

Optimization of a Data Race Detector

Bastiaan Stougie

October 2003

Optimization of a Data Race Detector

THESIS

to obtain the title of

Master of Science in Technical Informatics

at Delft University of Technology,

Department of Electrical Engineering, Mathematics and Computer Science,

Parallel and Distributed Systems group.

Work performed for

Ghent University

Department of Electronics and Information Systems

Parallel Information Systems group

by

Bastiaan Stougie

October 2003

Graduation Data

Author: Bastiaan Stougie

Title: Optimization of a Data Race Detector

Graduation committee: Prof. dr. ir. Koen De Bosschere Ghent University

Dr. ir. M. Ronsse Ghent University

Prof. dr. ir. H.J. Sips (voorzitter) Delft University of Technology

Dr. ir. D.H.J. Epema Delft University of Technology

Dr. ir. B.H.H. Juurlink Delft University of Technology

Graduation date: November 10, 2003

Abstract:

A data race condition occurs when multiple processes of a parallel program with a shared memory space access the same memory location in an unpredictable order, and at least one write access is involved. This makes the final value stored at the memory location and/or the value read by each process unpredictable. Sometimes a data race is intentional or harmless, but often it is a serious problem caused by a programming error.

We have improved an application for data race detection. This application analyzes the data accesses of a multi-threaded or parallel program by inserting, at run time, instrumentation code in front of each instruction of the program that accesses memory, and by intercepting all synchronization operations. This slows down execution considerably. Our task was to optimize this application as much as possible to speed up the execution.

We started with a data race detector that could only examine a very short execution span of a multi-threaded program before the detector used up all memory and had to give up. Even for such a short execution span this examination took a very long time. We have solved the memory usage problem by combining the registered memory accesses in the data race detector to free up memory, without loss of detection accuracy, and by optimizing almost all algorithms in the data race detector significantly, especially the memory allocation algorithms and the multi-level bitmap algorithms. This resulted in a significant speedup.

In addition we have optimized, eliminated and combined the instrumentation code that is inserted before each instruction where possible, without sacrificing the accuracy or correctness of the data race detection. The time the data race detector invests optimizing the instrumented code is earned back during the execution of the instrumented code for almost all benchmark programs we have tested.

Although data race detection on such a low level remains time- and memory-consuming, we have achieved a significant improvement of the performance.

Preface

This is my MSc Thesis for the study Technical Informatics at Delft University of Technology. I performed the task described in this report for the Parallel Information Systems (PARIS) group of the Department of Electronics and Information Systems (ELIS) of Ghent University in Belgium.

I wish to thank all my advisors for the help they have given me: especially lic. Jonas Maebe for the pleasant cooperation while working on the same sources and his answers to my many questions about DIOTA, dr. ir. Michiel Ronsse and prof. dr. ir. Koen De Bosschere at Ghent University, and dr. ir. D.H.J. Epema at Delft University of Technology.

Bastiaan Stougie,

Ghent,

October 2003.

Table of Contents

1	Introduction.....	1
2	DIOTA and the data race detector.....	3
2.1	DIOTA.....	3
2.2	The data race detector.....	4
3	Optimization of the data race detector.....	7
3.1	Efficient administration of thread segments.....	7
3.1.1	Segment removal.....	7
3.1.2	Segment combination.....	8
3.1.3	A faster segment allocation mechanism.....	8
3.1.4	An ordered list of allocated segments for each thread.....	9
3.2	Efficient administration of vector clocks.....	9
3.2.1	Administration of vector clocks for objects.....	9
3.2.2	Dynamic size of vector clocks.....	10
3.3	Optimization of the bitmaps.....	10
3.3.1	Increasing the number of levels of indirection.....	11
3.3.2	A faster bitmap block allocation mechanism.....	12
3.3.3	Bitmap block caching to skip the many levels of indirection.....	12
3.3.4	Combined bitmaps for LOAD/STORE access.....	12
3.3.5	Making use of SIMD technology.....	13
4	Optimization of DIOTA for data race detection.....	15
4.1	Basic rules for elimination and combination.....	15
4.2	Opportunities for optimization.....	15
4.3	A new design for DIOTA's instrumentation module to allow optimization.....	16
4.4	The IA-32 instruction decode module.....	17
4.5	The instruction instrumentation module.....	18
4.6	Plain optimization of the registration code.....	18
5	The control flow analysis module.....	21
5.1	Correct optimization of memory access registrations.....	21
5.2	Alternative methods for correct optimization.....	22
5.2.1	Method 1: Writing double instrumented code.....	22
5.2.2	Method 2: Skipping subsequent registrations.....	22
5.2.3	Method 3: Control flow analysis.....	23
5.3	Generating a control flow graph.....	24
5.3.1	Basic method for building a graph.....	24
5.3.2	Invalid addresses and invalid instructions.....	25
5.3.3	Call instructions and system call instructions.....	26
5.3.4	Recovering from incorrect identification of instruction blocks.....	27
5.4	Other benefits of control flow analysis.....	28
6	Optimization of memory access registrations.....	31
6.1	Intra-node optimization of indirectly addressed memory accesses.....	31
6.1.1	Keeping track of register relations.....	31
6.1.2	Keeping track of memory accesses for optimization.....	33
6.1.3	Optimization strategy.....	35
6.2	Inter-node optimization of indirectly addressed memory accesses.....	36
6.2.1	Determining the relations between registers at the begin of every node.....	37
6.2.2	Determining which accesses are guaranteed to precede and follow each node.....	38
6.2.3	Identifying candidates for elimination.....	39
6.3	Eliminating registrations of stack memory accesses.....	40
6.4	Loop unrolling.....	40
6.4.1	Identifying loops.....	40

6.4.2	Unrolling the first iteration of a loop.....	41
6.4.3	Loops containing call instructions.....	42
6.5	Why inlining is not practical to optimize data race detection.....	43
6.6	Building and using DIOTA and diota-dr with the optimizations.....	43
7	Evaluation.....	45
7.1	Benchmark setup.....	45
7.2	The cost of instrumentation.....	45
7.3	Segment combination settings.....	46
7.4	Bitmap optimizations.....	47
7.5	Memory access registration optimization by DIOTA.....	49
8	Conclusions and future work.....	51
8.1	Conclusions.....	51
8.2	Future work.....	52
8.2.1	Partial inlining of subroutines.....	52
8.2.2	Moving memory access registrations out of loops.....	52
8.2.3	Postponing memory access instrumentation.....	53
8.2.4	Limiting the number of context switches for instrumentation.....	53
8.2.5	Eliminating stack accesses dynamically.....	53
8.2.6	Segment allocation, comparison and combination strategies.....	54
8.2.7	Bitmap optimizations.....	54

1 Introduction

In this thesis, we describe the optimization of a data race detector. A data race is the term used for a situation where multiple processes of a parallel program with a shared memory space access the same memory location in an unpredictable order, and where at least one write operation that changes the value at the memory location is involved. This makes the final value that is stored at the memory location and/or the value read by each process unpredictable.

If not intended, such a situation is dangerous for the correctness of the program's operation and results. Therefore it is normally avoided by executing synchronization operations to determine explicitly in which order the processes may access the shared memory location. However, data races are frequently encountered as programming error, because multi-threaded and parallel programs are often very complex and because even the best programming practices cannot prevent all programming errors. An application that can detect data races in a running program is a valuable tool to uncover such errors.

DIOTA is a just-in-time instrumentation library for Linux on an Intel IA-32 architecture [MAE02]. It is a dynamically loaded library that overrides certain standard library functions to take control of the execution of an arbitrary program when that is started. After DIOTA has gained control, it can rewrite the machine code of the program during the execution of the program, as desired by an application built on top of DIOTA.

The data race detector that we must optimize is built on top of DIOTA. For the data race detector, DIOTA adds instrumentation code to all instructions of a multi-threaded application that access data. The instrumentation code reports all memory locations that are accessed by each thread to the data race detector, enabling the data race detector to determine if the same memory location is accessed by multiple threads without a synchronization operation in between the accesses. If so, and if at least one of the accesses is a write access, the application contains a possible data race and the data race detector will report it.

The problem under consideration in this report is that the extra instrumentation code slows down the execution of the program very much, and this makes the detection process very slow. We have implemented several methods to improve DIOTA and the data race detector to reduce the time needed to analyze programs.

Our goal is to optimize the detector to improve the performance in the longer run, for multi-threaded applications that may run for a long time, such as the internet browser Mozilla. This means that we can afford to put effort into the one-time optimization of code that will be executed many times. This gives us plenty of opportunities.

Before we can go into detail about our work, we will first explain the basic operation of DIOTA and the data race detector in Chapter 2. Chapter 3 describes the improvements to the data race detector itself. Although these improvements have increased performance considerably, the detection process remains slow, because programs perform a huge number of memory accesses.

Fortunately, many instructions access the same memory locations, often we can eliminate double registrations of an access of the same memory location. If adjacent memory locations are accessed by different instructions, the registrations of these accesses can sometimes be combined. In Chapter 4 we describe which conditions must be fulfilled to allow for elimination and combination, and we describe our new design for the instrumentation module of DIOTA to implement this. In Chapter 5 we introduce several alternative methods to realize elimination and combination. One of these methods requires control flow analysis, and this is explained in the rest of Chapter 5. Because the detection of opportunities for elimination and combination is complex, we have created a separate

Chapter 6 to cover our work on this subject.

In Chapter 7, we evaluate the achievements of our work by comparing the results of our work with the results before we started, and we draw our conclusions and present suggestions for further work in Chapter 8.

2 DIOTA and the data race detector

Before we go into detail how we have optimized the data race detector, we will first briefly explain how the data race detector [RON99] works. The detector is an application based on the DIOTA [MAE02] library.

2.1 DIOTA

DIOTA is an acronym for Dynamic Instrumentation, Optimization and Transformation of Applications. It is a library for instrumentation of program binaries, developed by dr. ir. Michiel Ronsse and lic. Jonas Maebe at the Parallel Information Systems group of the department of Electronics and Information Systems of Ghent University.

DIOTA is able to intercept the start of the execution of any binary program on a Linux operating system on an IA-32 architecture, and it can then alter the program at the machine code level. DIOTA has been designed to do the dirty work for other libraries that wish to analyze a running program. These backend libraries (*'backends'*) can make DIOTA call their own functions at prescribed points in the running program.

