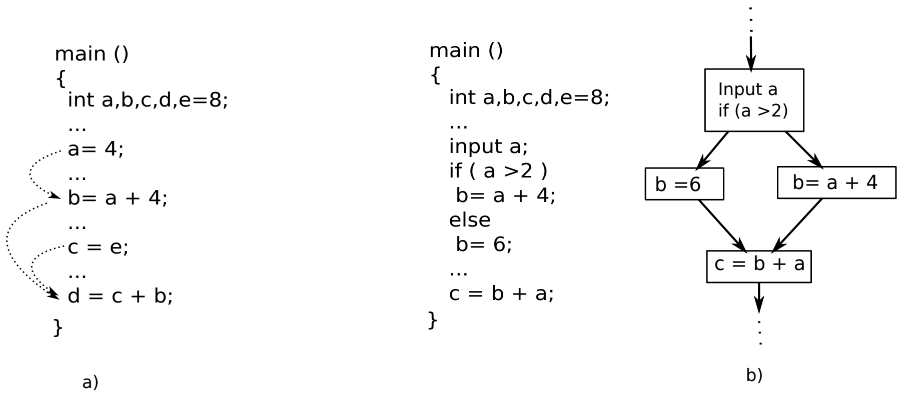
2.5 Data dependency in parallel programming

Parallel programming is becoming more promising in terms of performance. But the complexity of the writing parallel sections is also increasing due to consistency issue in the shared memory. It is difficult to figure out relations between parallel sections in terms of the shared data. One of the relations between parallel sections is called *data dependency*. Two parallel code sections are said to be data dependent when they try to access the data at the same address in the shared memory. There can be two types of data dependencies- static and dynamic. A *static data dependency* is a dependency which is explicit to the programmer, for instance, accessing a global variable. On the other hand, *dynamic data dependency* occurs when an address is accessed implicitly like in the case of pointer.

In order to help programmer in the case of a data dependency, both type of algorithm analysis can be used. In dynamic analysis, the exact address can be detected which is being accessed by the algorithm when it is executing especially in the case of dynamic data dependency. But in static analysis, it can only be sure of static data dependencies. Consider, for example, an array being accessed in a loop with undetermined number of iterations. Dynamic analyzer can detect each and every element accessed for all the iterations executed at run-time. In static analyzer, since it is not possible to determine number of iterations, it can only inform about the array but not the exact element which is being accessed. In this case, dynamic analyzer outperforms static analyzer. However, static analyzer is really helpful for developers concerned of a worst case analysis. Static analyzer apart from detecting static data dependencies, can also estimate dynamic dependencies. Now, for a developer it is really important to take worst case in consideration while writing the code. In dynamic analyzer, to detect each and every dependencies, developer has to execute the algorithm for every possible case which is tedious to do. Instead, to save time and effort, programmer can take worst possible case model from static analyzer. Therefore, in this thesis it is believed that static analyzer is better than dynamic.

2.6 Static program analysis

To analyze a program statically for data dependencies, it is necessary to check the behavior of the program in terms of data. A load and store of data in the memory, i.e. data definition and use of that definition is the main concern. A data definition can be of interest if it is being used by some other definition later in the program. Analyzing this "def-use" or "generation and consumption" relationships is analyzed through data flow analysis [6] [5].



```
main ()
{
 int a,b,c,d,e=8;
 ...
 a= 4;
 ...
 b= a + 4;
 ...
 c = e;
 ...
 d = c + b;
}
```

```
main ()
{
  int a,b,c,d,e=8;
  ...
  input a;
  if ( a >2 )
   b= a + 4;
  else
   b= 6;
  ...
  c = b + a;
}
```

Figure 2.1: a) Data flow b) Control flow

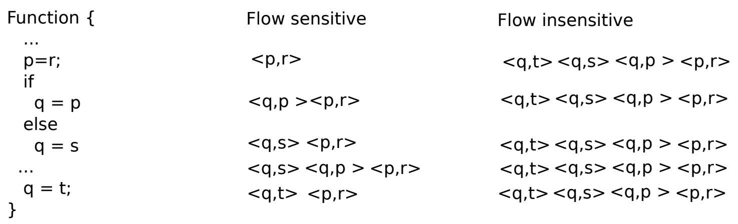
The Figure 2.1 a) is depicting one such example of *data flow*. *Control flow* of a program represents sequential paths of execution of code sections in the program as shown in the Figure 2.1 b). The data flow is not limited to a function, but also between functions. Similarly, control flow can be of inside a function and between functions. Generally, the control flow is called as *call graph* when it is between functions, as it basically represents calls from inside a function to other functions. The data flow and control analysis inside a function is called *Intra-procedural analysis*, and between functions is called *Inter-procedural analysis.* The intra and inter procedural analysis can further be categorized into flow in/sensitive analysis and context in/sensitive analysis respectively.

2.6.1 Flow in/sensitive analysis

In flow sensitive [7], intra procedural control flow is considered when data flow analysis is being done. In flow insensitive, the data flow inside the function is developed without considering the control flow.. For example, in the Figure 2.2 data definitions at particular statement is according to the control flow, but in flow insensitive all data definitions is considered.

2.6.2 Context in/sensitive analysis

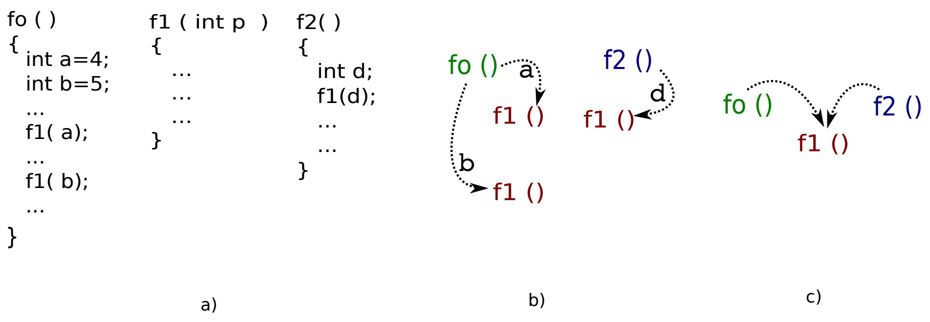
In context sensitive, when a function invoke another function context is considered, i.e. data being passed through arguments. For example, consider the example code given in the Figure 2.3 a). In context sensitive, f0 calls f1 twice with context *a* and



Function {	Flow sensitive	Flow insensitive
...		
p=r;	<p,r>	<q,t> <q,s> <q,p > <p,r>
if		
q = p	<q,p ><p,r>	<q,t> <q,s> <q,p > <p,r>
else		
q = s	<q,s> <p,r>	<q,t> <q,s> <q,p > <p,r>
...	<q,s> <q,p > <p,r>	<q,t> <q,s> <q,p > <p,r>
q = t;	<q,t> <p,r>	<q,t> <q,s> <q,p > <p,r>
}		

Figure 2.2: Difference between flow sensitive and insensitive

b respectively is considered (Figure 2.3 b)). On the other hand, context insensitive just consider calls made by a function without any context (Figure 2.3 c)). To make data dependency check more accurate, it is important to have context sensitive as it is necessary to know which data is being passed to other functions.



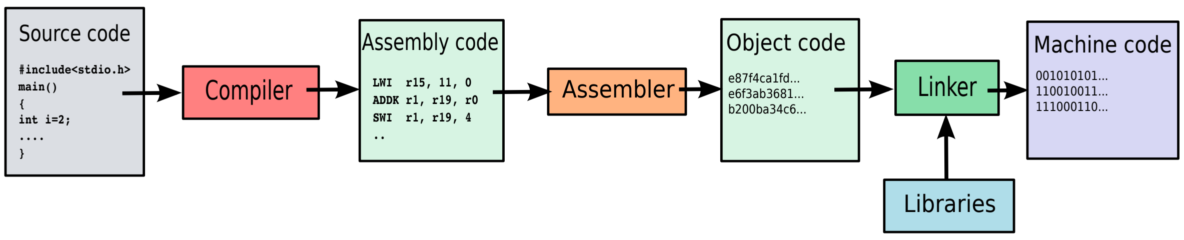
```
fo ( )            f1 ( int p  )     f2( )
{                 {                 {
  int a=4;          ...               int d;
  int b=5;          ...               f1(d);
  ...               ...               ...
  f1( a);         }                   ...
  ...                               }
  f1( b);
  ...
}
```

Figure 2.3: Example code for context in/sensitive analysis

This work is focused on the static analysis during compilation. Therefore, it is required to know fundamentals of compilation process. In the next section, compilation process is briefly explained which is followed by our target compiler (GNU Compiler Collection (GCC)) architecture description and succinctly explained its internals and later about the plug-in feature which is used in this thesis.

2.7 Compilation

Compilation process is composed of four main stages which eventually convert the source code of a computer programming language into the machine language. The process is being shown in Figure 2.4.



```
#include<stdio.h>
main()
{
int i=2;
....
}
```

```
LWI  r15, 11, 0
ADDK r1, r19, r0
SWI  r1, r19, 4
..
```

Figure 2.4: compiler

Compiler take the source code as input and transform into assembly language according to the instruction set of targeted architecture.

Assembler converts the assembly code into the objective code which contains different code segments (text, data, bss etc.) and the symbol table. A symbol table is required by the linker to link other libraries.

Linker integrates existing libraries which is required by the source code. It uses linker script, which describes the memory of the target architecture, to mark data and instruction in their respective address space.

2.8 GNU Compiler Collection

GNU Compiler Collection (GCC) is one of the oldest and widely used compiler. In GCC, a language source code goes through three main stages- front-end, middle-end and back-end; before it gets converted into the machine language, as Figure 2.5 depicts the architecture of GCC.

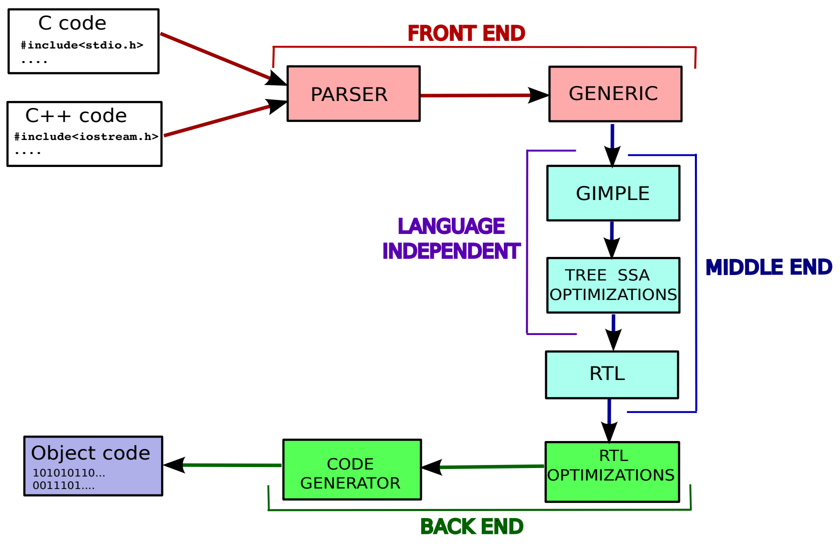


Figure 2.5: GCC Architecture

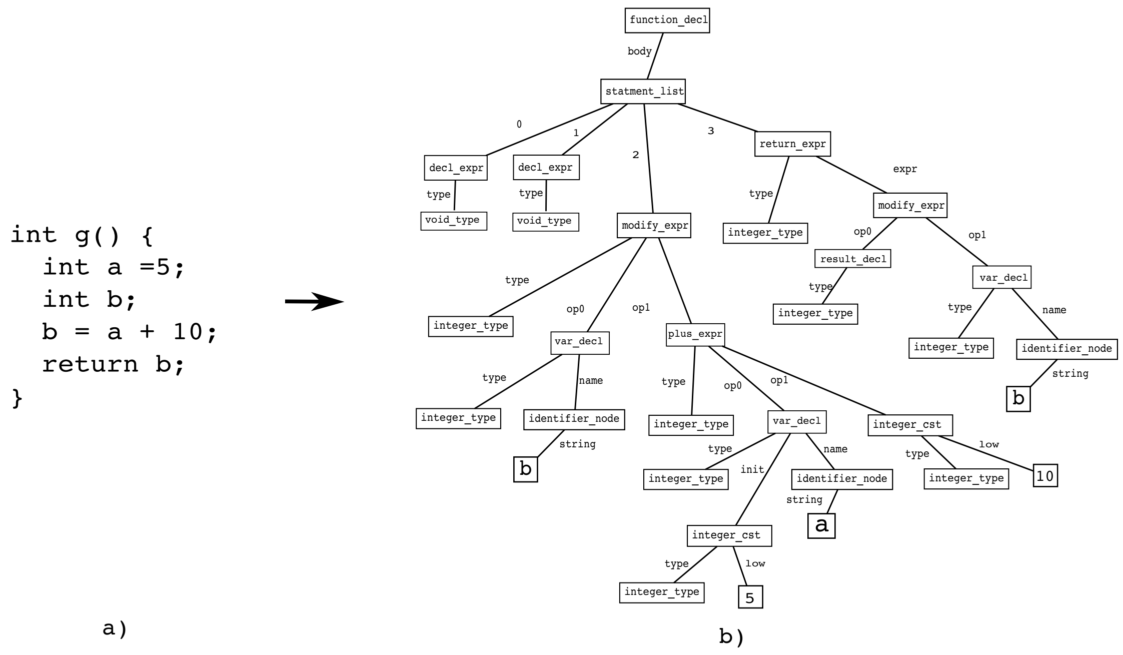
Front End : The front-end of GCC is responsible for parsing the source code and convert it into trees which is used for further compilation and analysis in the later stages of the compiler. Inside the front-end, source code goes through lexical and syntax analysis. It is then converted into intermediate language representation called GENERIC [22] [29]. The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees (explained in the section 2.8.1).

Middle End : The output from front-end, i.e. GENERIC trees, is converted into a simplified form called GIMPLE [22][30] (mentioned in the section 2.8.2). This process is known as gimplification. Inter-procedeural optimizations and all optimizations inside the function, i.e. intra-procedural, is done at GIMPLE. This follows the development of Register Transfer Language (RTL) from intermediate representation(IR). RTL is basically definition of statements in the form of registers.

Back End : The back-end part of the compiler converts the intermediate representation into machine dependent code. Therefore, for each target the back-end is separate. It takes the RTL developed in the middle-end as input and does optimizations according to target architecture. Further, it gets converted into assembly and assembler takes control from there which eventually end up on generating objective code.

2.8.1 GENERIC Trees

GENERIC is a simple way of representing a function in the form of trees. It reorganizes the abstract syntax trees from the parser, so that every function has a separate tree. Further, each statement is a subtree of the function tree. An operand points to other nodes which are representing characteristics of it. For example, in the Figure 2.6-a the C code function `int g()` is conceived in the GENERIC form of tree as Figure 2.6-b depicts. GCC internals provides separate macros in order to access nodes.



```
int g() {
  int a =5;
  int b;
  b = a + 10;
  return b;
}
```

Figure 2.6: A GENERIC tree representation of a function

2.8.2 GIMPLE

GIMPLE is a part of middle-end of GCC. It is a language independent and simplified internal representation of the source code. It mainly uses trees developed at the front-end as GENERIC is already providing a language independent representation of a function. However, GIMPLE breaks down complex statements or expressions into three addresses form. This simplified version of intricate statements is achieved through introduction of temporary variables. For example, the following left expression is converted into simple and three operands form statements.

```
                          T1 = c * d;
a = b++ / (c * d) ;  -->  a = b / T1;
                          T2= b;
                          b = T2 + 1;
```

The gimplified version of the source code can be dumped by using the flag -fdump-tree-gimple. The Figure 2.7-a shows a original C code and Figure 2.7-b is a GIMPLE representation of the function `void f()` of the same C code.

```
struct A {
int a;
int b;
};

int i;
int g();
void f()
{
  int x;
   struct A object;

  object.a= object.b + 2;
   int j = (--i, i ? 0 : 1);

   for (x = 42; x > 0; --x)
   {
      i += g()*4 + 32;
   }
}
```

a)

```
f ()
{
  ... // declarations of temp and local
  D.2560 = object.b;
  D.2561 = D.2560 + 2;
  object.a = D.2561;
  i.0 = i;
  i.1 = i.0 - 1;
  i = i.1;
  i.2 = i;
  j = i.2 == 0;
  x = 42;
  goto <D.1692>;
  <D.1691>:
  D.2565 = g ();
  D.2566 = D.2565 + 8;
  D.2567 = D.2566 * 4;
  i.3 = i;
  i.4 = D.2567 + i.3;
  i = i.4;
  x = x - 1;
  <D.1692>:
  if (x > 0) goto <D.1691>; else goto <D.1693>;
  <D.1693>:
}
```

b)

Figure 2.7: Gimple transformation

A statement in GIMPLE are present in the form of tuple. Each tuple has several components reflecting the information about the statement like code, sub-code, operands etc. A code is an identifier for a GIMPLE instruction. Further, to identify the type of instruction, GIMPLE has separate instruction set. For example, an assign or modify operation is named as GIMPLE_ASSIGN ; an condition is named as GIMPLE_COND; a call for a function is named as GIMPLE_CALL and etc. To know the whole instruction set one can refer [30]. To distinguish different variants of the same basic instruction, a sub-code is used to give more detail about the instruction. In assignments, sub-code has most prominent use, to indicate the operation done on right hand side of the assignment. For example, the statement a = b + c is encoded as GIMPLE_ASSIGN<PLUS_EXPR, a, b, c>. In this example, code (or better say instruction) is GIMPLE_ASSIGN through which a value is assigned to operand "a" (left hand side) by an operation PLUS_EXPR on the operands "b" , "c". Similarly, every statement of the function `void f()` of the C code used in Figure 2.7-a), in GIMPLE is converted into a tuple as Figure 2.8 b) is showing.

```
f ()
{
  ... // declarations of temp and local
  D.2560 = object.b;
  D.2561 = D.2560 + 2;
  object.a = D.2561;
  i.0 = i;
  i.1 = i.0 - 1;
  i = i.1;
  i.2 = i;
  j = i.2 == 0;
  x = 42;
  goto <D.1692>;
  <D.1691>:
  D.2565 = g ();
  D.2566 = D.2565 + 8;
  D.2567 = D.2566 * 4;
  i.3 = i;
  i.4 = D.2567 + i.3;
  i = i.4;
  x = x - 1;
  <D.1692>:
  if (x > 0) goto <D.1691>; else goto <D.1693>;
  <D.1693>:
}
```

a)

```
f ()<
  ...// declarations of temp and local
  gimple_assign <component_ref, D.2562, object.b, NULL>
  gimple_assign <plus_expr, D.2563, D.2562, 2>
  gimple_assign <var_decl, object.a, D.2563, NULL>
  gimple_assign <var_decl, i.0, i, NULL>
  gimple_assign <minus_expr, i.1, i.0, 1>
  gimple_assign <var_decl, i, i.1, NULL>
  gimple_assign <var_decl, i.2, i, NULL>
  gimple_assign <eq_expr, j, i.2, 0>
  gimple_assign <integer_cst, x, 42, NULL>
  gimple_goto <<D.1692>>
  gimple_label <<D.1691>>
  gimple_call <g, D.2567>
  gimple_assign <plus_expr, D.2568, D.2567, 8>
  gimple_assign <mult_expr, D.2569, D.2568, 4>
  gimple_assign <var_decl, i.3, i, NULL>
  gimple_assign <plus_expr, i.4, D.2569, i.3>
  gimple_assign <var_decl, i, i.4, NULL>
  gimple_assign <minus_expr, x, x, 1>
  gimple_label <<D.1692>>
  gimple_cond <gt_expr, x, 0, <D.1691>, <D.1693>>
  gimple_label <<D.1693>>
>
```

b)

Figure 2.8: Gimple tuple

2.8.3 Plugin

As GCC is a open source compiler, it is open to any customization. GCC is a huge compiler and not easy to understand as it has millions of lines of code. In GCC, a source code goes through several passes. In order to customize GCC, a developer can build his own pass. To include the new pass in the actual stream of passes, the developer has to modify pass manager and manually insert the pass at appropriate location. To compile the new pass, full GCC has to be compiled which takes a lot of time. Therefore, customization is a cumbersome job.

To solve all aforementioned problems in customizing the compiler, GCC developers community has introduced the option of *Plugins*. It allows developer to build the new pass separately as it can be compiled using GCC as a normal C code. However, it needs special make file for that which is mentioned at [1]. To use GCC internals, developer can include GCC header files directly into the code. All data structures in GCC can directly be used and there is no need to declare them. To use the plugin, the programmer need to use the option `-fplugin` (arguments can also be given) while compiling a source code with GCC. Since, the plugin is compiled as a share object file, the path of folder where plugin is residing can be given to GCC through option. Further, developer doesn't need to change the pass manager, instead, he only need to mention the pass characteristics like location, type and name of the pass in the code.

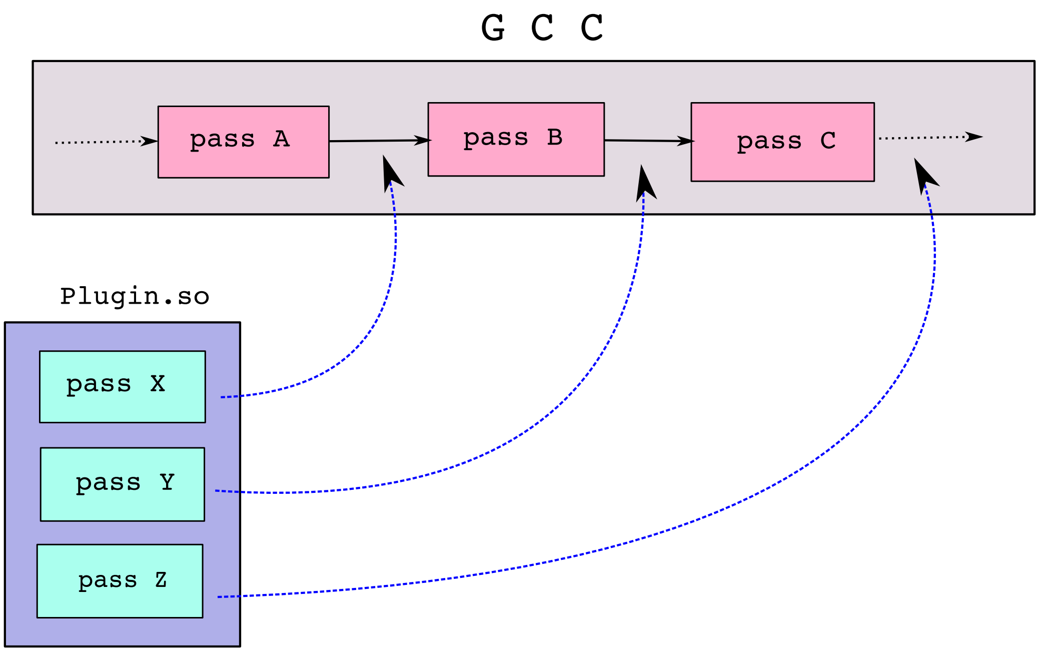


Figure 2.9: Plugin pass insertion in GCC

A plugin can have more than one pass. It also can replace existing pass in the GCC. When a source code is compiled using plugin, the pass manager initiate the plugin and insert addresses of all passes at appropriate locations. After insertion, GCC goes through passes which are newly included at their locations. As Figure 2.9 is depicting. after pass A , pass X is executed instead of pass B which eventually be executed once pass X is finished. Similarly, pass Y, C and Z are executed. With all advantages mentioned above, we decide that it is suitable to make tool as a plugin and use the information which GCC already have in the front end and middle end.

2.9 Summary

Multicore systems are becoming prominent due to limits in speed and instruction level parallelism. Therefore, parallel programming is emphasized more to exploit multicore systems. For synchronization in parallel programs conventional lock based system is used which requires extra memory and can cause problems like deadlock, convoying and priority inversion. Transactional memory can resolve these issues and make programmer's life easier, by resolving all conflicts related to the shared data at run-time. Software implementation of transactional memory is slower and expensive in terms of memory usage. Hardware transactional memory architectures, TMFab and TMFv2, provides a good speedup for independent transactions. However, they lose performance if a conflict occurs. They can have improvement in the performance if conflicts details are known at compile time by analyzing algorithm. Algorithm can be analyzed statically or dynamically. As to detect data dependencies, dynamic analysis requires multiple execution with different data set to know all dependencies. Therefore, we believe static analysis is better as it detects all potential data dependencies at one time.

To analyze algorithm statically, data and control analysis inside and between functions are required. A plugin feature can be useful in developing a static analyzer, as it can be used together with GCC and can access internal language representation information from GCC.

The CiT Overview 3

In this chapter, a conceptual overview of the CiT tool is explained. First, the necessity and features of the CiT tool is mentioned. Also, causes for data dependency between transactions are illustrated. It is followed by the approach used and conceptual details of it. Further, feedback provided by the CiT is briefly explained.

3.1 Overview

As parallel programming is becoming more complex, there are architectures which make it simpler for the programmer. TMFab and TMFv2 are examples of these kind of architectures which ease the programmability. In these architectures, hardware is responsible to find data dependencies and resolve them at run-time. Although, during this process these architectures lose performance. For instance :

- In TMFab, the overhead of validation increases with increase in the number of processors as well as data dependencies.
- In TMFv2 and TMFab, an abort can cause significant amount of wastage of time when transactions are huge in size.
- In TMFv2, hazard detection scheme can cause an overhead.

As the programmer doesn't need to be awared of concurent accesses of the shared data while writing a transaction, he may end-up with an unoptimized code for a transaction which can lower the performance.

The CiT[1] is a tool which analyzes parallel transactions statically. By using inter and intra-procedural analysis, it predicts potential data dependencies between transactions. Moreover, the CiT develops data flow and control flow graphs of the transaction which can aid the programmer to know how a transaction can behave at run-time. It provides feedback about load-store instructions in a transaction, dependencies inside loops and warnings related to system calls which can affect performance. The aforementioned collection of information allows programmer to write a transaction in an optimized way to improve the performance. We believe that CiT is the first tool which can detect, statically, potential data dependencies among transactions for hardware transactional memory.

At present, hardware schedulers of TMFab and TMFv2 can't execute according to the information provided by the CiT tool. The dependence aware hardware scheduler is next step to the CiT. Therefore, the CiT tool is in the mid-way of the road-map presented in Figure 3.1.

[1]CiT is derived from Sanskrit word "Chit"which means consciousness

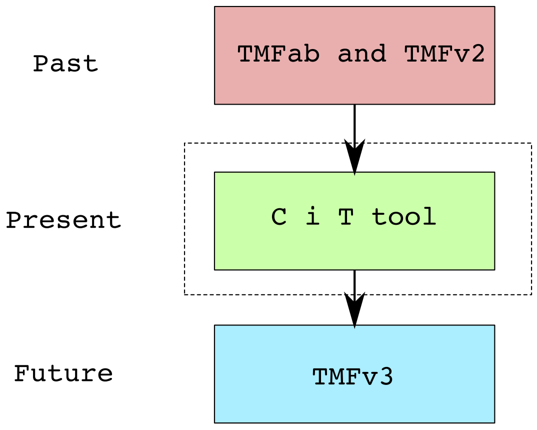


Figure 3.1: Roadmap

If data dependencies details from the CiT are passed on to the hardware scheduler, the performance of the TMFab and TMFv2 can be improved in the following way:

- In TMFab, if the dependencies are known then a transaction can validate with only those transactions to whom it is dependent. Therefore, a transaction doesn't need to validate with all other transactions which are not dependent, resulting in improvement of the execution time of the transaction.
- In TMFv2 and TMFab, if there is a data dependency between transactions then scheduling can be done according to the producer-consumer relationship found in the flow graph. Therefore, it can avoid validate, abort, restart and hazard detection overhead.
- In TMFv2 and TMFab, if there is a branch or loop, in which data dependency lies, is not taken then there is no need to validate. Scheduler can allow the transaction to commit to the memory.
- A transaction should be descheduled or restart if an abnormal termination occurs. This will unnecessary validation and abort overhead.

In subsequent sections, causes of a potential data dependency between transactions are discussed. Later, the CiT approach to detect them and feedback and its meaning in terms of performance is explained.

3.2 Data Dependency in Transactions

Basically, each transaction is a task which has to be executed in parallel on a separate processor. Before analyzing for data dependencies, it is required to investigate how a data dependency can occur between transactions. In other words, it is to be explored how can one transaction accesses addresses which are not local to it and possibly be a data dependency. There can be three ways through which transaction can access an address outside of its stack.

Global variables: As global variables are visible throughout the program, any function can access it directly from the data section of the memory. There is no restriction on writing or reading on a global variable. Therefore, information about all global variables should be collected before analysis.

To detect whether a transaction is accessing a global variable, this information can be used. During analysis, each variable in the transaction can be checked if it is a global variable. All global variables being used in the transaction can be pushed to a bin or collecter which collects all potential data dependency addresses for a particular transaction.

Stack section : Local variables in a function are stored on the stack which are local to that function only. However, these local variables can be accessed from another function as well. This situation occurs when a caller function passes the local variable by reference as an argument to the callee function. As a result of this, callee function can write or read the data at the address passed by the caller. If the callee function is a transaction then the address passed by the caller can potentially be a data dependency for other transactions. For instance, in the Figure 3.2 the variable `j`, which is on the stack of `main()` function, is being passed by reference to two forked functions `function1()` and `function2()` on two separate transactions TXN1 and TXN2. The variable `j`, is being modified in TXN1 and used in TXN2 which leads to a conflict. Therefore, a data dependency occurs due to the stack variable of a caller function outside transactions.