The DIOTA library takes control before the program to be analyzed starts, decodes the instructions at the start of the program, and writes a modified version with instrumented instructions of the initial sequence of instructions in the program in a buffer. After the instrumented instructions have been executed, control is returned to DIOTA. DIOTA then writes an instrumented version of the next instruction sequence, and so on. Only the instrumented instructions are in a different location. DIOTA does not move or copy the program's data, these reside at the original locations in memory. The data on the stack are accurate too, including the original return addresses.

Figure 2.1 depicts the basic working of DIOTA when the data race backend is used.

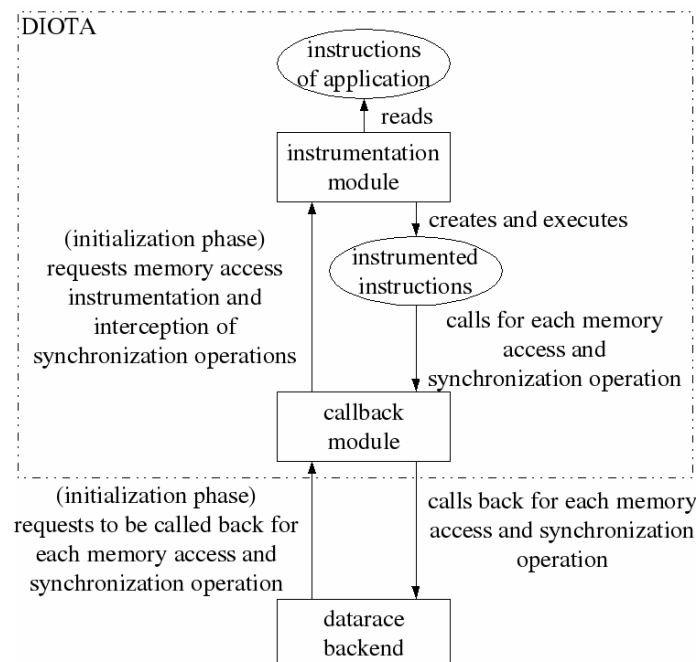


Figure 2.1: Schematic of DIOTA and the data race backend

Take for example the short Code fragment 2.1 of the application to be instrumented.

<u>address</u>	<u>instruction</u>	<u>comment</u>
11:	mov 0x8048400,%eax	; load a value from memory into register %eax
12:	add \$13,%eax	; add 13 to the value in register %eax
13:	mov %eax,0x8048410	; store the result at an other memory location

Code fragment 2.1: Example code fragment to be instrumented

This short code sequence loads a 32-bit value from hexadecimal memory location 8048400 into register %eax, adds the value 13 to it, and stores the result at hexadecimal memory location 8048410. If a backend instructs DIOTA to inform a function in the backend of every memory access that the program performs, DIOTA will insert a call to that function in the instrumented code. For example, it writes a modified version as shown in Example 2.2 for the code of Code fragment 2.1.

<u>address</u>	<u>instruction</u>	<u>comment</u>
2000:	push \$11	; address of instruction that accessed memory
2001:	push \$0x8048400	; address of accessed memory
2002:	push \$4	; 4 bytes of memory accessed
2003:	push \$1	; 1 = access type 'LOAD'
2004:	call diota_trace_	
2005:	mov 0x8048400,%eax	; original instruction
2006:	add \$13,%eax	; original instruction (does not access memory)
2007:	push \$13	; address of instruction that accessed memory
2008:	push \$0x8048410	; address of accessed memory
2009:	push \$4	; 4 bytes of memory accessed
2010:	push \$2	; 2 = access type 'STORE'
2011:	call diota_trace_	
2012:	mov %eax,0x8048410	; original instruction

Example 2.2: Instrumented version of Code fragment 2.1. Instrumentation code is in bold.

In this modified version, DIOTA has inserted a call to the function `diota_trace_` in front of each instruction that accesses memory. `diota_trace_` receives 4 arguments, that are passed to it via the stack (the `push` operations place them onto the stack). With these 4 arguments, `diota_trace_` will call the callback module, which will in turn call all functions of all backend libraries that have requested to be informed of memory accesses.

The instrumentation of instructions is very time-consuming. Therefore, sequences of instrumented instructions are kept in a cache for future use. In addition, a “profile” module has been written to analyze which code paths are executed most often. An “inline” module can then write optimized instrumented instruction sequences for these paths. The inline module can speed up the execution in some cases, but unfortunately in other cases it only slows down the execution further, and it consumes a lot of extra memory.

2.2 The data race detector

Below follows a very basic description of how the data race detection backend works. For more detail, see the article [RON99] by dr. ir. Michiel Ronsse, the author of the data race detector. The data race backend has the following major functions.

1. Registration of all memory accesses for each thread segment

As we have seen in the previous section, the data race detector is a backend library for DIOTA. When the program in which data races must be detected starts, the data race library instructs the DIOTA library to insert a call to a specific function for each instruction of the program that accesses memory. DIOTA will pass several arguments to this function:

- the address of the machine code instruction that is responsible for the memory access,
- the accessed memory region (begin-address and length),
- the type of the access (load / store / modify).

The function in the data race backend uses these arguments to record all memory accesses for every segment, which is the execution period of a thread between two consecutive synchronization operations such as locking a mutex, of every thread.

2. Identification of mutually unordered segments of different threads

Segments of different threads that are not ordered by synchronization operations can have executed in parallel and therefore data races can have occurred between these segments. The backend detects the ordering of the segments by associating a vector clock with each segment and with all synchronization objects such as mutexes and semaphores. A vector clock is a vector with one segment number for each thread as elements. A segment number for thread T in a vector clock V associated with an object O, $V_O[T]$, means that thread T has synchronized most recently with the object O at the end of the execution of its segment $V_O[T]$.

In Linux, multi-threaded applications can be written by making use of the 'pthread' library [LER02]. This library provides functions to start and terminate threads and provides several synchronization operations that a multi-threaded program can use to avoid data races (e.g., `pthread_mutex_lock`, `sem_post`). The data race detector instructs DIOTA to replace calls to these functions with a call to functions of the data race detector itself. Thus the data race backend can detect transitions between segments, and update the vector clocks of the mutexes, semaphores and the thread segment that are involved in the synchronization operations.

3. Detection of data races between unordered segments

Data races can be identified by comparing the memory accesses of a segment at the moment that that segment is completed with the memory accesses of segments of other threads that are not ordered with respect to the completed segment.

The detected data races are written to a file on disk (thread numbers, segment numbers, and memory address of the data race). A 'replay' (record-replay) backend that can record the thread synchronization operations and replay them during a subsequent executions of the same program, allows for a second identical execution of the application¹. During the second execution, the data race detector can obtain more information about the data races that were detected during the first run: the complete call stacks of the offending instructions can be dumped (Example 2.3). Of course the application programmer can re-execute the application identically as often as desired to find the exact cause of the data race.

¹Using the replay backend, a subsequent execution will only be identical if the input of the program (including the results of system calls) is also identical. An ingenious 'input-replay' backend that will record the entire execution of the application by registering all results of all system calls, and that will be able to 'replay' these during subsequent executions of the application, is being developed by Frank Cornelis at the ELIS department of Ghent University. This will allow for truly identical re-executions of the program, and will also provide functionality that allows the data race detector to register and inspect the memory accesses that are performed by the Linux kernel during system calls.

```

-----
Offending instruction of thread 2:
0x080485f4: mov    0x80497c4,%eax                a1 c4 97 04 08

/user/bastiaan/diota/test/ploop.c+20: <doit>
19:      local2 = global2;
20:      local = global; // data race
21:      global++; // datarace

Call stack of thread 2:
no source code or symbol information found for address 0x402d5438
/user/bastiaan/diota/SO/libdiota.so:diota_pthread_start+266 [0x401b4332]
no source code or symbol information found for address 0x4001ce66
/user/bastiaan/diota/SO/libdiota.so:diota_pthread_start+266 [0x401b4332]
no source code or symbol information found for address 0x401b44db
no source code or symbol information found for address 0x4004ec3a
/lib/i686/libc.so.6:__clone+53 [0x4014ab25]
-----
Offending instruction of thread 1:
0x080485fc: incl  0x80497c4                ff 05 c4 97 04 08

/user/bastiaan/diota/test/ploop.c+21: <doit>
20:      local = global;
21:      global++; // data race
22:      local3 = global3;

Call stack of thread 1:
no source code or symbol information found for address 0x402d520c
/user/bastiaan/diota/SO/libdiota.so:diota_pthread_start+266 [0x401b4332]
no source code or symbol information found for address 0x4001ce66
/user/bastiaan/diota/SO/libdiota.so:diota_pthread_start+266 [0x401b4332]
no source code or symbol information found for address 0x401b44db
no source code or symbol information found for address 0x4004ec3a
/lib/i686/libc.so.6:__clone+53 [0x4014ab25]
-----

```

Example 2.3: An example of the output when the data race detector is run for the second time to obtain more information about the dataraces that were detected during the first run.

3 Optimization of the data race detector

In this chapter, we present the methods we used to increase the performance of the data race detector. These optimizations involve the modification the data race detector backend itself. Other optimizations that were achieved by changing DIOTA are presented in the next chapter. The data race backend as it was at the start of this assignment had several shortcomings:

- Inefficient segment allocation and administration algorithms.
- Use of an unlimited amount of memory, because segments were only added, never removed. Needless to say it could only run for a very limited period before it ran out of memory.
- Inefficient administration of vector clocks for synchronization objects such as mutexes and semaphores.
- Relatively inefficient data structures for memory access registration.
- Support for only the first 7 threads that were created.

In addition, the data race detector contained several bugs, it even contained a data race condition itself. In the subsections, we describe how we solved the above problems and optimized several parts of the code.

3.1 Efficient administration of thread segments

Every synchronization operation starts a new thread segment. A segment object (referred to hereafter as 'segment') is associated with each thread segment. It basically contains a vector clock instance and a bitmap in which all memory locations that are accessed during the execution of the thread segment are registered.