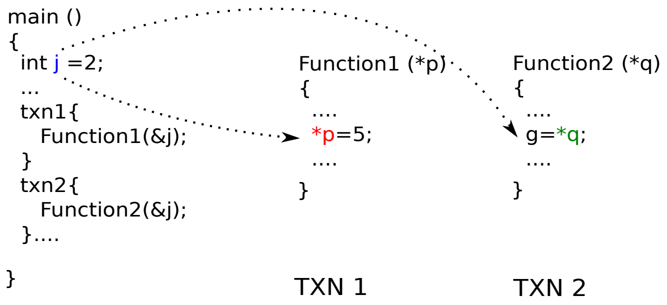


Figure 3.2: Dependency through stack section of caller function

To detect this type of dependency, call graph of the transaction with context is required. All indirect references should be monitored to check the address which is being accessed whether it is coming from outside the transaction through parameters or is it a local address.

Heap section: Dynamic allocation inside a transaction can also cause a potential data dependency [21]. When a memory management function like malloc() is invoked to allocate some amount of data, it checks the heap section for the availability of blocks of data which is equal or more than the amount required. Once blocks are allocated, they are removed from the list of available blocks. The list of available blocks can be accessed at the same time if malloc() is invoked in two transactions which are concurrent. Therefore, it may allocate same blocks of data to both transactions which can lead to the occurrence of a data dependency.

It is not possible to detect this type of dependency statically. Therefore, a warning can be issued to the programmer about usage of a memory management system call.

Combination of global and heap section: Another situation can also cause a potential data dependency, i.e., combination of global and heap section. An address generated through memory management function can also be accessed by any transaction through global variable. Linked list is the best example of this case, where any node can be generated, added or deleted anywhere in the program to/from the list by accessing the Head and Tail node, which are global. In this case, it would be difficult to detect the potential data dependency statically. However, a warning can be issued to the user that transaction is using global pointer and can access same addresses which may conflict with other transactions.

Apart from aforementioned causes for data dependencies, in TMFab and TMFv2 there are other conflicts between transactions. It is found that stack space of transactions can cause conflicts. Since GCC is a sequential compiler, it doesn't consider transaction as parallel task which leads every transaction to use the same stack. These kind of conflicts are unnecessary as they are not supposed to occur. The possible solution is to have separate stack for every transaction. It is explained in detail in appendix A.

3.3 Approach

In order to detect aforementioned data dependencies, it is necessary to track all addresses which are being passed into the transaction through arguments of the function and global variables. Therefore, it is required to have data flow between functions. To know whether data is being read or written on outside addresses which are being passed into the function, data flow inside the function is a requisite. Thus, the following information is required:

- Intra-procedural data flow, to keep track of data definitions (assignments and reading of variables) inside the function.
- Intra-procedural control flow, to keep track of control paths which decide the data flow inside the function.
- Inter-procedural data flow, to keep track of data flow between functions, i.e. through arguments.
- Inter-procedural control flow, to keep track of call graph inside a transaction.

In the compiler, the information about Intra and Inter- procedural data flow analysis is done in order to have optimizations like constant propagation, copy propagation, dead variable etc. In this thesis, we are analyzing parallel sections but the optimization in compilers is meant for sequential code. Thus, existing analysis in the compiler cannot be used and it is required to build new data flow analysis. Similarly, new call graph has to be build for each transaction. However, existing control flow information of a function ,i.e. intra-procedure, can be used since it is separate for every function. Therefore, data flow (both intra and inter procedural) and inter procedural control flow is developed and information regarding intra-procedural control flow from compiler is used for the analysis.

To understand the flow of the approach, Figure 3.3 can be referred. As it is depicting, the information from the intra-procedural analysis is used by inter-procedural analysis, i.e., the information about data flow inside the function, which is developed with the help of control flow of the function, is required to analyze the flow of data between functions. Call graph, from the control flow of the transaction, is required to know which functions are being invoked in the transaction.

Moreover, during the intra-procedural analysis feedback for the code of transaction written is prepared, as most of the information regarding loops, branches, load-store instructions, and system calls can only be analyzed within a function. Warnings related to recursion are analyzed during control flow of transactions.

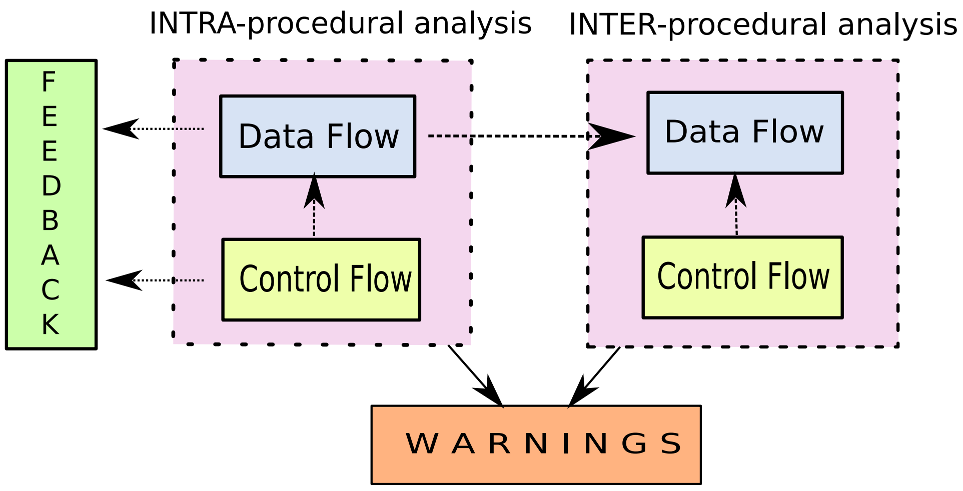


Figure 3.3: Approach

3.4 Intra-procedural analysis

Data flow analysis is a technique to observe the generation and consumption of a particular datum through variables. Generally, generation and consumption of the data refers to definition-use relationship. At every statement, the set of data definitions that:

- Reach the beginning of the statement is called En[ ], which can also be referred as *can be consumed*.
- Generated in the statement called gen[ ], which always have one element in the set in the case assigning and null in other type of statement.
- Used in the statement called use[ ].
- Reach the end of the statement called Ex[ ].

All four above mentioned conditions can be represented in the data flow equation 3.1. This equation is used to build up the database for variables.

$$Ex[S] \quad = \quad En[S] \quad \cup \quad gen[S] \cup \quad use[S] \tag{3.1}$$

For instance, consider the statement s in the Figure 3.4 in which x is being assigned

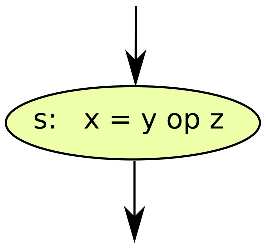


Figure 3.4: Example statement of a code

using y and z. Using equation 3.1,:

$$\begin{gathered} En[s] = \{y, z\} \\ gen[s] = \{x\} \\ use[s] = y, z \\ \therefore Ex[s] = \{x, y, z\} \end{gathered}$$

At the beginning of the statement s, y and z exist in the database. But these definitions are being used to generate x. Therefore, x is added to the database while y and z already exist. In the case when $En[s] = \emptyset$ and, y and z are being used to assign a value to x then both y and z should have been added to the database. There can be a case when x is again defined with older definition already exist. The older definition is termed as *killed definition* and discussed in the next section.

3.4.1 Killed definition

It is possible that $gen[]$ and $En[]$ have common elements, i.e. same variable has two definitions. In that case, each definition is considered to be a data definition of a new variable. Since instructions are executed sequentially inside a processor, previous definition is *kill*ed and no longer used. Therefore, making a new data definition for that variable is necessary.

For example, in the Figure 3.5, at statement s1:

$$Ex[s1] = y, x, a, b \equiv En[s2]$$

. Generating data variables at s2 is $gen[s2] = x$, since the element x belongs to $En[s2]$, it is considered that data definition of x is killed after that statement for whole function. Therefore, data flow equation is :

$$\begin{gathered} Ex[s2] = En[s2] \cup gen[s2] \cup con[s2] \\ Ex[s2] = Ex[s1] \cup gen[s2] \cup con[s2] \\ Ex[s2] = y, x, a, b \cup x \cup c \\ Ex[s2] = y, x, a, b, c \end{gathered}$$

3.4.2 Dependency among variables

From the statement s (Figure 3.4), it can be inferred that definition x is derived by using y and z on operation op. It means x is dependent on the data definitions y and z. This information is necessary in order to track variables inside the function. As this tool is operating on internal representation, a lot of data definitions are in terms of temporaries. The dependency information also helps to reach actual variable in the source code, which user can understand while giving feedback.

Consider V, a set of all variables (including parameters and global variables) used in a function. Dependency relation between two variables can be denoted as function f with domain X and co-domain Y:

$$f : X \to Y \text{ where } X, Y \in V$$

For instance, for statement s (in the Figure 3.4) dependency relation of x can be defined as:

$$x(s) = f(y, z)$$

Further, if y and z have dependencies on a and b respectively then x can be defined at statement s as:

$$x(s) = f(f_1(a), f_2(b))$$

Suppose a and b are pointer parameters and x is a indirect reference. Through dependency functions x can be traced down to a and b. Hence, it is recognized that an outside address is being accessed.

In the case of killed information (Figure 3.5), dependency relation of older definition of x remains intact as that definition exists before the statement s2. Further, in the case when new definition is dependent on older definition directly or through some other variables, the dependency relation can still be defined.

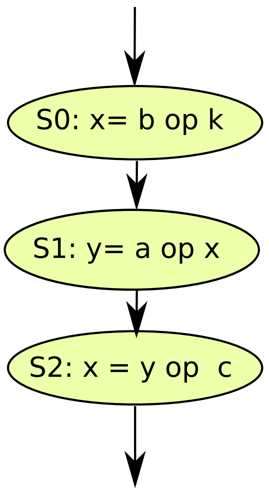


Figure 3.5: Killed definition

For instance, the dependency relation at s2 (above) for x is:

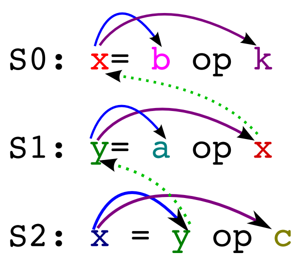


Figure 3.6: Dependency relation

$$x(s2) = f(y,c) \rightarrow x(s2) = f(f(a, x(s0)), c) \; where \; x(s0) = f(b,k)$$

By now, data definitions generated and consumed considering killed data definition information. Figure 3.6 can aid in visualizing the dependency relation between data definitions. In this figure, each data definition is shown in distinct color. It can be seen that there are two definitions of `x` and both have different color due to killed definition of older one at statement S2. In the next section, it is mentioned how a condition can affect a data definition.

3.4.3 Conditional definition

The code written inside each function is divided into basic blocks according to the control flow. Therefore, each node in the control flow graph is represents a basic block of the function. A *basic block* is a linear sequence of program instructions having one entry point (the first executed instruction) and one exit point (the last instruction to be executed). The order of execution of these basic block is defined by control flow.

Control flow can also affect the definition of a variable. One of the typical cases is `if` and `else` condition. As only one condition can be taken at a time, if a variable is defined under both conditions, it is not known, statically, which definition is taken actually until run-time. For example, for the section of code in the Figure 3.7, basic blocks B0 and B1 are under the condition in which y is being defined at statement S0 and S1 respectively. Consider, the basic block B2 which eventually is to be executed after B0 or B1. At the statement S2, definition of y is being used by x. The dependency relation of x can be defined straight in terms of y but definition of y is not certain. Therefore, all the dependency relations of y, have to be considered for x as well. Hence, the dependency relations for x and y are:

$$x = f(y,c)$$

$$y = \begin{cases} f(a,m) \quad when \quad B0 \quad is \quad taken \\ f(b,k) \quad when \quad B1 \quad is \quad taken \end{cases}$$

3.5 Inter-procedural analysis

Call graph information from compiler cannot be used as it is context insensitive. But as this thesis is focusing on the parallel task, it is required to make context sensitive

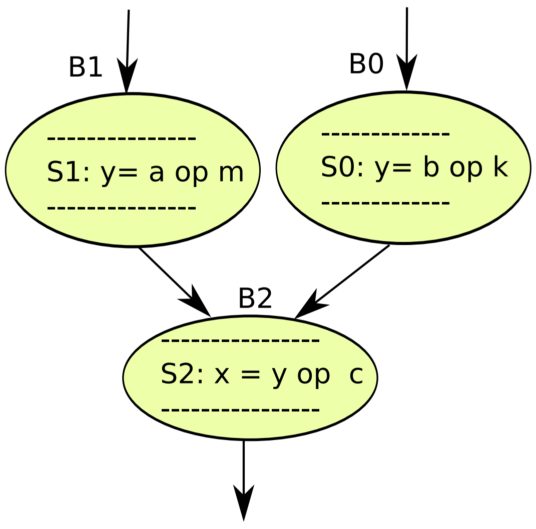
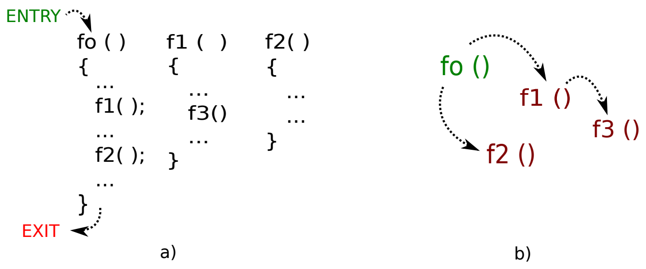


Figure 3.7: Conditional definition

call graph. The solution is already in the intra-procedural analysis. In intra-procedural analysis , information about calls which are being invoked from every function can be stored. Later, when the entry point (function) of the task is known, inter-procedural analysis can be done through each call made by that function. For instance, consider the code given in the Figure 3.8 a).



```
fo ( )      f1 (  )     f2( )
{           {           {
  ...          ...         ...
  f1( );      f3()         ...
  ...          ...      }
  f2( );    }
  ...
}
```

Figure 3.8: a) Example code for a task b) Call graph

For functions f_0 and f_1, calls are:

$$f_0 = \{f_1, f_2\}$$
$$f_1 = \{f_3\}$$

Thus, if the entry point[2] of the task is function f_0 then call graph would be like as it shown in Figure 3.8 b). It can also be defined as:

$$f_{entry} = f(f_{callee1}, f_{callee2}, ..., f_{calleeN})$$

It can also be defined in terms of dependency function. So, in this example dependencies for f_0 and f_1 are:

[2] In parallel tasks, entry point and exit point of the task is only one function which can be called as *forked* function.

$$f_0 = f(f_1, f_2)$$
$$f_1 = f(f_3)$$

Therefore, through entry point function all the other calls, which can possibly be invoked, can be reached.