At the begin of this project, the data race detector used an array with a fixed number of `MAX_SEGMENTS` (defined as 20,000) segment objects. Allocation of segments occurred in $O(\text{number of recorded segments})$, by searching from index 0 to index $(\text{MAX_SEGMENTS} - 1)$ for the first free segment in the pool. Deallocation was implemented in $O(1)$ by simply setting the 'inuse' flag in the segment to 0, but did not occur in practice because removing of segments was not implemented correctly. Not only was allocation very inefficient, but because segments were never removed, even for simple example programs the number of available segments was too small or the data race detector ran out of memory after a few seconds of execution time. Increasing `MAX_SEGMENTS` does not help to solve this problem.

We solved these problems with the solutions presented in the subsections.

3.1.1 Segment removal

Without segment removal, the number of synchronization operations the data race detector can process is very limited, and therefore the range of applications it can examine is very limited. Segments may be removed if and only if their vector clock value precedes that of all current segments of all threads. Note that the vector clocks associated with synchronization objects are not relevant to determine which segments can be removed, because every time they are used for synchronization by a thread, they are updated by taking the element-wise maximum of its own elements and the elements of the vector clock of the current segment of the thread.

However, often one or more threads of an application perform no synchronization operations for a long time (e.g., a main thread that waits for its child threads to finish their tasks), and their current segment remains unordered with respect to almost all segments of all other threads. Therefore in

practice only very few segments can be removed and implementing this did not improve much in practice.

3.1.2 Segment combination

Fortunately, there is another way to eliminate segments: by combining segments of the same thread of which the vector clocks all have the same ordering with respect to the vector clocks of all other threads and with respect to the vector clocks of all synchronization objects. Implementing this was not a simple matter, but was the key to success: the backend now needs extremely much fewer segments. Even though segment combination will cost performance, it also earns back some execution time by reducing the number of comparisons of different segments. Segment removal is a logical extension of segment combination.

Segment combination in no way harms the exactness of the data race detection. Even though you no longer know the exact segment that caused a data race, you still know the range of segments that caused the data race, and this is sufficient information to be able to pinpoint the exact location of the data race during the re-execution.

Segments contain multi-level bitmaps that are combined when the segments are combined (see Section 3.3). The combination of these bitmaps frees memory that can be re-used for bitmaps of other segments. This is also a very important achievement, because without it we would still run out of memory almost immediately, even though we re-used the segment structures themselves.

3.1.3 A faster segment allocation mechanism

Segment removal and combination can degrade performance, but are necessary to overcome the memory requirement problems. However, the improvement in this section increases performance. Segment allocation and deallocation can be implemented with a constant time algorithm instead of a linear time algorithm in the following way:

- allocate a pool of segments (e.g., 1000 segments), and put them in a linked list of free segments
- for allocation, remove the head of the list of free segments in constant time. If the list of free segments is empty, first try combining and removing segments. If that does not provide new free segments, allocate an extra pool of segments (there is a hard limit of 1000 segment pools, whereas only one or a few segment pools are needed in practice for our test-programs, and this hard limit can be eliminated if necessary in the future).
- for deallocation, prepend the segment to the list of free segments.

We have also tried a hybrid method:

- allocate a pool of segments (e.g., 1000 segments), and treat it as an array of free segments
- create an empty linked list of de-allocated segments.
- for allocation, first look in the list of de-allocated segments. If that is empty, look in the array of free segments. If that too is empty, try combining and removing segments. If that does not provide new de-allocated segments, allocate an extra pool of segments.
- for de-allocation, prepend the segment to the list of de-allocated segments

The hybrid method saves the initialization time required to put the segments into a linked list when a new pool is allocated (which can cause many cache misses). However, in practice the hybrid method is much slower than the first method. This is probably caused by branch mispredictions,

because the hybrid method executes more conditional statements during allocation.

3.1.4 An ordered list of allocated segments for each thread

At the completion of a segment, the segment needs to be compared with a number of the most recent segments of other threads to detect data races. Previously, all MAX_SEGMENTS segments were checked to see:

- if the segment belonged to an other thread
- if the segment was recent enough to be unordered with the completed segment (by looking at the vector clock).

If so, the segment was compared to the completed segment.

We improved this significantly by introducing an ordered list of segments for each thread. The segments in the lists are ordered by segment number, most recent first. All segments from the head of the list up to a certain point in the list must be compared to a segment that has just completed. This way, the segments that need to be compared to a segment that has just been completed can be identified very efficiently in a natural way. These ordered lists are ideal for segment combination, too, because only segments with consecutive segment numbers can be combined, and this corresponds with the ordering of the list. New segments can simply be added to the head of the list in constant time.

3.2 Efficient administration of vector clocks

To detect the ordering of segments of different threads, it does not suffice to associate a vector clock with every thread. Vector clocks must be associated with mutexes and semaphores as well. This was not done efficiently. Performance improvements are described in the subsections.

3.2.1 Administration of vector clocks for objects

Every time a synchronization operation is performed, the vector clock of the mutex or semaphore that is involved must be looked up. Since the address of the mutex or semaphore is passed to the synchronization operation, and the data race backend intercepts the arguments of the synchronization operations, the detector can use a table that maps these addresses onto vector clocks. This was implemented with a simple array, and a lookup took linear time by scanning the array from begin to end for the correct address. There was no mechanism to remove the vector clocks of objects.

We have replaced the table with a hash table with as key the address of the mutex or semaphore. We also introduced a mutex to protect this table, correcting an existing data race condition in the backend itself.

Analogous to the segment and bitmap memory block allocation, a pool of vector clocks is pre-allocated, and we have introduced a list of free vector clocks for fast allocation and de-allocation.

We implemented a mechanism to remove outdated vector clocks. For this purpose, we introduced an 'in use' counter for each vector clock, initialized to 0. Every time the data race detector looks up a vector clock for a synchronization operation, the 'in use' counter of the clock is incremented. After the synchronization operation has been performed, (and the vector clock has been used and possibly updated), the vector clock is released by the data race detector, and its 'in use' counter is decremented. Therefore, if the counter is 0, we can be sure the vector clock is not in use. If in addition the vector clock is outdated, we can remove it. Outdated vector clocks of objects are only looked for and cleaned up when the data race detector runs out of vector clocks.

3.2.2 Dynamic size of vector clocks

The data race detector used fixed size vector clocks, implemented as an array of 8 segment numbers, limiting the maximum number of threads to 8. Regardless of how many threads there really were, all 8 elements of the vector clock were updated.

Because we have implemented segment combination, the number of allocated segments has been reduced drastically. Each segment has a vector clock object inside. Now that there are fewer segments, and therefore more memory is available, we can make the clocks wider to support more concurrent threads.

In addition, we introduced a mechanism to map threads onto vector clock indices. One of the benefits is that the real width of the vector clock becomes known, and we can increase performance because it is no longer necessary to update the entire vector clock: we only need to update the elements associated with threads that really exist.

Note that when a thread exits, we cannot reclaim its index in the vector clock. This is because the last segment(s) of the exiting thread may not be ordered with any new segment of any other thread of the program, so we may need to keep checking for data races between these segments and all new segments of other threads forever. This problem is not for me to solve during this assignment.

3.3 Optimization of the bitmaps

The data race detector registers all memory accesses of a thread segment in two bitmaps inside a segment object. All LOAD accesses are registered in the first bitmap, and all STORE accesses are registered in the second bitmap (MODIFY accesses can be and are regarded as STORE accesses for data race detection). Each bit in the bitmap corresponds to a byte of memory.

A flat bitmap with one bit for each byte of memory is infeasible, this would claim all memory for only 8 thread segments. Therefore the bitmaps use indirection, taking advantage of the data locality principle. A 32-bit memory-address is divided into several parts: the 9 address bits 31-23 are used as index into a table of 512 pointers (level 3) (see Figure 3.1). These pointers each point to another table of 512 pointers (level 2), for which the 9 address bits 22-14 are used as index. The pointers in this table point to a memory region (level 1) of 16K bits, for which address bits 13-5 are used as index to a 32-bit doubleword, and finally address bits 4-0 are used as bit index inside that doubleword. A pointer table at level 3 and 2, and a bitmap-block at level 1 all take up a block of 2K bytes of memory. Equal block sizes have been chosen to avoid memory fragmentation problems.

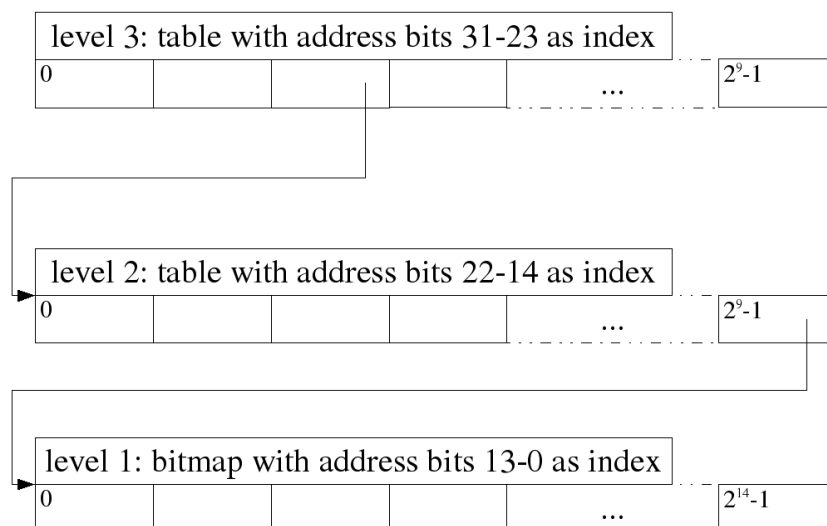


Figure 3.1: example of a multi-level bitmap

Whenever a byte of memory is accessed, the corresponding tables at level 3 and 2, and the level-1-bitmap-block are allocated if necessary, the correct pointers are filled in, and the corresponding bit in the level-1-bitmap-block is set to 1.

The optimizations to these bitmaps are described in the subsections.

3.3.1 Increasing the number of levels of indirection

We have examined how the performance is affected if we change the number of levels from 3 to 4, 5, or even more, and choose the optimal number of levels. Increasing the number of levels has the following advantages and disadvantages:

- it saves memory (at each level, the tables are smaller, and the bitmap at the final level is smaller)
- performance may increase, because less memory needs to be compared when comparing bitmaps
- performance may increase, because less memory needs to be OR'ed when combining bitmaps
- performance may increase, because less memory means fewer cache misses.
- performance may decrease, because every additional level of indirection implies extra sequential instructions to be executed.

For this reason, we implemented bitmaps with up to 9 levels (Table 3.1) and blocks of 32 bytes. To achieve blocks of 16 bytes, 15 levels would be needed.

<i>Number of levels of indirection</i>	<i>Address bit subdivision</i>	<i>Block size (in bytes)</i>	<i>Notes</i>
3 (original)	9,9,9,5	2048	
4	6,7,7,7,5	512	Level 1 uses only 256 bytes of the block.
5	3,6,6,6,6,5	256	Level 1 uses only 32 bytes of the block.
6	2,5,5,5,5,5,5	128	Level 1 uses only 16 bytes of the block.

<i>Number of levels of indirection</i>	<i>Address bit subdivision</i>	<i>Block size (in bytes)</i>	<i>Notes</i>
7	3,4,4,4,4,4,5	64	Level 1 uses only 32 bytes of the block.
9	3,3,3,3,3,3,3,5	32	

Table 3.1: Bitmap configurations for different numbers of levels of indirection

We compare the performance of the extremes, the 3-level bitmap and the 9-level bitmap, in Section 7.4.

3.3.2 A faster bitmap block allocation mechanism

Because the memory blocks used by the tables and by the bitmap at the deepest level all have the same size, we can improve the memory block allocation mechanism in the same way as we did for the segments: allocate pools of memory blocks, and use a linked list of free blocks. This saves calls to the slow memory allocation functions. In addition, we provide every thread with its own limited reserve of memory blocks that it can allocate without the delays of mutex locking.

3.3.3 Bitmap block caching to skip the many levels of indirection

Test results indicate that bitmaps with many levels give the best performance for applications of which the segments do not combine very well. However, every level of indirection is inherently sequential, so the CPU cannot parallelize this using ILP (Instruction Level Parallelism). To avoid these sequential operations, we cache the most recently modified blocks, using a LRU replacement policy. The cache code itself is highly optimized. Even though this software cache is sequential unlike a parallel hardware cache, performance is improved due to the high hit ratios (over 90%).

3.3.4 Combined bitmaps for LOAD/STORE access

Comparing two segments A and B for data race detection requires that you compare:

- the LOAD bitmap of segment A with the STORE bitmap of segment B,
- the STORE bitmap of segment A with the LOAD bitmap of segment B,
- the STORE bitmap of segment A with the STORE bitmap of segment B.

This means that you have to walk through two bitmaps three times. This can be improved by combining the separate bitmaps for LOAD and for STORE operations into one bitmap, by dividing the level-1 bitmaps into two equal-sized parts: the first half for LOAD access registration, the second half for STORE access registration. To compare segments, you then have to walk through the bitmap of segment A and the bitmap of segment B only once.

But we can even do better. Remember the definition of a data race: multiple threads access the same memory location, and at least one thread performs a STORE access to that memory location. If we use the first half of the lowest level for LOAD *and* STORE access registration, and the second half for STORE access registration, we can still detect all data races, we only have to compare twice instead of three times:

- the LOAD/STORE half of the level-1-bitmap-block of segment A with the STORE half of the level-1-bitmap-block of segment B,
- the STORE half of the level-1-bitmap-block of segment A with the LOAD/STORE half of the

level-1-bitmap-block of segment B.

This costs an extra level of indirection, because the distinction between LOAD and STORE costs an extra bit, so now we have 10 levels instead of 9. It also means that we have to register STORE accesses in both the first and the second half of the level-1-bitmap-block, but this is very cheap provided that you implement it without conditional branches, and it can be parallelized on the instruction level by the CPU.

We can implement this without conditional branches by doing something like the following:

Input: integer `access_type` has value 1 if STORE or MODIFY, and value 0 if LOAD.

Set the correct bits at index `level_1_index`. Also set the correct bits at index `level_1_index + access_type * (level_1_bitmap_block_size / 2)`.

This means that for a LOAD access, we set the same bits at the index `level_1_index` in the LOAD/STORE part of the level-1-bitmap-block twice, but for a STORE access we set the bits in both the LOAD/STORE part at `level_1_index` and the STORE part at `level_1_index + (level_1_bitmap_block_size / 2)`. The multiplication and division are cheap, because in practice we can replace the multiplication with a shift and the division is done by the compiler. If we implement it with conditional branches, e.g., `'if (access_type == LOAD) then ... else ...'`, we get a branch for which the direction is very hard to predict for the CPU, since it has a probability of around 50% to be taken or not. This immediately results in an enormous degradation of the performance. Of course we could also use two separate registration functions (one for STOREs and one for LOADs) to solve this problem, because we can hard-code these in the instrumented code.

The performance results are presented in Section 7.4.

3.3.5 Making use of SIMD technology

The bitmaps need to be compared (with 'AND' operations) and combined (with 'OR' operations). Today's Intel x86 processors [INT02] support special MMX and SIMD (Single Instruction, Multiple Data) SSE/SSE2 instructions that can perform 'OR' and 'AND' operations on 64-bit or 128-bit registers and on 64-bit or 128-bit memory regions, whereas traditional 'OR' and 'AND' are performed on 32-bit registers and 32-bit memory regions. In addition, there are instructions that allow for prefetching data from memory into the cache. Using these instructions may increase performance. Unfortunately, we have not been able to implement this successfully: the instrumented programs became unstable. The problem may be the saving and restoring the state of the shared space in the CPU for the MMX/XMM/floating point registers. If the instrumented program also uses one of these types of registers, conflicts may arise. There is not much documentation about this, so we cannot be sure.

4 Optimization of DIOTA for data race detection

DIOTA provides the data race detector with information about every memory access by the application during its execution. However, the data race detector does not need to be informed of repeated access of the same memory location during the same thread segment. Therefore, by optimizing DIOTA to prevent unnecessary registrations as much as possible, we can limit the performance loss caused by the instrumentation code.

We will describe the basic rules for optimization by eliminating and combining memory access registrations in Section 4.1. To show that these rules can be used in practice, Section 4.2 discusses some examples of adjacent memory accesses that occur frequently and can be optimized using the basic rules for optimization.

Section 4.3 presents a new design for DIOTA to allow for an analysis and optimization phase. Two modules of this design are discussed briefly: the instruction decode module Section 4.4 and the instruction instrumentation module in Section 4.5. The control flow analysis module is discussed in Chapter 5, and the memory access optimization module in Chapter 6.

We have also optimized the instrumentation code that passes the information about a memory access to the data race detector. This is explained in Section 4.6.

4.1 Basic rules for elimination and combination

Based on the following observations, several optimizations to improve performance are possible:

- For each thread segment (= execution period of a thread between two subsequent synchronization operations of that thread), the data race detector needs to register access to every different memory location only once, with type STORE if at least one of the accesses to the memory location is a STORE or MODIFY access, otherwise with type LOAD.
- Between synchronization operations, the data race detector is allowed to register accesses of adjacent memory locations with type STORE or MODIFY as a single STORE access of a larger memory region that includes all these adjacent memory locations.
- Between synchronization operations, the data race detector is allowed to register accesses of adjacent memory locations with type LOAD as a single LOAD access of a larger memory region that includes all these adjacent memory locations.

It is even allowed to register accesses of adjacent memory locations with type LOAD, STORE or MODIFY as a single LOAD access of a larger memory region that includes all these adjacent memory locations, as long as it registers the STORE and MODIFY accesses separately as STORE respectively MODIFY accesses as well.

Our top priority is to limit the number of registrations. The size of the registrations is our second priority, because more execution time can be saved by fewer registrations than extra time is needed to register an access of a slightly bigger memory region.

We have also considered buffering memory access registrations, to commit them just before the next synchronization operation.

4.2 Opportunities for optimization

A stack frame is an area of the stack used by a routine to store its arguments, return address, local variables, and storage to temporarily save and restore register values. The IA-32 has two registers that are normally used for stack operations: the ESP register, which is used as stack pointer, and the

EBP register, the stack-frame base pointer. Typically, a stack frame looks as depicted in Figure 4.1.

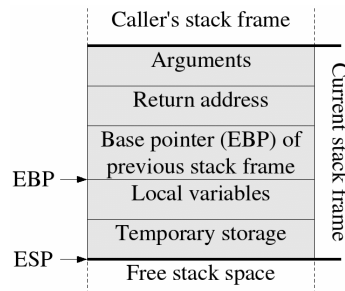


Figure 4.1: typical stack frame

To call a subroutine, the caller first pushes the arguments (if any) on the stack, then the call instruction pushes the return address on the stack. These push and call instructions access adjacent memory locations on the stack and can therefore be registered together. The called function first pushes EBP, then often initializes its local variables and sometimes uses temporary storage (if any). This often results in another series of accesses to adjacent memory locations that can be registered together. The initialization of a local variable with a STORE access makes the registrations for all subsequent accesses to that local variable redundant up until the next synchronization operation.

Other opportunities for optimization come from arrays (array elements are adjacent), and from data structures (their internal data fields are often adjacent). These arrays and structures may also reside in the stack frame.

So there is a lot of data locality that can be exploited to optimize memory access registrations.

4.3 A new design for DIOTA's instrumentation module to allow optimization

We have modified DIOTA thoroughly for a clear distinction between decoding instructions and writing instrumented code for these instructions, to allow for an analysis and optimization phase in between the decode phase and instrumentation phase.

Previously, the instrumentation module read an instruction, and instrumented it immediately. In our new design, it will first decode groups of instructions. Then it will analyze the opportunities to optimize memory access registrations performed by the instructions in the groups, taking the control flow into account. Then it will write the instrumented instructions and execute them. To this end, we have created the new modules depicted in Figure 4.2.

- the 'instruction decode module' decodes any IA-32 instruction (Section 4.4).
- the 'control flow analysis module' decides which instructions to group for optimization (Chapter 5), resulting in a control flow graph.
- the 'memory access registration module' performs analysis and optimizations on the grouped instructions (Chapter 6).
- the 'instruction instrumentation module' instruments the grouped instructions using the optimization information (Section 4.5).