3.6 Detection of data dependency

In above sections, data dependency inside functions and call graph of the transaction are build up. Using this information, data flow analysis is done between functions in order to detect data dependency between two transactions. Each external address being accessed inside the transaction is stored. Later, it is compared with other transactions. Suppose, the call graph for any transaction is:

$$f_1(l_0, g_1) \rightarrow f_2(l_1, g_2) \rightarrow f_3(l_2, g_3) \rightarrow ... \rightarrow f_{n-1}(l_{n-2}, g_{n-1}) \rightarrow f_n(l_{n-1}, g_n)$$

where:
$\rightarrow$ - data flow between functions.
$l_i \in L_i$ - set of local memory addresses which are passed on by caller function i. l_0 is set of locals from the parent function i.e. outside the transaction.
$g_i \in G$ - set of Global memory addresses.
$f_i \in F$ - set of Functions

Every address passed as a argument in the function is tracked through the dependency relation build up inside a function. All outside addresses in a function on which a data is being read or written is stored.

$\therefore$ For any transaction Ti potential data dependency set:

$$D_{Ti} = f_1(l_0, g_1) \cup f_2(l_1, g_2) \cup ... \cup f_n(l_{n-1}, g_n)$$

$\Rightarrow$ Dependency between transactions Ti and Tj:

$$D = D_{Ti} \cap D_{Ti}$$

$D = \emptyset$ when both D_{Ti} and D_{Tj} contain only read set or write set (when in same phase).

3.7 Feedback

Using inter and intra procedural analysis, there are several feedback which CiT provides to the programmer in order to have optimized code for the transaction.

3.7.1 Dependency within Loop

Dependency in a loop may act as catastrophic element for the performance of the architectures TMFab and TMFv2. As it is already explained in background chapter that in transactional memory architecture, a transaction only validates when it completes the execution of the code inside it. Hence, if there is a conflicting memory access inside a loop in transactions, then it is detected only when it completes all the iterations of that loop. Consequently, one transaction has to abort when it has completed all iterations, which may prove be really expensive. To provide this information to the programmer is important, to have an optimized code for the transaction.

3.7.2 Dependency inside a branch

Knowledge about a potential data dependency in terms of control flow can be really useful to hardware. If a potential data dependency is there in a branch then it may be speculated whether that branch can be taken or not through branch predictor, thereby, allowing hardware to take action according to the condition. For example, in the Figure 3.9, assume that there is a data dependency at node 4. If the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ is taken then the transaction has avoided the data dependency. Therefore, if scheduler get to know this situation, it can avoid validation overhead and can directly commit to the memory.

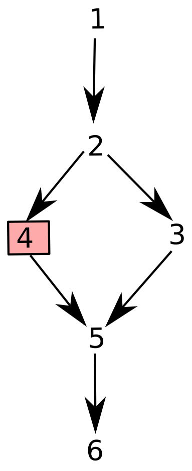


Figure 3.9: A potential dependency under condition

Each variable has information in which block and line it is used. Once the potential data dependency detected it can be checked through the control flow graph of the function. This information can be pass on to scheduler to take necessary action at run-time.

3.7.3 Load and Store

The number of load and store instructions can be inferred as the dataset. This dataset information can be useful in terms of cache size. Assuming all load and store instructions are different, the transaction can reside completely in the cache or not.

The load-store information is extracted through temporaries in the GIMPLE internal representation. Basically, each temporary variable used in internal representation eventually gets converted to registers. Rest of the variables which are declared by programmer are resides in the memory. Hence, accessing them is a load/store instruction. The total number of load and store operation are stored for each block. Using control flow graph, the maximum number of load store instructions can be calculated. However, loop should be considered but as statically it is not possible for estimating number of iterations, only one iteration is considered for each loop.

3.8 Warnings

CiT gives warning about the possible cases which may happen during execution. The following are warning descriptions:

3.8.1 Abnormal termination

Using GCC information about control flow, the CiT extracts the information about entry and exit of a basic block. It checks and warns if there is no exit from a basic block. If a basic block doesn't have any exit and it is not the terminal of the function then it is considered that it can be a abnormal termination of the program. Any abnormal termination in the transaction should mean that the transaction has to be descheduled.

3.8.2 Memory management functions

As it is discussed in section 3.2, invoking a memory management function like malloc may create a potential data dependency. Therefore, it is necessary to inform programmer about it. While making call graph inside the function, it can be checked whether there is any call to such system calls. A warning can be issued on discovering such memory management calls.

3.8.3 Recursive

Recursive functions are many times useful in programming. But it can cause an overflow of the stack or if there are speculative writes inside that function it may cause Speculative Write Buffer overflow. Also, if there is recursive call loop and if it is not programmed carefully then it can cause infinite call loop. For example, if a function A is invoking function B and further function B is invoking A then program may fall in the loop with functions A and B calling each other. To make programmer aware of this situation, the CiT checks this condition during interprocedural analysis.

CiT Tool- Architecture 4

In this chapter, the implementation of CiT is discussed in detail. First, CiT plugin placement in the GCC architecture is briefly explained. Thereafter, CiT architecture is introduced. Intra and Inter procedural analysis is illustrated in detail. It is followed by explanation of algorithm to detect potential data dependencies. In the end, process of extracting feedback information is discussed.

4.1 CiT plugin placement

The CiT tool is a plugin for GCC 4.5 and works on GIMPLE internal representation (IR) which is at middle end of GCC architecture. In GCC, there are several passes which work on GIMPLE IR; a set of them do inter-procedural optimization and others intra-procedural optimization. This (CiT) plugin is inserted between inter-procedural optimization and intra-procedural optimization as it is shown in Figure 4.1.

To detect a potential data dependency a data flow inside and within functions, i.e. intra and inter procedural, is required as it is already explained in chapter 3. Since there is no such data flow in GCC specifically for transactions or parallel tasks, it is required to implement it. To develop the data flow between functions call graph is required which can be extracted through intra-procedural analysis. However, GCC also has call graph information but it is context insensitive which is not suitable for our objective. Therefore, plugin should be placed as GIMPLE intra-procedural pass. At intra procedural optimization, intermediate language becomes more complex in terms of temporary variables. Thus, CiT plugin is placed just before intra-procedural optimization passes (Tree SSA is the first pass of this kind).

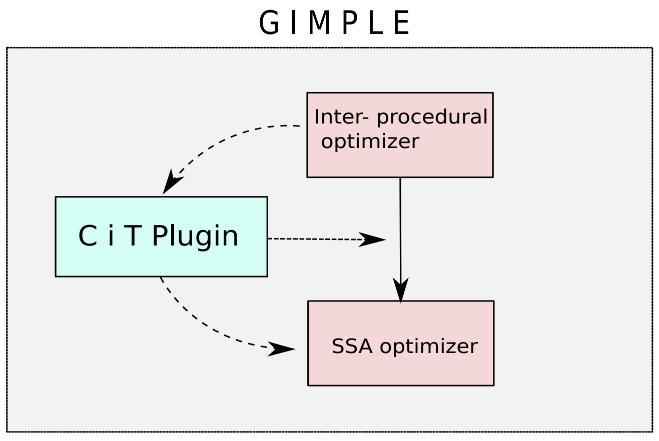


Figure 4.1: CiT Plugin placement

4.2 CiT Architecture

CiT plugin composed of various passes as shown in Figure 4.2: Initial analysis, Flow sensitive analysis, Context sensitive analysis.

- **Initial Analysis**: All declarations of functions (with their arguments) and outside functions, like global variables, are analysed in inter-procedural analysis(IPA) in GCC. Therefore, to get the information about these declarations from GCC, this pass is inserted as a GCC inter-procedural analysis pass.
- **Intra-procedural analysis**: To analyze each function individually, this pass is inserted as an intra-procedural in GCC. In this pass, each variable at every statement is extracted with all information related to it and stored it into a new database. Further, it builds up the flow sensitive data flow tree using dependency among variables and kill information as explained in the chapter 3. Moreover, control flow graph is made using the basic block details in GCC to help in developing the data flow and feedback.
- **Inter-procedural analysis**: This pass is used to build up inter-procedural data and control flow using details from previous pass for each transaction. During this analysis, all addresses (which may occur as potential data dependency) and feedback information are stored. Later, another analysis is done to detect conflicts and report is generated.

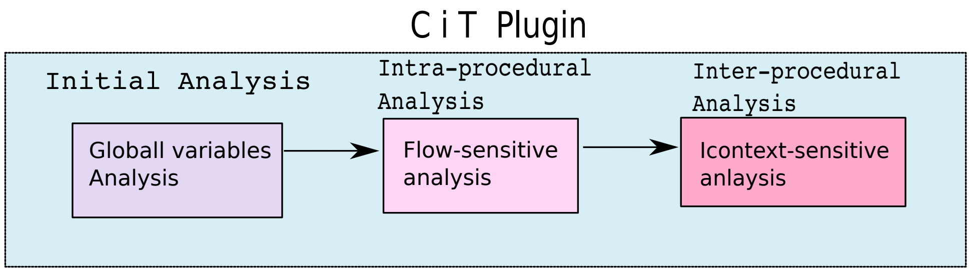

C i T Plugin
Initial Analysis
Intra-procedural Analysis
Inter-procedural Analysis
Globall variables Analysis
Flow-sensitive analysis
Icontext-sensitive anlaysis

Figure 4.2: CiT Architecture

4.3 Initial analysis

The main purpose of this pass is to have details of global variables and function declarations with their arguments. Global variables are one of the most potential data dependencies as they can be accessed anywhere in the program. Thus, the set of global variables is made up so that it can be used to detect a global variable while analyzing variables in intra-procedural pass.

In GCC, during its inter-procedural analysis, information about global variables is stored in the linked list data structure called `varpool_node`. Since, it cannot be accessed in intra-procedural passes, it is required to make a new database for them which can be accessed during data flow in the next pass. The new data structure for the same is named as `gbs`. In this database, every global variable is stored in their corresponding type of bin. Here, a *bin* is an array type of data structure `gbs` and for each data type, there is a separate bin. For example, a global variable is of structure(record) type then it is put into bin called `recglobal`.

For inter-procedural data flow analysis, it is required to know the arguments of the functions as to identify the data transfer. GCC stores all declarations information about functions in a tree which can be accessed through a pointer to structure called `cgraph_node`. Each node in the tree is a function. Since arguments of the function are also stored as a tree, `DECL_ARGUMENTS` macro is used to get the address of the root of the tree of arguments. By traversing the tree, all arguments can be accessed and stored into a separate database for functions.

4.4 Intra-procedural analysis

In intra-procedural analysis, every function is analyzed individually and all details are stored in an array data structure called $func$ (refer Table B.2 in Appendix C). It contains information related to variables, parameters, callee functions with arguments pushed to them, and basicblocks. In this analysis, variables are extracted through GCC trees which are used in building the data flow. Also, the control flow of the function is analyzed.

In order to perform aforementioned analysis, GCC internal macro `FOR_EACH_BB` traverses through all basic blocks. Subsequently, to work at granularity level of statements, each statement is processed through statement iterator as given below.

```
 FOR_EACH_BB(bb) {          // Traversing each basic block
  gimple_stmt_iterator gsi;
  gsi = gsi_start_bb(bb);
  for (gsi; !gsi_end_p(gsi); gsi_next(&gsi))   // Statement iterator
    processGimpleStatement(gsi_stmt(gsi), &gsi);
}
```

Each statement is further identified as an assignment (`GIMPLE_ASSIGN`), call (`GIMPLE_CALL`) and condition (`GIMPLE_COND`). To extract variables, `GIMPLE_ASSIGN` and `GIMPLE_COND` is used. To make call graph with arguments, i.e. for context sensitivity, `GIMPLE_CALL` is useful.

4.4.1 Variable information extraction

Before building the data flow, it is necessary to extract a variable with its attributes from GCC trees. Most of the variables are extracted by analyzing `GIMPLE_ASSIGN` statement. `GIMPLE_ASSIGN` can have three types of statements: `GIMPLE_BINARY_RHS` (operands with binary operation), `GIMPLE_UNARY_RHS` (operands with unary operation) and `GIMPLE_SINGLE_RHS` (single assignment). Each operand is taken individually and analyzed using a general framework called *processOperand*. In this framework, all types of declarations (given in Table B.3 in Appendix C) are checked through macro TREE_CODE of the operand and stored with their attributes information into database in the form of variable data structure (Table B.1 in Appendix C) as sample code is given below.

```
 processOperand (operand) {
   switch(TREE_CODE(operand)) {
     case VAR_DECL: variable[].name= DECL_NAME(operand);
     case INDIRECT_REF: variable[].referas =INDREF ;
     case ARRAY_REF: variable [].type = AR;
  }
}
```

There can be many combinations of declaration for a variable, e.g. *P, where a pointer is dereferenced. It is indirect reference and also a variable declaration. For such cases, GCC extends the tree and both (or more) information can be extracted through nodes. Some important cases which are required to know are mentioned below.

Dereference of a pointer It is necessary to know the pointer declaration in the case of indirect reference. Thus, processOperand is again invoked to get the exact declaration, e.g. indirect reference of a parameter or structure.

Array offset In the case of an array, apart from a variable declaration the offset which represents a particular element in the array can also be extracted. It may be a constant or another integer type variable. Using the offset, it may be predicted that which elements are being accessed or which variable is responsible for that.

Structure field Sometimes it is not enough to know a structure variable as it is comprises of many fields. Therefore, to be more precise in terms of memory it is necessary to know which field is being accessed.

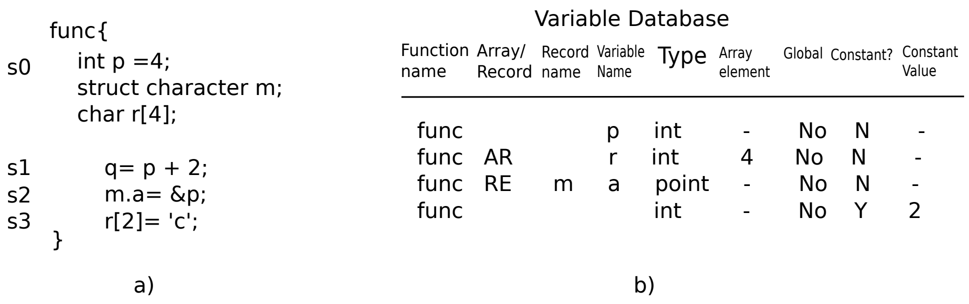
Each variable can be of three types: pointer, integer, real. It is important to know the type as it depicts whether a declaration holds a address or a value. For example, in the statement below:

```
int* p;
p = &a;
*p = b;
```

p is a pointer type which is holding address of a but *p is a integer type declaration which is holding the value b. Therefore, *p is considered as a new variable data definition which is different from p. In the Figure 4.3 b) an example of a database created for source code in Figure 4.3 a) is given.