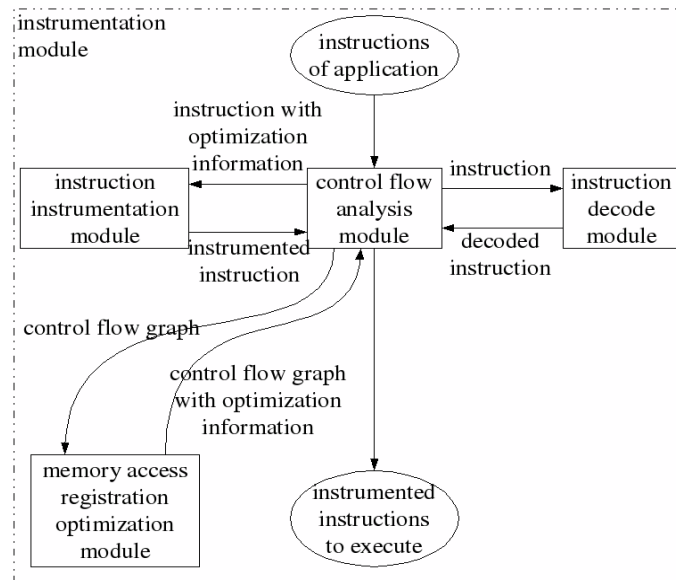


Figure 4.2: The new design for the instrumentation module

4.4 The IA-32 instruction decode module

We have written a module to decode the IA-32 instruction set with control flow analysis and memory access registration optimization in mind.

A single IA-32 instruction can perform up to two distinct memory accesses. Most instructions of the IA-32 instruction architecture may access memory by using an address that is based on a combination of registers and optionally an offset. If at least one register is involved it is called indirect addressing, otherwise it is called absolute addressing. An address can be calculated from (almost) all combinations of the following components:

- a 32-bit displacement
- a base register
- an index register with scale factor 1,2,4 or 8 (handy for addressing an element in an array)

The base and index register can be the same, and can be one of the eight 32-bit general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI.

For example, an instruction could refer to the indirect address $(12345 + EBX + 4 * EAX)$, with 32-bit displacement 12345, base register EBX and index register EAX with scale factor 4. If EAX has value 3, and EBX has value 100, this results in address $12345 + 100 + 4 * 3 = 12457$. For more information, see the IA-32 basic architecture manual [INT02].

For each memory access, we decode:

- the size of the access in bytes
- the type of the access (load, store or modify)
- which registers are involved in indirect addressing, with which scale factor.
- the displacement of the access (a 32-bit integer)

We identify exceptions to the general way of accessing memory that we cannot optimize statically (e.g., bit test and bit set instructions, string instructions with a repeat prefix, because these depend on the “direction flag” of the CPU, and conditional move instructions).

We determine which registers may be changed by an instruction. This is important to be able to detect the relative locations of memory accesses when indirect addressing is used.

For control flow analysis, we identify instructions that change the control flow, such as branch instructions, so that we can easily recognize where to start new blocks and where to create edges in the control flow graph.

The IA-32 instruction set is extensive and there are many exceptions to the general instruction format and memory access types and sizes, so this module has cost a fair amount of development time.

4.5 The instruction instrumentation module

We have derived much of this module from the original instrumentation module. However, we have made it re-entrant. Therefore we no longer have to worry about locking and we can let different threads use it simultaneously if desired.

The module uses the information generated by the memory access optimization module to insert memory access registrations before the instruction.

Because we needed to modify the instruction instrumentation module according to the new design, it was cheap to implement another feature that was on the wish-list for DIOTA: separate instrumented code for each thread. Although this costs more memory, it also allows for more optimizations. DIOTA can now write separate memory access registration handler functions such as `diota_trace_` for each thread, and incorporate information in them that is specific to the thread, such as the thread ID and the address of the `errno` variable of the thread. This saves time because calls to the functions `pthread_self` and `__errno_location` are eliminated.

The handler function `diota_ret_handler_` that translates return addresses to an address of instrumented code for instrumented 'return' instructions has been optimized in the same way, by incorporating a thread-specific cache-lookup in the handler. Other optimizations that do not directly relate to data race detection have also been implemented.

4.6 Plain optimization of the registration code

In front of every instruction that accesses memory, DIOTA inserts instrumentation code that calls back the data race backend. Basically, this works as follows (see also Figure 2.1 on page 3):

- The data race backend tells the callback module of DIOTA that it wants to be informed of all memory accesses that the application performs, via the backend's function `register_memory_access`.
- DIOTA inserts the appropriate code into the application, which calls DIOTA's assembly function `diota_trace_`. The inserted code must be kept as short as possible to minimize memory requirements and improve cache performance.
- During execution of the instrumented code, `diota_trace_` saves registers to make the transition to C code safe, then calls a C function `diota_execute_data_callbacks`.
- `diota_execute_data_callbacks` calls all functions of all backends that must be informed of memory accesses (several different backends may be active at the same time, the data race detector is just one of them). One of these functions is the `register_memory_access` function of the data race backend.
- `register_memory_access` registers the memory access.

Simple things first: most accesses to memory are 4-byte accesses. By writing three optimized versions to replace the function `diota_trace_` for 4-byte read access, 4-byte write access, and 4-byte modify access, several instructions can be saved for each 4-byte memory access.

```

address   instruction           comment
2001:      push %eax
2002:      push $13                ; address of instruction that accessed memory
2003:      leal 0x8048410,%eax    ; address of accessed memory
2004:      push %eax
2005:      push $4                ; 4 bytes of memory accessed
2006:      push $2                ; 2 = access type 'STORE'
2007:      call diota_trace_      ; register the access for data race detection
2008:      pop %eax
2009:      mov %eax,0x8048410    ; original instruction

```

*Code fragment 4.1: Instrumented code fragment, unoptimized. Instructions to call the data race backend are in **bold**.*

To clarify this, the instrumented code of Code fragment 4.1, as DIOTA generated it before the optimization (destination operands are the rightmost operands), can be reduced to the instrumented code of Code fragment 4.2.

```

address   instruction           comment
2001:      push %eax
2002:      push $13                ; address of instruction that accessed memory
2003:      leal 0x8048410,%eax    ; address of accessed memory
2004:      call diota_trace_S4_   ; optimized diota_trace_ for STORE of 4 bytes
2005:      mov %eax,0x8048410    ; original instruction

```

*Code fragment 4.2: Instrumented code fragment, optimized. Instructions to call the data race backend are in **bold**.*

The address of the accessed memory is now passed to `diota_trace_S4_` via register `%eax`, and the arguments 'STORE' and '4 bytes' are implicit. The subroutine `diota_trace_S4_` restores the value of register `%eax` itself. This eliminates 4 push instructions of the inserted code. In addition, instructions are saved inside the optimized function `diota_trace_S4_`. The elimination of the push instructions also saves space, which is better for the instruction cache performance and leaves more space for data race detection, and saves DIOTA some time when writing the instrumented code. We also introduced optimized versions for stack access (push and pop of 4 bytes, push and pop of any number of bytes), with the same advantages and even more saved instructions.

We modified the interface of the instrumented code with the backend functions. Now a pointer to a structure with fields (address, length, type, ip, esp) is passed on as argument by all variations of the `diota_trace_` function. This saves time with respect to the old method that required copying the separate fields of the structure again and again. The `diota_trace_` functions now call the backend functions directly instead of via the callback module.

Inside the data race backend itself we removed a level of indirection by removing an intermediate function call. The performance gain is evaluated in Section 7.2.

5 The control flow analysis module

In this chapter, we present a basic analysis that allows for the optimization of memory access registrations. It is not straightforward to combine or eliminate memory access registrations, because any instruction is a potential branch target. We will explain this in Section 5.1. We will look at several candidate methods to overcome this problem in Section 5.2, and motivate why we choose the control flow analysis method. We solve several difficulties that are encountered during the generation of a control flow graph and present solutions to these problems in Section 5.3 and its subsections. In Section 7.2 we evaluate the cost of control flow analysis and show that control flow analysis is not more expensive than the original method.

5.1 Correct optimization of memory access registrations

It is not straightforward to combine the registration of memory accesses, based on the observations in Section 4.1. To explain this, Code fragment 5.1 presents an example code fragment to optimize.

```
original instructions
address  instruction
1:         movl %ebx,8      ; store the value of register %ebx at memory loc. 8
2:         movl %eax,12    ; store the value of register %eax at memory loc. 12

unoptimized instrumented instructions
address  instruction
2000:     <call back the data race backend to register the 4-byte store at
          address 8 by 'movl %ebx,8'>
2009:     movl %ebx,8      ; original instruction
2010:     <call back the data race backend to register the 4-byte store at
          address 12 by 'movl %eax,12'>
2019:     movl %eax,12    ; original instruction
```

Code fragment 5.1: Code fragment with corresponding instrumented instructions that can be optimized

It seems safe to combine the registration of the first write access of 4 bytes at address 8 with the registration of the second access of 4 bytes at adjacent address 12 into a single registration of a write access of 8 bytes at address 8, as shown in Code fragment 5.1.

```
address  instruction
2000:     <call back the data race backend to register the combined 8-byte
          store at address 8 by 'movl %ebx,8' and 'movl %eax,12'>
2009:     movl %ebx,8      ; original instruction of address 1
2010:     movl %eax,12    ; original instruction of address 2
```

Code fragment 5.2: Unsafe optimization for Code fragment 5.1

DIOTA will replace every branch in the original program to address 1 by an instrumented version of the branch instruction that jumps to address 2000 in the instrumented code. However DIOTA will replace every branch to address 2 in the original program by an instrumented branch instruction with address 2010 as target. This means that if a branch to address 2010 is executed, we will not correctly register the access of 4 bytes at address 12. So this optimization is unsafe.

Unfortunately, there is no way to be sure that a branch to instruction 2 will never occur, because it is too time-consuming to analyze the entire program, and we may not be able to analyze the program completely anyway because of indirect branches. So an optimization can only be applied if we deal with (unexpected) branches into the middle of code for which the registration has been optimized correctly.

Another problem are signals and exceptions that interrupt the normal flow of code of the program to

execute a signal/exception handler. The added instrumentation will not cause exceptions, so this cannot be a problem. However, a signal/exception handler may never return to the instruction that was about to be executed. So if we combine registrations and register memory accesses early, and an instruction causes an exception or an interrupt occurs before all instructions for which the accesses have already been registered have been executed, we may register too many accesses. If we register memory accesses late, we may register too few accesses and miss a data race. We therefore choose to register accesses early, because we prefer a possible false data race over a possible missed data race. We do not guarantee correctness if a synchronization operation is executed by a signal/exception handler. This can be a problem for normal instrumentation without combination of memory access registrations, too, because a signal may occur in between the instrumentation and the instrumented instruction. To treat this correctly it may be necessary to treat signal/exception handlers as threads themselves, and that is outside the scope of this project.

5.2 Alternative methods for correct optimization

The subparagraphs present methods to optimize memory access registrations correctly. We have chosen to implement the control flow analysis the method of Section 5.2.3, because it has several important advantages over the other two methods.

5.2.1 Method 1: Writing double instrumented code

One way to optimize the code correctly is by creating separate instrumented code for each branch target. For the code of Code fragment 5.1, with both instruction 1 and 2 as branch target, this would mean that we create a separate instrumented version for the case that instruction 2 is a branch target. This is shown in Code fragment 5.3.

```

address   instruction
2000:      <call back the data race backend to register the combined 8-byte
           store at address 8 by 'movl %ebx,8' and 'movl %eax,12'>
2009:      movl %ebx,8      ; original instruction of address 1
2010:      movl %eax,12    ; original instruction of address 2

address   instruction
2020:      <call back the data race backend to register the 4-byte store at
           address 12 by 'movl %eax,12'>
2021:      movl %eax,12    ; original instruction of address 2

```

Code fragment 5.3: Correct optimization for Code fragment 5.1

With this method, a branch to instruction 2 is correctly translated to the separate instrumented version at address 2020 instead of to address 2010. The disadvantage of this method is that because of the multiple instrumented versions you need very much more memory for the instrumented code. But more importantly, DIOTA must follow the policy that every instruction of the original program has at most one instrumented version, so that if the program modifies its own code, we can quickly look up and replace the instrumented version of that code. The lookup and replacement would become very difficult and time-consuming if instructions occurred several times in the instrumented code (e.g., instruction 2 is present in Code fragment 5.3 at 2010 and 2021, and both occurrences would have to be replaced), because we then would have to search and replace more than one occurrence of every modified instruction. For these reasons, we have rejected this method.

5.2.2 Method 2: Skipping subsequent registrations

Another way to optimize the code of Code fragment 5.1 correctly is the method of Code fragment 5.4. A branch to original instruction at address 1 will branch to address 2000 in the instrumented

version of the code, and a branch to address 2 will be replaced by a branch to address 2010. In either case, the registration code is executed only once, and it will register the accesses correctly. This will save time, which is our goal, and it satisfies the requirement that every instruction of the original program has at most one instrumented version. It will not save memory: it costs an extra jump instruction after every registration, and it still costs space for two registrations.

```

address  instruction
2000:      <call back the data race backend to register the combined 8-byte
          STORE access at address 8 by 'movl %ebx,8' and 'movl %eax,12'>
2009:      jump to address 2020
2010:      <call back the data race backend to register the 4-byte STORE access
          at address 12 by 'movl %ebx,8' and 'movl %eax,12'>
2019:      jump to address 2021
2020:      movl %ebx,8      ; original instruction of address 1
2021:      movl %eax,12     ; original instruction of address 2

```

Code fragment 5.4: Safe optimization method 1 for the code fragment of Code fragment 5.1

This method of jumping over subsequent redundant memory access registrations can easily be expanded to combine the registration for more than two instructions. By using a slightly more sophisticated version, we can also efficiently combine the registration if there are other harmless instructions in between the instructions that access adjacent memory locations. A harmless instruction is an instruction that can not influence the control flow, and that does not change a register that is used for indirect addressing by the instructions for which the memory access registration must be combined. Examples of instructions that are not harmless are branch, call and system call instructions.

5.2.3 Method 3: Control flow analysis

If we were able to tell which instructions are guaranteed never to be jumped to by the program, we can identify instruction blocks starting with an instruction that can be jumped to, followed by all subsequent instructions that are guaranteed never to be jumped to by the program, until we encounter the next instruction that can be jumped to. We could then safely use the optimization of Code fragment 5.2 on these blocks. Unfortunately, we would have to analyze the complete program and we may not even be successful because of, e.g., indirect branches that defy analysis, or self-modification of code. However, as we will explain now, we can guess with a very high accuracy.

The code of a normal program consists of functions that call one another. With the exception of returns from subroutines, jumps from one function into the middle of an other function almost never occur. Some programming languages do not even allow such jumps: if only structured control flow statements, such as if-then-else, while-do, continue and break are used, such jumps are impossible. An explicit 'goto' statement is necessary to accomplish such a jump (and sporadically a compiler will put in such jumps itself to optimize the generated machine code).

Therefore, if we know all possible target addresses of all the branch instructions inside a function, and we take returns from subroutines into account, we know which instructions of the function are likely jumped to. We call these instructions and the first instruction of the function 'entry points'. All other instructions are very unlikely jumped to, and considered not to be an entry point until proven differently.

We associate an instruction-block with each entry point, each block consisting of the entry point itself and all subsequent instructions up to but not including the first subsequent entry point. We build a *control flow graph* with instruction-blocks as nodes, and control flow transitions, such as the transition from a branch instruction to its target, as edges that flow from the end of one instruction

block to the begin of another. In fact, we are reverse-engineering the function to try and identify its 'basic blocks' (this is a term for a well-defined sequence of consecutive instructions used in the field of compiler technology; see [AHO86], chapter 9, for a definition of a basic block). We can analyze and optimize the instruction blocks individually, or we can analyze the control flow graph to identify memory access registrations that always precede (or follow) other memory access registrations, and see if we can combine them or leave some of them out.

We need a way to deal with the situation that an instruction that we considered not to be an entry point (an instruction in the middle of an instruction block) proves to be an entry point after all. As long as we forbid DIOTA to ever translate a branch target into the middle of an optimized instrumented instruction block, and DIOTA can recover from the incorrect guess, this is not a problem. We basically solve this as follows: DIOTA should conclude that there is an additional entry point, and replace the existing instrumented instruction block by splitting it into two new instruction blocks, of which the first block starts with the entry point of the replaced block, and the second block starts with the newly discovered entry point.

We can only take this approach because it is very unlikely that branches from one function into the middle of another function occur. Otherwise, DIOTA would identify the instruction blocks incorrectly very often and would lose too much time re-instrumenting code to correct its guesses. This would ruin the performance optimization we are trying to achieve.

The advantages of this approach with respect to the other method are:

- The resulting optimized code will execute faster and will be smaller than that of method 1. Smaller code means reduction of the CPU's instruction cache load. It also means more space for instrumented code in the cache (DIOTA has a large cache for instrumented code and when it is full, it is emptied completely), and thus longer intervals between re-instrumentation, which also improves performance.
- More thorough optimization becomes possible than with the method of the previous section. If we are careful, we can even optimize the entire control flow graph of a function as a whole, or subsets of its instruction blocks.

5.3 Generating a control flow graph

We will use control flow analysis to analyze a function and build a connected directed graph with instruction blocks as nodes, and directed edges that reflect all possible branches within the function: a control flow graph. We will build this graph as described in the subsections.

DIOTA can perform control flow analysis successfully because it instruments code dynamically. Static analyzers, which analyze the program file without executing the program, have a very hard time dealing with problems such as indirect branches. Because DIOTA is dynamic, it can simply solve or recover from such problems just-in-time before the problematic code is executed and the dynamic information that the static analyzers do not have becomes available.

5.3.1 Basic method for building a graph

The analysis starts an instruction block at the target of a call instruction: the entry point of a function.

- After the first instruction of an instruction block, we add all instructions in the subsequent memory locations to the block, until we encounter a (*system*) *call instruction*, a *branch instruction* or a *return instruction*, or an instruction that forms the start of an other block that we have already identified.

- If we encounter a call, branch or return instruction, this instruction is added as the last instruction of the block, and the block is complete. The completed block will be referred to as the '*previous block*' in the text below.
 - If we encounter the first instruction of an already identified block, this instruction is not added to the current block, and the current block is complete.
- If we encounter a (*system*) *call instruction*, and the branch target is not already part of an instruction block, we will start a new instruction block with the branch target, and add a directed edge to it from the previous block. We will not analyze the subroutine referred to by the call instruction.
 - If we encounter a *direct branch instruction*, and the branch target is not already part of an instruction block, we will start a new instruction block with the branch target, and add a directed edge to it from the previous block.
 - If the branch target is the first instruction an existing instruction block, we will only add a directed edge to that block from the previous block.
 - If the branch target is inside an existing instruction block, but not its first instruction, we will split that block into two parts at the point of the branch target, and add an edge from the first part to the second part, and an edge from the previous block to the second part.

If it is a *conditional branch instruction*, we will also start a new block with the instruction at the memory location after the branch instruction, and add an edge to it from the previous block, unless this is the first instruction an existing instruction block, in which case we only add an edge to this existing block.

- If we encounter an *indirect branch instruction*, we have a problem, because we can not know the branch target (except in a limited number of trivial cases that almost never occur), because the target is stored in memory or in a register, and may be altered or calculated dynamically during execution. This kind of branch instruction can appear for example if a 'switch' statement is used in a C program. We have no choice but to end the current instruction block here. We cannot start one or more new blocks, because we do not know all possible branch targets of the indirect branch instruction. We will have to analyze and instrument the code at the branch targets just-in-time by intercepting the indirect branch. Therefore, the control flow graph will not represent a complete function. The instructions that we cannot analyze yet can also jump to the middle of blocks we have already analyzed. Therefore we are less confident that we have identified all blocks correctly: some of them may be too big and should have been split.
- If we encounter a *return instruction* (an exit point of the function), we only complete the current block, we do not start a new block for the instruction after the return instruction. We treat a *halt* instruction the same way.

If we do not encounter indirect branches in the function, control flow analysis will yield a control flow graph of a complete function with a very high probability. This control flow graph contains every instruction of the function in exactly one block, and has edges that, when combined, represent all possible flows of control through the function. For an example control flow graph see Figure 5.1 on page 27.