4.4.2 Building Dataflow

In creating variable database, data flow equation 3.1 mentioned in the chapter 3 must be applied. This is done by following conditions:



```
     func{
s0      int p =4;
        struct character m;
        char r[4];

s1          q= p + 2;
s2          m.a= &p;
s3          r[2]= 'c';
     }
```

a)

Variable Database

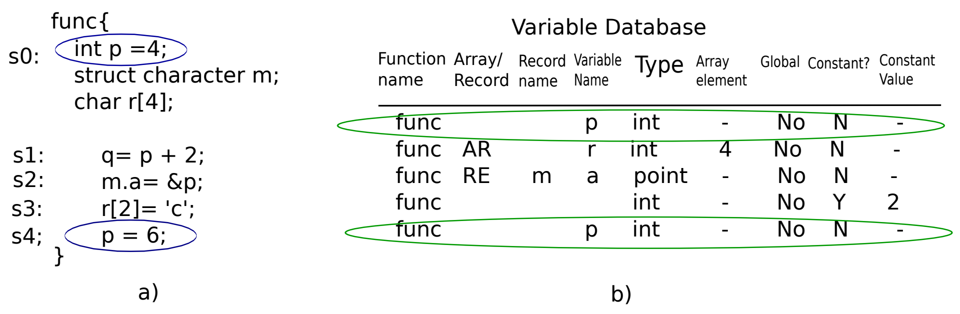
Function name	Array/ Record	Record name	Variable Name	Type	Array element	Global	Constant?	Constant Value
func			p	int	-	No	N	-
func	AR		r	int	4	No	N	-
func	RE	m	a	point	-	No	N	-
func				int	-	No	Y	2

b)

Figure 4.3: a)Source code b) Variable database

Generation and Consumption : According to eq 1, if set $En[s]$and $use[s]$ have common variable, then variable in use[s] is not pushed to the database. Instead, the variable in$En[s]$ is considered. For instance, consider the code given in the Figure 4.3. At statement s1, $En[s1] = \{p\}$ and $use[s1] = \{p, 2\}$, thus, in database existing variable p is considered. But at the same statement, constant "2" which didn't exist before s1 is pushed to the database.

Kill information : It is possible that $gen[]$ and $En[]$ have common elements. It means a data definition is generated which is also being consumed. Thus, a new value is being assigned to a variable which has already been read before. Since, it is sequential execution inside a function, the older value is not used further. Therefore, it is necessary to create a new variable in the database. Thus, the database has now two duplicate variables. Figure 4.4 is depicting an example of it. At the statement s5, $En[s5] = \{p, q, 2, m.a, r[2]\}$ and $gen[s5] = \{p\}$. Hence, new variable p is pushed to the database. Now, generated definition of p is killing the older definition which is not pop out of the database. It is important to have both definitions of the variable, since dependency relation of older definition has to be maintained. It is explained in detail in the next section.



```
      func{
s0:     int p =4;
        struct character m;
        char r[4];

s1:         q= p + 2;
s2:         m.a= &p;
s3:         r[2]= 'c';
s4;         p = 6;
      }
```

a)

Variable Database

Function name	Array/ Record	Record name	Variable Name	Type	Array element	Global	Constant?	Constant Value
func			p	int	-	No	N	-
func	AR		r	int	4	No	N	-
func	RE	m	a	point	-	No	N	-
func				int	-	No	Y	2
func			p	int	-	No	N	-

b)

Figure 4.4: a)Source code b) Variable database

Dependency between variables : It is necessary to know the dependency of variable in order to estimate exact value by tracking dependencies of the variable. Consider the source code below, for instance, we need to know the value of variable a.

```
  c = 7;
  b = c;
a = b + 3;
```

Now, variable `a` is dependent on variable `b` and constant `3`. Further, `b` is dependent on `c` which is defined as `7`. It can be observed that the ultimate value of `a` is `7 op 3`, since `c` has value of `7` and `b` is equal to `c`. As op can also be known, then actual value is calculated which is `10` in the case of '+'.

4.4.2.1 Approach

In order to make dependency relation, two approaches are conceived. In the approach I, right hand side variable, i.e. assigning variable, points to the left hand side variables as shown in Figure 4.5 a). On the other hand, Figure 4.5 b) shows the approach II in which left hand side variables are pointing to right hand side variable. In the approach I, a variable can be the dependency for many other undetermined number of variables. Thus, a variable in the database needs unknown number of pointers. Dependency relation is made from source to destination. Therefore, all the destinations can be reached. But in approach II, a variable is dependent to maximum two operands. However, operands may have their own dependencies which must be taken care of at the time of their assignment. Therefore, for any assignment there are maximum two dependencies which limit the number of pointers required to two. In the approach II, dependency relation from destination to the source is build up. Hence, from any destination the source can be reached.

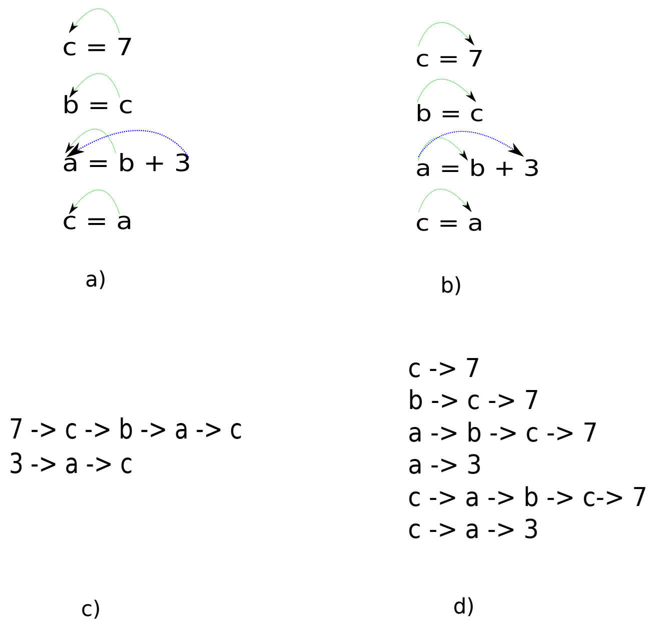


Figure 4.5: a) Approach I b) Approach II

Apart from implementation difficulty in the approach I as it needs unknown amount of pointers, there is one conceptual problem for the requirement of this thesis. The problem is in order to track each and every dependency, we have to start with the source. For all reachable destinations, context can't be cleared. For instance, consider the approach I used in the Figure 4.5 a). In the dependency relation in Figure 4.5 c) constant 7 and 3 can be reached to `a` but with out any context of `a` being used. On the other hand, through dependency relation in Figure 4.5 d), developed using approach II (Figure 4.5 b)), `a` is reaching both 7 and 3. Now, if operation `+` is recorded in database of `a` when it is being defined then the context is known. Therefore, context can be known in approach I.

Although, the approach I can be useful when parameter is a pointer. Through parameter pointer all destinations where it is dereferenced can be reached. However, the case where dereference of pointer is being assigned, is not reachable since its source is different. Therefore, it can only be helpful when dereference parameter pointer is being read or address in the pointer is assigned to other pointer.

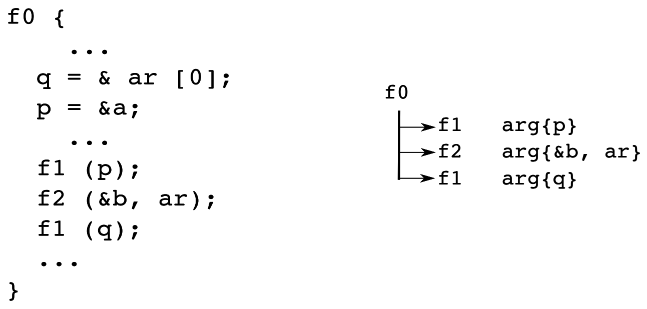
In approach II, source can be reached from any destination. It maintains the sequential consistency, i.e. at every statement it considers all the definitions reached to that statement. It is quite useful in the point-to analysis where it tries to include all offsets. Hence, the approach I fits perfect to our requirement. To hold the right hand side operands in left hand side variable database, two pointers `variable *value` and `variable *value1` are used. By accessing these pointers, dependency of the variable can be reached.

Special Cases There are special cases where dependency relation is considered in different style.

- **Address reference (&p)**: An address of the variable can be assigned to a pointer using an address operator. For this case, a special pointer `variable *addr` is included in the variable database. Thus, a pointer variable points the variable whose address is being assigned but in context is different. While accessing the pointer during analysis, through `variable *addr` the variable can be reached whose address is stored.
- **Dereference of pointers (*p)** When the pointer is dereferenced, a new variable is generated which points to the pointer variable ,which is being dereferenced, using the special pointer `variable *addr`.
- **Condition** When a variable is dependent on two definitions of other variable in condition then special pointers `variable *cond1, *cond2` is used.

4.4.3 Building Call graph

A call to a function can be recognized using GIMPLE_CALL statement. Callee function name is extracted using GCC internal function `gimple_call_fndecl(stmt)`. All calls in a function is recorded in the database of the function, i.e. `function` ,as first come first basis so as to maintain sequential consistency. Basically, `fnptr` array field in the function structure points to all callee functions. Moreover, the arguments can also be extracted which is being pass to the callee function using `gimple_call_arg(stmt,argno)`. As we are only interested in checking arguments which are passing the addresses, all the pointer type variables are only considered. To record arguments, `argu` array points to all variables which are arguments to a particular function call. Figure 4.6 is depicting an example call database of a function. This information is really useful as it has context calling which is required in our work.



```
f0 {
    ...
  q = & ar [0];
  p = &a;
    ...
  f1 (p);
  f2 (&b, ar);
  f1 (q);
  ...
}
```

Figure 4.6: Building context sensitive call graph

Normally, only functions which are defined in the source code are considered to be recorded. All library functions and system calls are ignored. System calls related to memory management can be a cause of data dependency which cannot be detected as it is explained in the chapter 3. Therefore, these calls are recorded so as to issue a ***warning*** if a function which is invoking them comes under transaction. Further, it recognizes function calls which spawns a transaction or parallel task. For example, if there is a call like `transaction_start` which spawns transactions, it searches call arguments for the function pointer (the entry point of the transaction). Further, variables are recorded which are being passed to it as arguments.

4.5 Inter-procedural analysis

In inter-procedural analysis, addresses which are potential data dependency in the transaction is detected by using context sensitive call graph. In the section 4.4, all the callee functions is recorded with the context, i.e., argument being passed. Further, the entry point of the transaction is also perceived through the parent function which is spawning transaction. Using this information, transaction call graph can be developed. For example, consider the transaction in the source code given in Figure 4.7. In this code, `main` function is spawning transaction with entry function `f0` and `m` as argument to it. The function `f0` is further invoking functions `|f1|`and `f2` with arguments `a` and `b` as context. To simulate the execution of the code in the transaction, the call graph is made up accordance to the location and context, functions are being invoked with. In this example, f0 is calling f1 which is further calling f3. Subsequently, f2 is being invoked by f0 , which eventually invoked f3. Hence, call graph for transaction is developed as it is shown right hand side of the Figure 4.7.

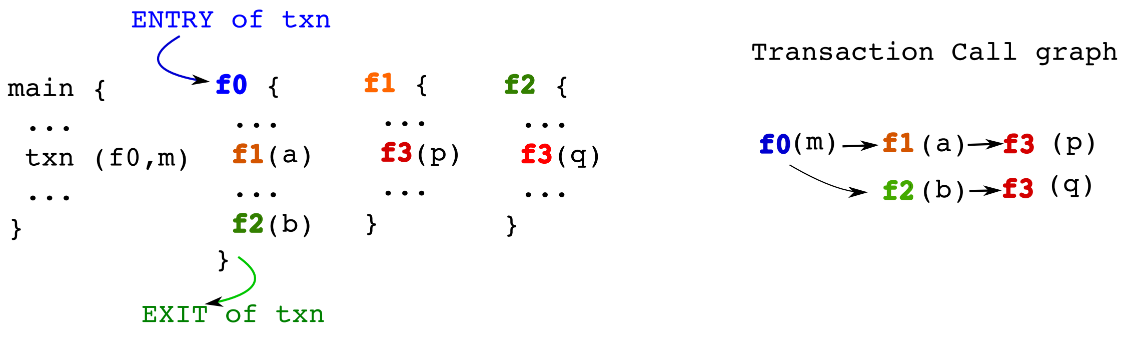


Figure 4.7: Call graph of a transaction

4.6 Address detection in Transactions

In the intra-procedural analysis, variable information is extracted and stored into a database. Dataflow equations are considered while building up the the database. Further, dependency relation is made up among variables. Also the call graph for each function is recorded which is used in inter-procedural analysis . Therefore, to detect the dependencies every information is available.

In order to detect the potential data dependency, it is required to know whether the address being accessed inside the transaction is local or not. To check this, each variable in the function is tracked. It is done using the dependency graph created among variable in the data flow. The source of the variable can be outside the function. This will be the case for most of the potential data dependencies except global variables. Therefore, it is necessary to use the call graph of transaction.

The process of analyzing the transaction is being depicted in Figure 4.8. The process starts with analyzing the entry function. First, each variable in the entry function is tracked and the address is recorded which is not local to transaction and might be a potential data dependency in transaction's address collector or bin. By now, entry is analysed but there can be a case where other calls in the call graph might be accessing shared data. Therefore, using the call graph, the callee function is connected with context to the caller to do data flow analysis i.e. tracking variables. Further, the process is repeated for calls inside the callee functions. This process is done for each call invoke by any function inside the transaction.

In data flow analysis, objective is to track: 1) external addresses which can be accessed through parameter, 2) global variables. All indirect references are main concern, where a pointer is dereferenced and the address, residing in it, is accessed.