5.3.2 Invalid addresses and invalid instructions

Our method of building the graph seems to have covered everything, but if we look closer, even if there are no indirect jumps, it is not guaranteed to work. Consider Code fragment 5.5.

```

address  instruction
1:         mov $1, %eax
2:         add $1, %eax
3:         jne 8          ; this instruction always branches to instruction 8
4:         <some data here, but no valid instructions>
8:         movl %eax,4

```

Code fragment 5.5: The problem with conditional branches and call instructions

Because the result of the 'add' instruction is never zero, the 'jne' instruction always branches, and the execution never reaches address 4. However, our control flow analysis would assume that there were instructions at address 4. This example illustrates that in general we have no way of knowing if there are valid instructions after a conditional branch, or at the target of a conditional branch. The target address may not even be a valid address (we recently learnt that a version of Sun's Java Virtual Machine generates conditional jumps that are always taken and that are not followed by valid code). A similar problem occurs for call instructions and for system call (INT 0x80) instructions. These instructions may never return (they can exit the program or enter an endless loop), so there is no guarantee that instructions follow after a call or INT instruction. In practice, there are virtually no programs that contain such code. However, we will have to deal with this possibility correctly to prevent that DIOTA crashes.

One way to make sure that our control flow analysis of a function is both correct and complete, is by delaying it until all conditional branches in a function have been taken at least once and have been 'not taken' at least once, and all calls and system calls in the function have returned at least once. Since we may need to wait for this a long time (even forever), this is not feasible. However, there is an alternative, because the damage that can be done by analyzing an unreachable memory area is limited:

- we can encounter an invalid instruction. We can then safely stop our analysis at that point.
- we can encounter an invalid address (which would cause a segmentation fault if we tried to access it to analyze an instruction there). So we have to validate the addresses of all instructions before we access them. We can then safely stop our analysis at that address.
- we can encounter a branch instruction that causes a valid existing instruction block to be split in two. This would mark the target instruction as a branch target, and could lead to suboptimal optimization.
- we can encounter a branch instruction that has a target in the middle of a multi-byte instruction in an existing instruction block. We can detect this, and our analysis will fail in a safe way (except for some special cases where this is actually ugly but valid code).

So analysis of unreachable memory areas can never lead to unsafe optimizations, the worst that can happen is suboptimal optimization, and some extra work.

5.3.3 Call instructions and system call instructions

One other problem deserves our attention: call instructions and system call instructions may never return, but exit the program instead. To deal with this, we break off instruction blocks at these instructions as well, and we create an edge from the end of the block to an artificial exit node (artificial because it contains no instructions) as well as an edge to the block with as first instruction the instruction following the call instruction. We also connect all other exit points of the function to this exit node.

In addition we create an edge from a virtual entry node that has no instructions to the block with as

first instruction the instruction following the call instruction. This is necessary because the graph may be entered here as well, either by a return instruction of the called subroutine, or by any other instruction (we cannot tell the difference, so we cannot forbid other instructions to jump here).

Figure 5.1 displays an example control flow graph for a simple function.

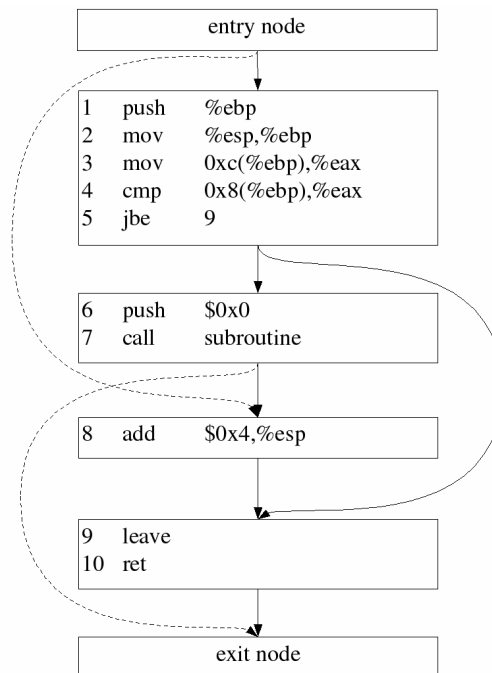


Figure 5.1: example control flow graph with exit node. Note the edge from the call instruction to the exit node to account for the fact that the called subroutine may not return. Also note the edge from the entry node to the instruction 8 that indicates that the graph may be entered from outside there (normally by the return from the called subroutine).

We also need to be able to detect if an instruction (or node) C is guaranteed to have been executed prior to an instruction (or node) D. Again 'call' instructions cause problems: the first instruction D' after a call instruction C' can be jumped to by the 'ret' instruction of the called subroutine, but also by another instruction outside the control flow graph (even a 'ret' instruction of another subroutine that has modified its own return address), and without rigorous and time-consuming administration there is no way to tell if D' has been executed prior to instruction C'. This is an additional reason that we cannot optimize through call instructions.

5.3.4 Recovering from incorrect identification of instruction blocks

If, during analysis, DIOTA encounters a branch that targets an instruction somewhere inside what it had identified earlier as an instruction block in an other control flow graph, and that block has already been instrumented, a problem arises. The instrumented version of the block may be optimized individually, as part of an entire function, or as part of a subset of blocks of a function. Thus the targeted instruction may not be safely jumped to. We will describe how to deal with this problem for the case that the block has been optimized individually first.

An option that is not open to us is to leave the instrumented version of the instruction block as it is, and create an additional instrumented version of the last part of the block, starting at the newly discovered branch target. We cannot do this, because DIOTA has a policy that every instruction of the original program may have at most one instrumented version. This policy is required for

efficiency of one of DIOTA's other features (self-modifying code support) that is under development by lic. Jonas Maebe at the moment of this project. Therefore, the existing block will have to be invalidated.

However, since we know that the block can currently be entered only at its first instruction (its entry point), we can replace the instruction at that entry point by a branch instruction. For this purpose, we require that every instrumented instruction block is big enough to be replaced by a 5-byte branch instruction, meaning we have to add 'nop' (no operation) instructions to the blocks that are smaller than 5 bytes (which almost no block is).

The 5-byte branch instruction is enough to be able to make DIOTA re-instrument the block when it is entered the next time. After its re-instrumentation this branch can be redirected to the re-instrumented version. We cannot re-instrument it immediately because we miss some information (for the insiders: DIOTA needs the value of the GOT pointer, which is stored in register EBX or ECX, which is not available until the next time the block is about to be executed), so we will instrument it later. However, DIOTA can already re-instrument the last part of the block starting at the newly discovered branch target, and continue execution normally.

If the instrumented block has been optimized as part of a set of instruction blocks (such as an entire function), we will have to invalidate and re-instrument all blocks that are involved. This is time-consuming, so we should avoid optimizing sets of instruction blocks if we are not confident that all the blocks involved have been identified correctly.

Note that blocks that replace invalidated blocks can be invalidated itself in exactly the same way, although it is highly unlikely to be necessary in practice.

5.4 Other benefits of control flow analysis

In this chapter we have explained how to build a control flow graph. With respect to the previous method of instrumentation, it has several benefits:

1. The previous implementation would follow the not-taken direction of branch instructions, and continue into called subroutines. Often the code at a branch target would not be instrumented, making it necessary to place a 'trampoline' in the instrumented code, a short sequence of instructions that returns control to DIOTA if the branch is taken during the execution of the instrumented code, so that DIOTA can instrument the code just in time. This means extra context switches between the instrumented code and DIOTA. Often the control flow graph will represent exactly one function or procedure, with only local branch targets. Forward references by branch instructions can thus be patched in a natural way, by following the edge from the block that ends with the branch instruction to the target block and using the address of the instrumented code of that block. This means a nearly 100% patch rate, resulting in fewer context switches and more compact instrumented code. In addition, we can now tell very easily if we will be able to patch a branch target beforehand (we will only be able to patch it if the target block is in the control flow graph or the target instruction has already been instrumented as part of a different control flow graph). This means that no time is wasted on failed patch attempts.
2. A function often consists of several instruction sequences with alignments in between (alignments are often filled with an unreachable valid instruction of the correct length with no effect, such as 'mov %esi,%esi'). These alignments are not handy for DIOTA, because each memory region for which instrumented code is generated needs to be registered separately in the cache of instrumented code. Because a control flow graph often represents a complete function, and the blocks of the control flow graph are ordered by the order in which their original code appears in memory, it is easy to detect alignments in between the blocks, and to register only one

big memory region in the cache of instrumented code. This saves a lot of space in the index tree of the instruction cache.

Control flow analysis has other memory usage requirements than the old method. The blocks with the decoded instructions and the edges connecting them cost extra memory. However, two internal tables of the old method (the translation table and the relocation table) are no longer needed.

6 Optimization of memory access registrations

In this chapter we present a basic method and more advanced methods to optimize the registration of the memory accesses that are performed by an application, for faster data race detection.

First we generate a control flow graph. Once we have a control flow graph, we can start the analysis for optimization of memory access registrations. There is a performance trade-off between the time we spend optimizing and the time that is saved during execution. Therefore we will try several optimization methods. Section 6.1 describes a simple method that optimizes individual nodes of the control flow graph. A more thorough method that takes the entire control flow graph or subsets of its nodes into account and that can detect redundancy and opportunities for combination for instructions across different nodes is described in Section 6.2. Even more aggressive optimizations are discussed in Section 6.3 and 6.4.

Although we create control flow graphs for individual functions, we may also create a control flow graph for other sets of instructions: as long as we prevent that DIOTA ever writes an instruction that jumps from outside the graph to an instruction other than an expected entry point of the graph, our optimizations are guaranteed to be correct. Thus we can apply the same methods to DIOTA's *inline* module. However, this appears impractical for data race detection, as we explain in Section 6.5. Finally, in Section 6.6, we briefly discuss how to build and start DIOTA and the data race detector to enable the different optimizations.

6.1 Intra-node optimization of indirectly addressed memory accesses

As we have shown in Section 4.2, many calls to functions that accept arguments are preceded by a number of push instructions. These push instructions are an example of instructions that are almost always in the same node of the control flow graph, and of which the memory access registrations can be combined without much effort. More generally speaking, it takes much less effort to analyze each node separately and combine as many registrations in it as possible than it takes to analyze the entire control flow graph to also find opportunities for optimization among instructions of different nodes. We will try to analyze each node separately first to see if we can improve performance with a minimum of effort.

In Section 6.1.1 we explain how we can keep track of the relations between registers. Once we know the relative distance between the value of two registers or between a current register value and its previous value, we can tell if memory accesses that use these registers for indirect addressing overlap or are adjacent. This is determined in Section 6.1.2. We need a strategy to combine the access registrations based on the analysis results, and for every byte in a combined memory range, we need to be able to tell which instruction accessed it. These topics are discussed in Section 6.1.3.