4.6.1 Implemented algorithm

In the data flow analysis algorithm (Table 4.1), each variable is tracked if it is a indirect reference. In tracking algorithm (Table 4.2), every dependency of concerned variable is tracked to reach the source. For example, in the following code, if we need to reach the source of indirect reference `*m`, algorithm checks pointer containing the address i.e. `m`. After reaching `m`, it can reach to another pointer `l` and also the offset `4`. Here, `l` is first and `4` is second dependency of `m`. The offset is stored to have precision. Using

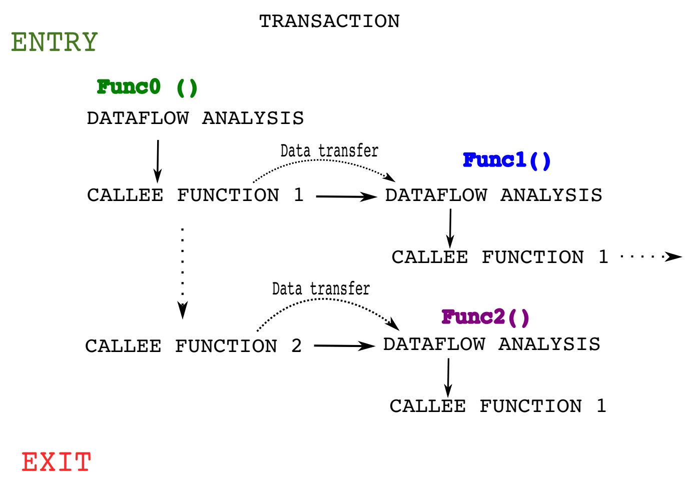


Figure 4.8: Methodology to detect data dependency

the dependency of pointer variable `l`, parameter pointer `p` is reached which further connected to its dependency in the caller function. So, if caller function lies outside transaction then reached variable through pointer parameter `p` is the source. The source is stored with offset `4`. It is compared with stored addresses in other transactions to detect the dependency. In the case of structure, when there is indirect reference to a field then it is stored as offset. Although, tracking continues to search for the actual data structure variable.

```
func ( p ) {

  int *l;

  l = p;

  m = l + 4;

  *m=7

  *p = 5;

  fn2 (m,l,p);

}
```

4.7 Control Flow

A function is composed of several basic blocks which has linear sequence instructions with one entry and one exit instructions. Conditions inside a function decide which basic block to be executed. Thus, the flow of these basicblocks is called control flow of the function. As it is mentioned in B.1, each basic block can be accessed by traversing the tree in which they are store. In GCC, information about preceding and succeeding basic blocks is stored in the form of a tree, where a node is a basicblock and each edge

Algorithm for **data flow analysis**

```
for all variables in function
{
     if cur. var. is refered as INDIRECT referece
    {
            source var. = track (cur. var.)
            if source var. outside of txn.
                txn. add. = source var.
                if cur. var. is record
                    txn. add. offset = cur. var.
    }
    if cur. var is GLOBAL
        txn. add = cur. var.
}
```

Table 4.1: Data flow analysis algorithm

out of that node are successors of it. Using macro `FOR_EACH_EDGE`, all the succeeding or preceding edges can be accessed. If a basicblock has condition as a last instruction, it has two edges: true and false. It can be checked which edge is true through the flag `EDGE_TRUE_VALUE` and similarly `EDGE_FALSE_VALUE` for false. However, in the case of switch number of edges is equal to number of cases. This can be recognized when statement is `GIMPLE_SWITCH`. Therefore, the control flow graph can be build up using all the aforementioned information as it being depicted in Figure 4.9.

```
function {
 a=1;
 if (a>2)
  b =3;
 else
  b = 4;
 switch (b)
 {
   case 1: c =3;
   case 3: c = 5;
   default: c = 6;
 }
 a=c;
}
```

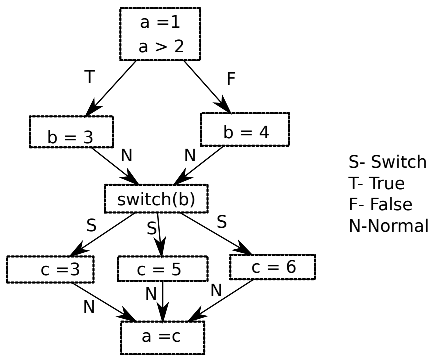


Figure 4.9: Control Flow graph

A basicblock must have atleast one edge coming out of it. If a basicblock is the terminal then there is no edge out of it. However, many times a normal basicblock may not have any succeeding edge, e.g., in the case of exit (a system call which terminate the program) or assert. This type of termination can be termed as "Abnormal termination" and discovery of it will be given to the user as ***warning***.

Algorithm for **tracking** a variable

```
while variable exists
{
    if var. is a parameter
    {
        if var. resides outside txn
            return var.
        else
            var. = dependency of var. in caller function
    }
    else // if var. is not a parameter
    {       if var. resides outside txn
        {
            if var. is pointer
            if var. has 2nd dependency
                tx. addr. offset = 2nd dependency
            if var. has 1st dependency
                return 1st dependency of var.
        return var.
        }
        if var. is pointer
            if var. has 2nd dependency
                tx. addr. offset = 2nd dependency
            if var. has 1st dependency
                if var. is para
                    return track var.
        else // var is not a pointer
            var. = 1st dependency of var.
             if var. is paratmeter
                return track var.
}
```

Table 4.2: Tracking algorithm

4.8 Extraction of information for feedback

Warnings related to recursive functions and loop, abnormal termination and suage of memory mangement function is discussed in previous sections. In the following sections, extraction of loop information and load-store information is explained.

4.8.1 Loop Information

Loops in GCC are broken into several basicblocks at GIMPLE level. Thus, loops can't be recognized until unless control flow is analyzed. However, GCC while breaking the loop store its characteristics and mark each basicblock according to them. The basicblock which is entry to the loop body is named as *header* and the back edges leading to the entry block from inside the loop is called *latch*.

Loop analysis done by GCC can be used by initiating `loop_optimizer_ini`. In this analysis each basicblock has information regarding the loop. A basicblock data structure contains loop number if it comes under a loop. Also, it has information regarding loop depth, i.e. if a basicblock is in nested loop. Now, this information is passed on to the variables being defined or used in that basicblock. Therefore, every variable has information whether they are in a loop or not and at which depth. This is one of the *feedback* given to the programmer if the variable is a potential data dependency.

4.8.2 Load-Store instructions

Load and store can be percieved in GIMPLE internal representation. All temporaries are eventually converted into registers at assembly langauge. Therefore, all variables declared by programmer resides in the memory. Thus, load and store instructions is used to fetch them form he memory. CiT does make difference between temporaries and actual variables. Each variable has information in the database about being temporary or actual variable.

In order to calculate maximum number of load-store instructions, it is necessary to consider the control flow which decides which instructions are going to be executed. As control flow graph is developed in terms of basic blocks, number of load-store instructions for each basic block is calculated. Now, control flow graph can be considered as weighted directive graph with number of load-store instructions as weight as shown in Figure 4.10 a). All paths are considered to find out the longest weighted path. At each node, all possible path is taken as one by one. For instance, Figure 4.10 b), from node 1 red is chosen first. Moving on red path, again at node 3 there are two different. First, teal color path is taken and then blue path. Similarly, when all paths are covered with node 3, it returns back to node 1 to cover all possible path through green. While traversing through the graph, maximum weight is calculated.

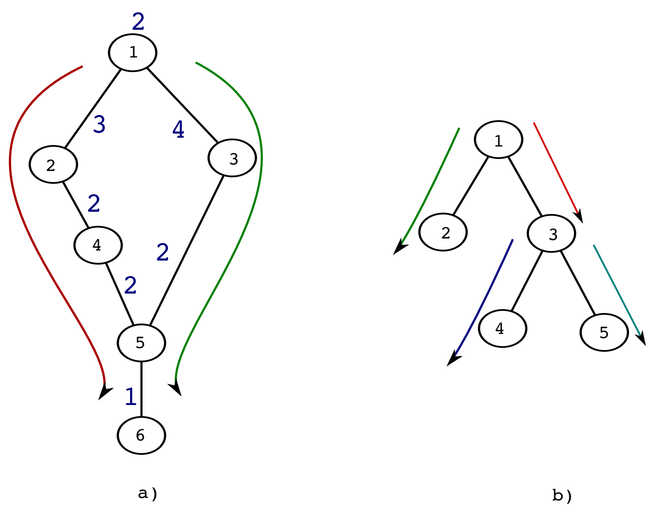


Figure 4.10: a) Weighted control flow graph b) Conditions inside control flow

A control flow graph can't be always directed acyclic graph. It become cyclic when there is a loop inside it. The methodology explained above will not work since a node find itself in the loop and it again try to search the same path. To overcome this problem, header information from loop can be used. Every loop start and end with header. It is the condition which decides whether to take iteration or not. Using the loop information, the basic block with header is named as header (H). The approach is to skip the header and go out of the loop. For instance, in the Figure 4.11, from node 1 which is header if path taken for node 2 then after node 6 it will again reach to node 1 header. To avoid it before reaching to node 1 it check the next node if it is header then it skip and choose the other path. Therefore, node 6 skips node 1 and reach node 3. Thereby, it covers one iteration of the loop and avoid to get stuck in infinite loop.

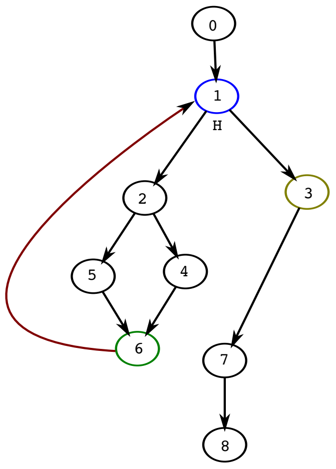


Figure 4.11: Loop in a control flow

Results 5

This chapter describes the performance of the CiT tool when it was tested with a custom made application, more realistic application found in literature. Moreover,the first application was tested on the TMFv2 simulator, and the results are analyzed.

5.1 Experimental Evaluation

The evaluation of the CiT tool is done on the basis of its efficiency in detecting data dependencies. Efficiency is measured in terms of detecting dependencies which are *static*, and will certainly produce a conflict. To verify the CiT tool, applications are required in which data dependencies are already known. These applications should have characteristics which can bring out the feedback aspects and all types of data dependencies that the tool can detect. In order to do so, we have written a custom application. Further, several applications from the STAMP benchmark suite [24] are also chosen. Based on the feedback remarks are made for each application.

5.1.1 Custom Application

All data dependencies which are induced into the custom application are through:

1. Global variables.
2. Accessing addresses which are passed through parameters.
3. Accessing elements through global pointers in the heap section of the memory, like for example, in a linked list.

In this test, there are six parallel tasks. All data dependencies between tasks are already known. The dependency relations between tasks are shown in Figure 5.1 and explained below:

Task 1 and Task 2 : Task 1 and Task 2 are accessing the same address in the stack of the `main` function, which is being passed to both tasks as an argument. Task1 is writing on the address by passing it to the `callee` function. However, Task2 is accessing the address in itself.

Task 2 and Task 5 : Task 5 is reading the address of a local variable in the function which is outside the transaction, while Task 2 is modifying it.

Task 2 and Task 3 : Both tasks are accessing the same array `C`.

Task 1 and Task 4 : Task 1 is writing on array `A` and Task 4 is reading the same array. However, they are not accessing the same element.

Task 3 and Task 4 : Task 3 and Task 4 both are writing on a field of the structure in the function.

Task 5 and Task 6 : Task 5 is generating a node of the list and Task 6 is changing the key of the given node.

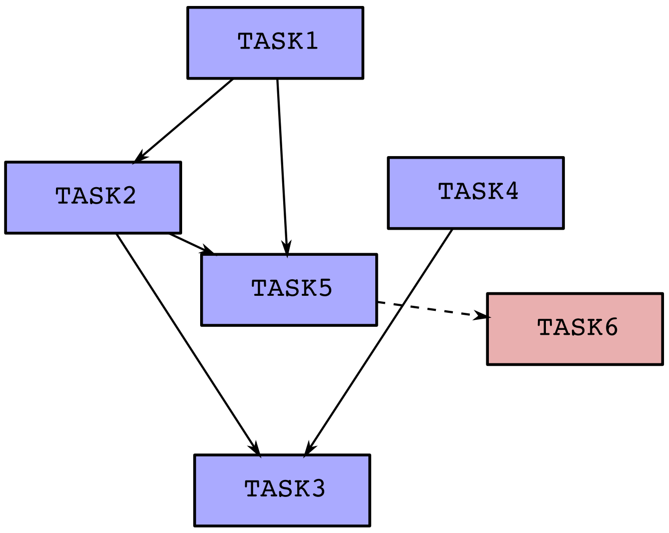


Figure 5.1: Dependent tasks in Test application

The application was compiled with the CiT tool and created the dependency graph of Figure 5.2. This graph is similiar to the one depicted in Figure 5.1. In Table 5.1, the results are almost as expected except in one case. In case there is an array access, the CiT doesn't take the offset into consideration and shows dependency if any element is being accessed . Therefore, Task 1 and Task 4, which access different elements, are shown as dependent.

Type	**Expected**	**Result**
Static conflicts	5	6
Dynamic conflicts	2	2

Table 5.1: Result of the CiT tool

All the green lines represent static dependencies between tasks, while the red lines are depicting dynamic dependencies. Since Task 5 and Task 6 both are accessing the global pointer variable, they might access addresses which are residing in the heap section. Therefore, the CiT tool is showing dynamic dependency. In case there is a memory management function being used, by a transaction a warning is produced that indicates a potential dynamic dependency when another transaction uses the same function. In this application, only Task 6 is invoking the function and subsequently the

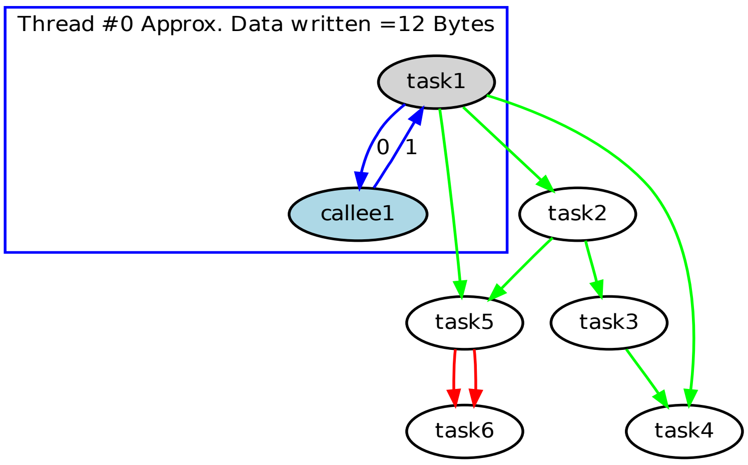


Figure 5.2: Graph generated for test code

Test Application	
Dependency under loop	2
Warning abnormal termination	Yes
Warning recursive	No
Warning memory management system calls	Yes

Table 5.2: Feedback for custom application

warning can be ignored. An abnormal termination warning is generated when system call `assert` is being used.

```
WARNING: memory management function is being used
WARNING: Abnormal Termination found
```

5.1.2 Realistic Applications

There are many realistic applications in STAMP benchmark suite [24] version 0.9.10 in which static conflicts are known through [20]. The native (non-simulator) input sets of the STAMP benchmarks suite are used. In order to test the tool, we have chosen two applications, Kmeans and Bayes, with different characteristics as shown in Table 5.3.

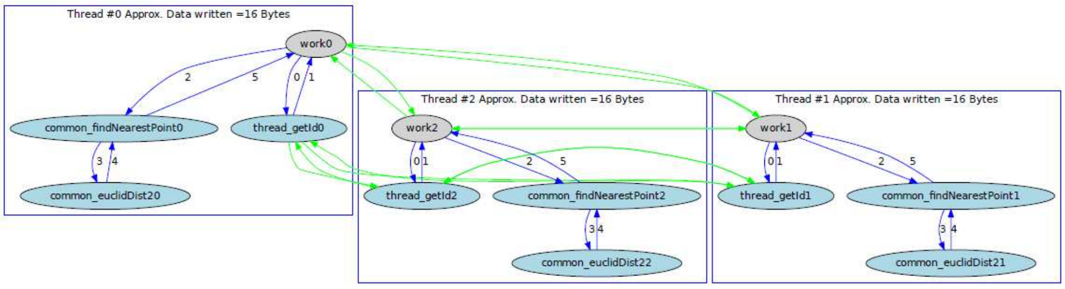
5.1.2.1 Kmeans

Kmeans groups objects in an N-dimensional space into K clusters. In the transactional memory version of it, the update of cluster's center that occurs in each iteration

Application	**Tx Length**	**R/W Set**	**Tx Time**	**Contention**
bayes	Long	Large	High	High
kmeans	Short	Small	low	low

Table 5.3: Characteristics of Kmeans and Bayes [24].

is protected through transaction. The application has only one transaction, therefore, is compiled using GCC with the CiT plugin integrated. The output of the CiT tool is depicted in the following graph in which the green lines show the data dependencies. The two static data dependencies were then compared with the results of [20] and found to be the same. In addition to those, the CiT is depicting also dynamic data dependency. According to STAMP benchmark paper [24], the variable which is being detected as a dynamic dependency by CiT, can't be changed by other transactions. The reason for which CiT is detecting it as a dynamic dependency is that the argument passed to both transactions is the same. If the argument is changed, the CiT doesn't show any dynamic dependency. According to the feedback report, which is summarized in Table 5.4, both static dependencies are found inside the loop and also inside the branch. This information is confirmed by the control flow graph inside the transaction as it shown in B.1 in Appendix C. There can be a possibility when loops are not taken then the static conflicts may not occur. Therefore, even static data dependencies are not always the cause for conflicts, which is contrary to the definition of always conflicting atomic sections in this paper[20].



Static conflicts	2
Dynamic conflicts	1
Maximum load and store assuming one iteration per loop	78
Dependency inside loop	2
Dependency inside branch	2
Warning abnormal termination	No
Warning recursive	No
Warning memory management system calls	No

Table 5.4: Feedback report for Kmeans

5.1.2.2 Bayes

Bayes is a machine learning algorithm. It is focused on learning the structure of a Bayesian network. In the transactional memory version, Bayes has two transactions containing different methods: createTasklist and learnStructure. In [20], 5 static conflicts are found, however it is not mentioned that which methods of transactions are considered. We have considered four transactions, to predict conflicts in each combination of methods. Figure 5.3 is showing the graph buildup between createTasklist and learnStructure by compiling Bayes with the CiT tool.

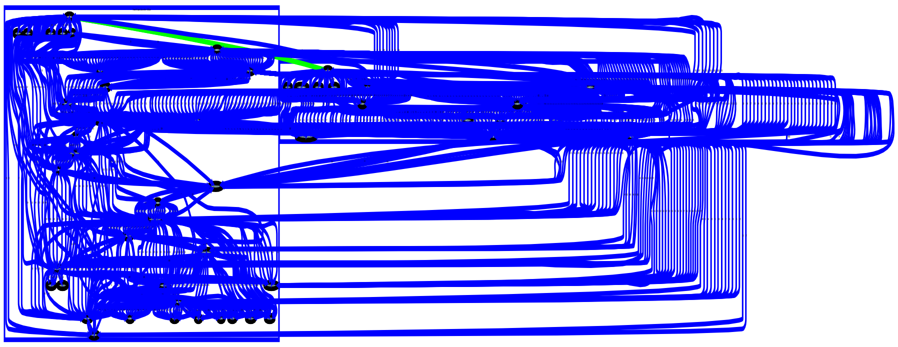


Figure 5.3: Control flow of learnstructure and createtasklist

Transactions containing the method createTasklist Two static dependencies between transactions having the same method `createTasklist` are found. As depicted in Table 5.5 , both are in the loop. Assuming that the branches are taken then the percentage of performance degradation will be dependent on the number of iterations.

Transactions - createTasklist and createTasklist	
Static dependencies	2
Dynamic dependencies	0
Maximum load and store	1122
Dependency inside loop	2
Dependency inside branch	2
Warning abnormal termination	No
Warning recursive	No
Warning memory management system calls	No

Table 5.5: Feedback report for Bayes transaction createTasklist

Transactions containing methods createTasklist and learnStructure In transactions with methods `createTasklist` and `learnStructure`, 2 static conflicts are found. Besides that, there is a warning about memory management functions being called. However, only transaction with method `learnStructure` is invoking as it is not found with `createtasklist` mentioned in the above section. As it is mention in the Chapter 3 section 3.2, if two transactions are using memory management system calls then it may cause a dynamic dependency. Therefore, it is safe to use memory management system calls when `createtasklist` and `learnStructure` are transactions. As Table 5.6 is depicting, it is also found that there are warnings about abnormal termination and recursive functions. When abnormal termination occurs then the amount of work done till termination is wasted. Therefore, if it happens inside of a transaction, then the processor should not validate and commit but rather should be descheduled.

Transactions - createTasklist and learnStructure	
Static dependencies	2
Dynamic dependencies	0
Dependency inside loop	2
Dependency inside branch	2
Warning abnormal termination	Yes
Warning recursive	Yes
Warning memory management system calls	Yes

Table 5.6: Feedback report for Bayes transactions createTasklist and learnStructure

Transactions with method learnStructure In transactions with the same method `learnStructure`, 5 static data dependencies are found (Table 5.7) . Since both transactions are using memory management functions, there is a possibility of having dynamic data dependencies. In these transactions, the programmer needs to be careful about using recursive functions as they can overflow the speculative write buffer. The programmer needs to check whether the arguments are being passed to the function.

Transactions - learnStructure and learnStructure	
Static of dependencies	5
Dynamic dependencies	0
Maximum load and store	2870
Dependency under loop	5
Dependency under branch	4
Warning abnormal termination	Yes
Warning recursive	Yes
Warning memory management system calls	Yes

Table 5.7: Feedback report for Bayes transaction learnStructure

It can be observed from Table 5.4 and Table 5.7 that load and store information is representing the size of transactions. According to the above results, kmeans is much smaller in size than compared to bayes. Also, it can be inferred from the resulting control flow graphs of kmeans and bayes that the occurrence of context switching is much more often in bayes. However, it is assumed that loops have taken only one iteration. Therefore, the number of context switches may change according the number of iterations and branches taken if there are calls inside them. To avoid context switching the programmer can check if any function can be in-lined. As mentioned earlier in Table 5.3, bayes has higher contentions than kmeans, the CiT tool report is also suggesting the same in terms of static dependencies and number of warnings related to dynamic data dependencies.

5.1.2.3 Non blocking Linked list

The non blocking linked list implementation [14] is used in applications where there can be multiple accesses to a single list. It is chosen for testing the tool since two tasks are accessing the same linked list, and therefore there is a possibility of having dynamic dependencies between them. Also, the insertion method is using the memory management system call `malloc` which can again cause a potential data dependency, in case there are two requests for insertion. This implementation with two tasks, which can call both insertion or deletion method, is compiled with the CiT plugin. The output graph of the CiT tool is shown in Figure 5.4. Table 5.8 is depicting results, in which it can be seen that there are two dependencies between task, which are basically global pointers. These data dependencies are taken as dynamic since the list can be accessed and modified using global pointers. Furthermore, both transactions are using memory management functions which will cause a dynamic data dependency, since both transactions are working on the same list.

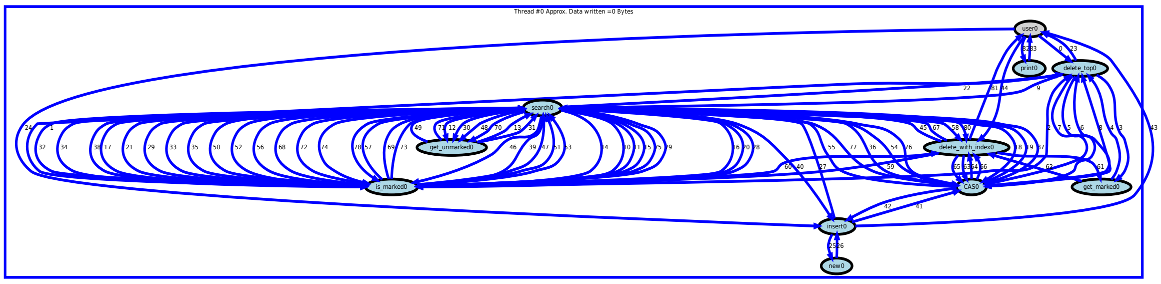


Figure 5.4: A thread in Non-blocking linkedlist

Static dependencies	0
Dynamic dependencies	2
Maximum load and store	140
Dependency inside loop	0
Dependency inside branch	0
Warning abnormal termination	No
Warning recursive	No
Warning memory management system calls	Yes

Table 5.8: Results for Non-blocking linked list

5.2 Application in TMFv2 with existing scheduler

In TMFv2, when a transaction completes the execution it validates all speculative writes on the shared memory but not with other transactions which are being executed in parallel. Consider two transactions A and B working on same dataset, for example. If transaction A has committed all its speculative writes to the memory, transaction B which was being executed on the same data will get aborted when it validates. Transaction B has to abort but the time frame from the time when transaction A committed to the time transaction B validates was wasted. Therefore, the performance degradation depends on the difference between the size of two transactions. However, the CiT tool can help resolve this issue for both for existing scheduler.

Using the information given by CiT about conflicts can help the scheduler to schedule conflicting transactions on the same processor. Consequently, it does not only avoid the overhead of abort and restart, but also the execution time difference between the two transactions. The programmer can also combine these two transactions together which can even, further reduce the validation overhead. Eventually, reduction in the overhead of abort, restart and validation will cause decrement in the traffic on the network.

The custom made test application (Appendix C.2) was tested on the TMFv2 simulator. At first, both transaction were scheduled and executed in parallel on different processors. Upon completion of execution, task 2 validated its read set and detected a conflict with task 1 that had already committed. This caused for task 2 to abort and restart the execution. In the second case, the tasks were scheduled one after the other, so that there are no conflicts between the two tasks. This mode of execution resulted in a 20% performance improvement in comparison to the previous one as Figure 5.5 is depicting.

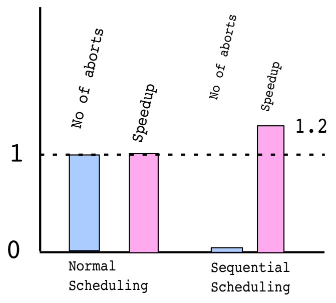


Figure 5.5: Improvement in performance with sequential scheduling

6 Conclusion

This chapter summarizes the goal, motivation and methodology of the CiT tool. Further, intuitive areas related to the CiT tool which can be explored in the future are described.

6.1 Summary

In hardware transactional memory architectures like TMFab and TMFv2, all conflicts due to shared data between transactions are resolved at run-time , thereby, the programmer is relieved from the synchronization of the shared data between transactions. However, these architectures lose performance in the process of finding and resolving conflicts. Although, as it is mentioned in [23], if conflicts are known before execution then these architectures can have improvement in terms of performance. This was the primary goal of the thesis, to find out all possible data dependencies between transactions at compile time.

In this thesis, the Compiler insights to Transaction memory (CiT) tool is presented which statically analyzes transactions and predicts potential data dependencies between them. Further, it gives the feedback report which helps the programmer to write an optimized code for transaction. In order to develop the tool, the plugin feature of GCC is used. The CiT analyzes each function individually to extract information about variables and creates a variable database. Using the database, all variables are connected to each other according to dependencies, thereby, making a data flow. Further, it records all calls (with arguments) made by a particular function. Subsequently, call graph for each transaction is developed when entry point/function is known. Using call graph with arguments information, data flow between functions is analyzed. Consequently, all addresses in a transaction which can be potential data dependencies are collected and compared with other transactions to get the final result. Further, during analysis information about abnormal termination, recursion, memory management system calls, maximum load and store instructions and loops as feedback report is delivered.

To validate the CiT tool, several test applications and some applications from the STAMP benchmark suite, in which data dependencies are already known, are compiled on it. Experimental results stands on expectations as CiT tool detects all the static data dependencies and predict dynamic dependencies with the correct feedback report.

The output of the CiT tool can be used as an input to the hardware scheduler in TMFab and TMFv2. Since, existing scheduler doesn't support this, it can be modified in order to avoid overhead of validation, abort, restart and hazard detection in these architectures.

6.2 Future Work

This section describes following suggestions for future work based on this thesis, in terms of improvement of the performance on TMFab and TMFv2 architectures and enhancing the functionality of the CiT tool:

1. The simple scheduler of both architecture TMFab and TMFv2 can be modified to accept data dependencies information from the CiT tool. Subsequently, the scheduler can take necessary actions to avoid conflicts, overhead of abort, validate, restart and hazards.

2. To provide a processor independent software interface in the form of a protocol to send the output of the CiT tool to the above mentioned new hardware scheduler.

3. CiT tool only supports language C. As object oriented programming is becoming prominent, it can be extended to support C++ or JAVA language.

4. Using control flow and data flow, independent sections of the code inside a function can be recognized, thereby, exposing a opportunity to divide a transaction into several transactions.

5. This plugin can be extended to search data dependencies in a sequential program [26], subsequently, dividing a sequential program into parallel sections. This extension can convert the CiT into tool-chain which can programmer completely discharged from any difficulty in parallel programming.

Porting A

This chapter describes one of the problems in porting an application on TMFab and TMFv2 architecture which is followed by the solution.

A.1 Stack Conflicts

In a program, there is separate memory section for the stack which is used to keep the local variables of a function. When a function is invoked, each local variable and parameter is pushed to the stack. On the return of the function call, all local variables are pop out of the stack to make it free. In the case, when a function invokes another function, a contiguous memory space in the stack is allocated for the callee. Therefore, stack space for callee grows over the caller's one as it is shown in Figure A.1.

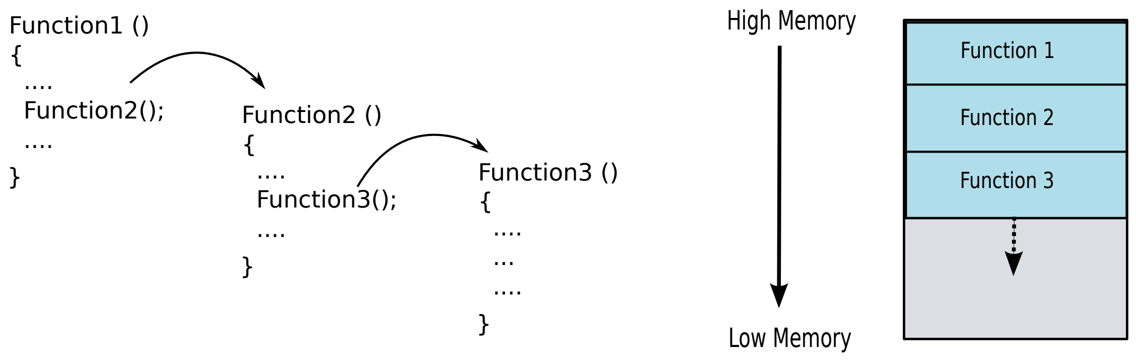


Figure A.1: Stack Memory usage

There is no compatible library or compiler for parallel transactions for TMFab and TMFv2. In GCC, which is a sequential compiler, all transactions are considered as sequential code. Subsequently, when transactions are invoked their stack grows over the caller's stack. But in TMFab and TMFv2, transactions are scheduled on different processors and are executed in parallel. Consequently, stack space allocated for both transactions are on the same memory space as it is shown in Figure A.2. Since both transactions are accessing same stack memory, when one of them validate all speculative writes (which also includes local variables residing in the stack), it results in conflict. As the stack is local to every function, this conflict is unnecessary.

A.2 Solution

The solution conceived for this aforementioned problem is to have separate stack space for each transaction. This can be done in two ways.

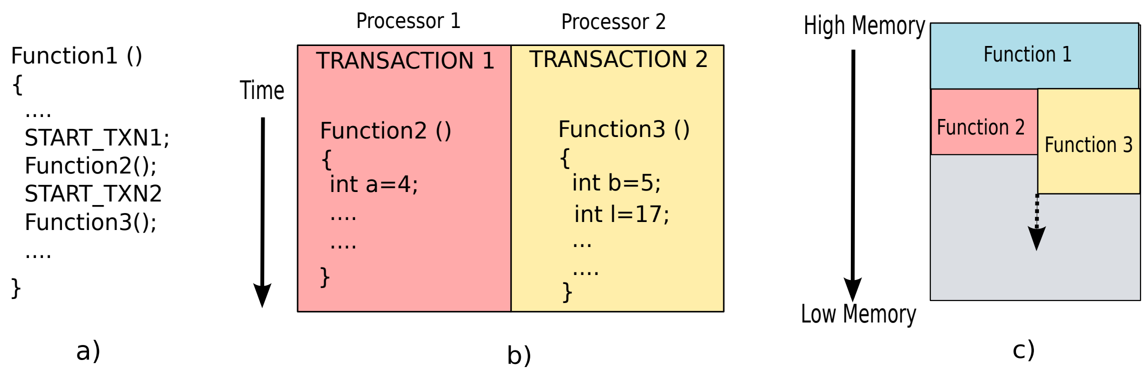


Figure A.2: Stack space contention between txn

A.2.1 Linker Script

Apart from the stack space for the program, a separate stack space is allocated especially for transactions by modifying the linker script. Further, the newly created section of the memory can be divided into number of transactions. Each transaction has its own stack space. Therefore, all functions, inside the transaction, stack can grow in the allocated memory space as it is shown in Figure A.3. However, it is required to mention the amount of memory for each transaction. Also, the stack pointer at start has to be reset according the address of the stack memory allocated for the transaction. Therefore, there is a need of a tool/script which can automate it. However, there is another solution which doesn't require any script and discussed in the next section.

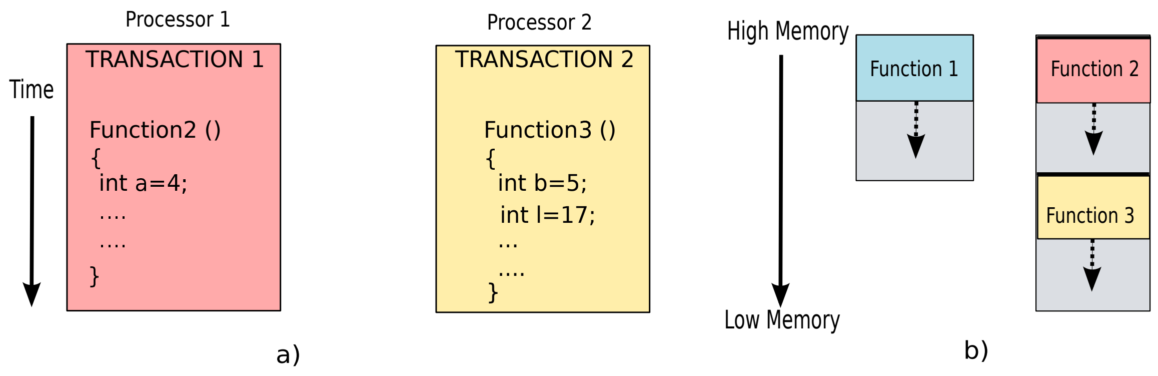


Figure A.3: Stack solution

A.2.2 GCC - Plugin (Proposal)

A Plugin to GCC can be used to reset the stack pointer at the start of the transaction. This can be done by developing two passes, one GIMPLE and another RTL. At the GIMPLE pass, marker indicating the start of a transaction can be perceived. This information can be used at RTL pass to reset the stack pointer at the appropriate location.

B Data Structures

Variable	
function	Name of the function where variable defined
type	To identify variable is record or array
recordname	Record name
name	Name
vno	Number of variable
isglobal	For global variable
funcno	Number of the function where variable defined
write	If variable is assigned
line	Line number
read	variable is read
inloop	variable in loop
istemp	variable is temporary
ispara	parameter
referas	Indirect refernece or with Address operator
bytes	Size of variable
valueis	Pointer, Integer or Real
value	Value of the constant
value2	value of the offset in array
bbno	basicblock number
loopno	loopno
cond1	in case of conditional dependency
cond2	in case of conditional dependency
var1	Pointer to right hand side operand
var2	Pointer to 2nd right hand side operand
adr	Pointer to the pointer variable or variable (with address expression)
paradr	Pointer to point parameter
arrsize	for pointing variables for offset in the array

Table B.1: Data structure of variable

Function	
byteamount	Amount of byte written in the function
varno_s	Starting variable of the function
varno_f	Last variable of the function
paras[]	Parameters of the function
variable *arg[][]	Pointer to all arguments for each callee
block[]	All basicblocks in the function
function *fnptr[]	Pointer to all callee functions

Table B.2: Data structure of Function

Declarations	
FIELD_DECL	field of the structure
VAR_DECL	variable
PARM_DECL	Paramter
INDIRECT_REF	Indirect reference
ARRAY_REF	Array
INTEGER_CST	Integer number
REAL_CST	real number
COMPONENT_REF	Structure variable
STRING_CST	Constant string
ADDR_EXP FUNCTION_DECL	address expression (variable with unary operator &) function decalaration

Table B.3: Declarations

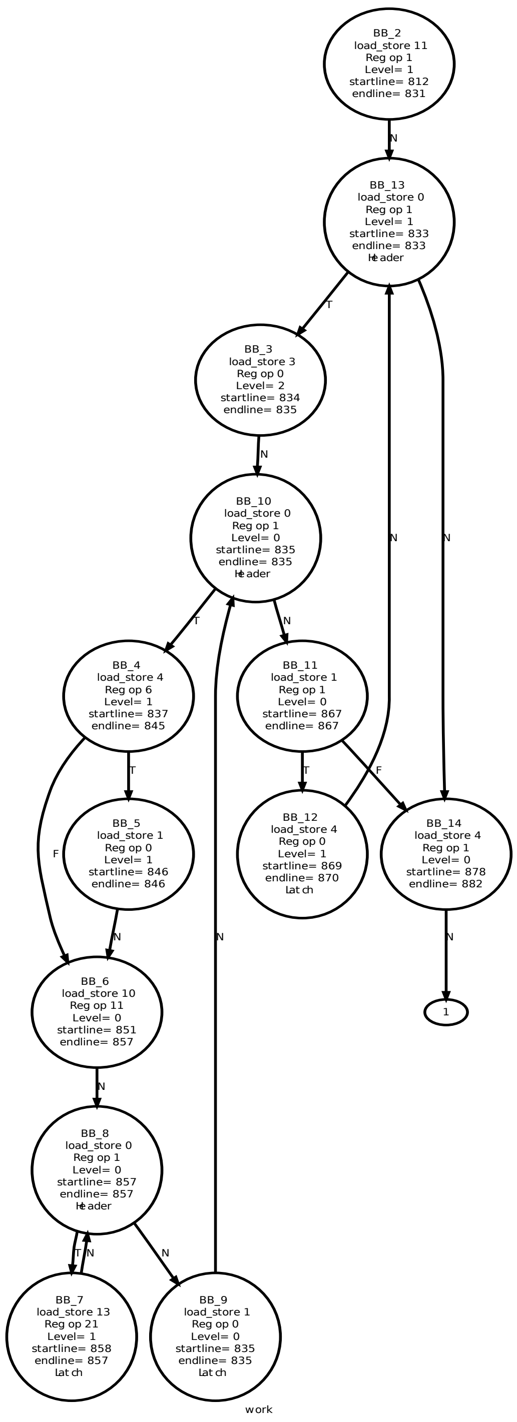

BB_2 load_store 11 Reg op 1 Level= 1 startline= 812 endline= 831
N
BB_13 load_store 0 Reg op 1 Level= 1 startline= 833 endline= 833 Header
T
BB_3 load_store 3 Reg op 0 Level= 2 startline= 834 endline= 835
N
BB_10 load_store 0 Reg op 1 Level= 0 startline= 835 endline= 835 Header
N
N
T
N
BB_4 load_store 4 Reg op 6 Level= 1 startline= 837 endline= 845
BB_11 load_store 1 Reg op 1 Level= 0 startline= 867 endline= 867
T
T
F
F
BB_5 load_store 1 Reg op 0 Level= 1 startline= 846 endline= 846
BB_12 load_store 4 Reg op 0 Level= 1 startline= 869 endline= 870 Latch
BB_14 load_store 4 Reg op 1 Level= 0 startline= 878 endline= 882
N
N
N
1
BB_6 load_store 10 Reg op 11 Level= 0 startline= 851 endline= 857
N
BB_8 load_store 0 Reg op 1 Level= 0 startline= 857 endline= 857 Header
T
N
N
BB_7 load_store 13 Reg op 21 Level= 1 startline= 858 endline= 857 Latch
BB_9 load_store 1 Reg op 0 Level= 0 startline= 835 endline= 835 Latch
work

Figure B.1: Control flow graph for function work in Kmeans

C Test Applications

C.1 Code Test1

```c
#include <stdio.h>
#include<pthread.h>
#include<malloc.h>

int A[100],B[100],C[100];
struct queue {
        int key;
        struct queue *next;
};

struct queue *head, *tail;

struct str {
   int field1;
   int field2;
};

void callee1 (int *p)   {

    int d = 12;
  *p = d;

}


void * task3(void *b) {

    struct str *t =(struct str *) b;

    C[0] = t->field1;

}

void * task1(void *c )
{
  int i,j;
    int *p = (int*) c;
  callee1(p);
  for ( i; i<100 ; i++)
        A[i] = B[i];

}
```

```
void * task2(void *a)
{ int i,j;

  int *p = (int*) a;
  for ( j; j<100 ; j++)
        *p = *p + C[i];
    callee1(p);

}

void * task4(void *b)
{
  struct str *t =(struct str *) b;

        t->field1 = A[0];

}

void * task5 (void *par)
{
    struct queue *new = (struct queue *) malloc(sizeof(struct queue));
    assert(new);
    struct queue *temp=head;
    while (temp->next!=tail)
        temp=temp->next;

    temp->next= new;
    new->key= * ((int *) par);
    new->next= tail;

}

void * task6 (void *par)
{
    struct queue *temp=head;
    while (temp->key!= * ((int *) par))
        temp=temp->next;

    if(temp!=tail)
     temp->key=0;

}




int main()
{
    int *p,*p2,a=2,c=5;
    struct str rec, *prec;

    rec.field1= 4;
  rec.field2= 5;

    struct queue *q ;
    q= (struct queue *) malloc(sizeof(struct queue));
    head=tail=q;
    prec = &rec;
    pthread_t   threads[4];
    p=&a;
    p2=&c;
    pthread_create(&threads[0], NULL, task1, (void *)(p));
    pthread_create(&threads[1], NULL, task2, (void *)(p));
    pthread_create(&threads[2], NULL, task3, (void *)(prec));
    pthread_create(&threads[3], NULL, task4, (void *)(prec));
    pthread_create(&threads[4], NULL, task5, (void *)(p));
    pthread_create(&threads[5], NULL, task6, (void *)(p2));
}
```

C.2 Code Test2

```
#include <stdio.h>
#include<pthread.h>
#include "ctm.h"

int A[100],B[100],C[100];

void task1(int *c )
{
 int i,j;
    int *p = c;

  for ( i=0; i<100 ; i++)
        A[i] = *p;

  for (j=0; j<100 ; j++)
        B[j] = *p;

  for (j=0; j<100 ; j++)
        C[j] = *p;

}

void  task2(int *a)
{ int i,j;
  int *p = a;
  for ( j=0; j<100 ; j++)
        *p = *p + 1 +C[j]+A[j];

}

int main()
{
    int *p, a=2;
    p=&a;

    START_TXN("0","1","512");
    task1(p);
    END_TXN("0","1");

    START_TXN("1","1","1024");
    task2(p);
    END_TXN("1","1");

    return 0;
}
```

Bibliography

[1] Gcc plugin.

[2] *QUAD: a memory access pattern analyser*, ARC'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] R. Simoni A. Agarwal, J.L. Henessy and M. A. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, 1988.

[4] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66 –76, dec 1996.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, March 1976.

[7] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 29–41, New York, NY, USA, 1979. ACM.

[8] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] K.-F. Faxen, K. Popov, L. Albertsson, and S. Janson. Embla - data dependence profiling for parallel programming. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 780 –785, march 2008.

[10] W.W.L. Fung, I. Singh, A. Brownsword, and T. Aamodt. Kilo tm: Hardware transactional memory for gpu architectures. *Micro, IEEE*, 32(3):7 –16, may-june 2012.

[11] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

[12] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, Honggo Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 102 – 113, june 2004.

[13] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.

[14] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[16] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[17] Intel. *Intel C++ STM Compiler Prototype Edition 3.0*. Intel, 2008.

[18] S. S. Kumar. Tmfab: A transactional memory fabric for chip multiprocessors. Master's thesis, 2010.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[20] S. Mannarswamy and R. Govindarajan. Handling conflicts with compiler's help in software transactional memory systems. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 482 –491, sept. 2010.

[21] Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Transactional memory: The hardware-software interface. *IEEE Micro*, 27(1):67–76, January 2007.

[22] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *Proc. of the GCC Developers Summit*, Ottawa, Canada, May 2003.

[23] Anastasios Michos. A novel concurrent validation scheme for hardware transactional memory. Master's thesis, 2012.

[24] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35 –46, sept. 2008.

[25] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[26] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Trans. Parallel Distrib. Syst.*, 7(11):1121–1132, November 1996.

[27] K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *Parallel and Distributed Systems, IEEE Transactions on*, 15(3):196 – 213, march 2004.

[28] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[29] Richard Stallman. Generic. In *GCC Internals*. http://gcc.gnu.org/onlinedocs/gccint/GENERIC.html.

[30] Richard Stallman. Gimple. In *GCC Internals*. http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.