6.1.1 Keeping track of register relations

Access to memory via pointers is usually done by moving the pointer into a register, and then using indirect addressing. For example: take a local variable that resides in the stack frame at address $EBP - 8$. A pointer to this variable can be obtained in register EAX with the instruction `'lea -8(%ebp),%eax'` (Load Effective Address, with as result: $EAX = EBP - 8$). If after such a statement EAX is used for indirect addressing by an instruction, e.g., `'movl $1, (%eax)'` (which stores the value 1 at address EAX), we know that the access is in fact in the stack frame at address $EBP - 8$. If there has already been an access to that location or an adjacent location using EBP instead of EAX, the registration of the access using EAX for indirect addressing might be redundant or we may be able to combine the registrations.

We wish to detect redundancy and opportunities for combination, even if different registers are used to access the same or adjacent memory locations. We can accomplish this by monitoring all instructions that assign a new value to a register. This enables us to detect relative distances between the values of different registers.

To be able to do this, we associate with each register i a state called 'register_relation': {register, begin_address, begin_register, relative_offset}. This state means that register i has a value that differs by 'relative_offset' from the initial value that was assigned to the 'begin_register' by the instruction that ends at 'begin_address'. We will illustrate this by applying the method to Code fragment 6.1.

```

address  instruction
1:      push %ebp
2:      mov  %esp,%ebp
3:      sub  $4,%esp      ; reserve 4 bytes on the stack for local variable L
4:      movl $0,-4(%ebp) ; initialize local variable L
5:      push $4          ; push the only argument '4' for Afunction
6:      call Afunction   ; call Afunction with argument '4'
7:      ...

```

Code fragment 6.1: Instruction sequence at the begin of a function, to demonstrate our register relation analysis

We start with fresh register_relation states. A fresh register_relation state for register i has:

- register = i
- begin_register = i
- begin_address = address of the first instruction of a node of the control flow graph or of the first instruction after an instruction that has assigned a value to the register that we cannot relate to its previous value or to the value of an other register
- relative_offset = 0.

For the EBP and ESP registers, the fresh states at the begin of the instruction sequence of Code fragment 6.1 are:

```

{ register = EBP, begin_address = 1, begin_register = EBP, relative_offset = 0 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = 0 }

```

After the instruction at address 1, which implicitly subtracts 4 from ESP, this becomes:

```

{ register = EBP, begin_address = 1, begin_register = EBP, relative_offset = 0 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -4 }

```

The instruction at address 2 assigns the value of ESP to EBP. Therefore we copy the begin_address, begin_register and relative_offset of the state of ESP into the state of EBP:

```

{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -4 }

```

We have determined that there is a relation between registers EBP and ESP: they now both have a value relative to the value that was assigned to register ESP by the instruction preceding the instruction at address 1. This means that from this point on, we can detect if accesses relative to ESP and accesses relative to EBP overlap or are adjacent.

The instruction at address 3 again subtracts 4 from ESP. The instruction at address 4 does not affect the value of ESP or EBP, the instruction at address 5 again implicitly subtracts 4 from ESP, as does the 'call' instruction at address 6 by pushing the return address onto the stack. So in the end we

have:

```
{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }  
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -16 }
```

Since we do not know anything about 'Afunction' (we do not want to spend the time to recursively analyze all functions completely before running the program, nor is it likely that we can do so, since some parts of the program are loaded dynamically), after the return from the call to the subroutine 'Afunction' (if it returns at all) we no longer know the relations between the registers, nor do we know if one or more synchronization operations have been performed. So unless we are lucky and know something about the subroutine, we have to take a fresh start after the call instruction. Synchronization operations are all functions that are 'call'ed, so these are handled correctly automatically.

It is also important to note that strictly taken, we are forced to use a fresh `register_relation` state for a register every time its value is loaded directly from memory by an instruction. This is because memory is volatile. Even a simple push of a register that is immediately followed by a pop of the same register (`push %eax; pop %eax`) is unsafe, because a parallel thread may overwrite the value in between the push and the pop. So if we assume incorrectly that the register still has the same value after the pop, we may forget to register some accesses at run-time because we have optimized their registrations away, or we may register some accesses incorrectly because we combined them and now they prove to be accesses to some other location. However, the first data race will still be detected correctly (namely, the change of the stored register value in between the store and the load). Therefore we could present the user with the choice to allow DIOTA to retain the data structure associated with a register if its value is loaded directly from memory, allowing for more optimization, but then we can only guarantee that the first data race is reported correctly.

6.1.2 Keeping track of memory accesses for optimization

In the previous paragraph we have explained how we keep track of register relations. Every (`begin_address`, `begin_register`) combination uniquely identifies a set of memory accesses of which the relative positions are known. We can keep track of these memory accesses by associating two tables with each (`begin_address`, `begin_register`) combination:

- a table with all STORE and MODIFY accesses relative to `begin_register`: the 'STORE table'.
- a table with all LOAD, STORE and MODIFY accesses relative to `begin_register`: the 'ANY table'.

The STORE tables will be used to determine which STORE and MODIFY access registrations are redundant or can be combined, following the rules of Section 4.1. The ANY tables will be used to determine which LOAD access registrations are redundant or can be combined. Note that we are still able to determine the actual types of the accesses in the ANY table.

```
address   instruction  
1:         push %ebp  
2:         mov  %esp,%ebp  
3:         sub  $4,%esp      ; reserve 4 stack bytes for local variable L  
4:         movl 8(%ebp),%eax ; initialize local variable L  
5:         addl 12(%ebp),%eax ;  
6:         mov  %eax,-4(%ebp) ;  
7:         push -4(%ebp)    ; push the value of L as argument for Bfunction  
8:         call Bfunction  ; call Bfunction  
9:         ...
```

Code fragment 6.2: Begin-instructions of a function to demonstrate our memory access analysis

For the example Code fragment 6.2, we get the following state after instruction 1, both the ANY table and the STORE table are associated with (*begin_address* = 1, *begin_register* = ESP):

```
{ register = EBP, begin_address = 1, begin_register = EBP, relative_offset = 0 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
ANY table: STORE at location -4 to -1 by instruction 1.
STORE table: STORE at location -4 to -1 by instruction 1.
```

Instruction 2 and 3 do not access memory, but change the register relations to:

```
{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -8 }
ANY table: STORE at location -4 to -1 by instruction 1.
STORE table: STORE at location -4 to -1 by instruction 1.
```

Instruction 4 LOADs 4 bytes at offset $-4 + 8 = 4$ relative to the initial value of ESP, so we get:

```
{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -8 }
ANY table: STORE at location -4 to -1 by instruction 1.
LOAD at location 4 to 7 by instruction 4.
STORE table: STORE at location -4 to -1 by instruction 1.
```

Instruction 5 LOADs 4 bytes at offset $-4 + 12 = 8$ relative to the initial value of ESP, so we get:

```
{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -8 }
ANY table: STORE at location -4 to -1 by instruction 1.
LOAD at location 4 to 7 by instruction 4.
LOAD at location 8 to 11 by instruction 5.
STORE table: STORE at location -4 to -1 by instruction 1.
```

Instruction 6 initializes local variable L with a STORE access:

```
{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -8 }
ANY table: STORE at location -8 to -5 by instruction 6.
STORE at location -4 to -1 by instruction 1.
LOAD at location 4 to 7 by instruction 4.
LOAD at location 8 to 11 by instruction 5.
STORE table: STORE at location -8 to -5 by instruction 6.
STORE at location -4 to -1 by instruction 1.
```

Instruction 7 performs two memory accesses: it LOADs the value of variable L and STOREs it by pushing the value onto the stack.

```
{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -12 }
ANY table: STORE at location -12 to -9 by instruction 7.
STORE at location -8 to -5 by instruction 6.
STORE at location -4 to -1 by instruction 1, LOAD by instruction 7.
LOAD at location 4 to 7 by instruction 4.
LOAD at location 8 to 11 by instruction 5.
STORE table: STORE at location -12 to -9 by instruction 7.
STORE at location -8 to -5 by instruction 6.
STORE at location -4 to -1 by instruction 1.
```

And finally, the call instruction STOREs the return address by pushing it onto the stack.

```

{ register = EBP, begin_address = 1, begin_register = ESP, relative_offset = -4 }
{ register = ESP, begin_address = 1, begin_register = ESP, relative_offset = -16 }
ANY table:  STORE at location -16 to -13 by instruction 8.
            STORE at location -12 to -9 by instruction 7.
            STORE at location -8 to -5 by instruction 6.
            STORE at location -4 to -1 by instruction 1, LOAD by instruction 7
            LOAD at location 4 to 7 by instruction 4.
            LOAD at location 8 to 11 by instruction 5.
STORE table: STORE at location -16 to -13 by instruction 8.
            STORE at location -12 to -9 by instruction 7.
            STORE at location -8 to -5 by instruction 6.
            STORE at location -4 to -1 by instruction 1.

```

So in this case we can combine the registrations of the LOAD accesses by instruction 4 and 5, and the registrations of the STORE accesses by instructions 1, 6, 7, and 8. The registration of the LOAD access by instruction 7 is redundant and can be eliminated because the STORE access at the same location is already registered. So instead of seven separate access registrations, we now have only two left. These two may be slightly more expensive because they register a bigger memory region, so the effect is that during execution we will need the time needed for 5 original registrations or slightly less than that per execution of the code fragment.

We can also encounter a situation like the following:

```

ANY table:  LOAD at location -16 to -13 by instruction 3.
            STORE at location -12 to -9 by instruction 2.
            LOAD at location -8 to -5 by instruction 1.
STORE table: STORE at location -12 to -9 by instruction 2.

```

Here, we can make use of the STORE operation to combine the registrations for the accesses by instructions 1, 2, and 3, and register a LOAD access from address -16 to address -5 relative to the initial value of ESP. We also need to register the access by instruction 2 separately as STORE access. So we are left with two registrations instead of three.

6.1.3 Optimization strategy

After we have constructed the STORE tables and ANY tables, we use a greedy algorithm to optimize the accesses:

For all instructions of the node, from first to last:

For all memory accesses by the instruction:

If it is a LOAD access and there is an earlier instruction that has registered an access that is at least equivalent (either LOAD, STORE, or MODIFY, look in the ANY table) to the same location:

Eliminate the registration for this access.

Else if it is a STORE or MODIFY access and there is an earlier instruction that has registered a compatible access (either STORE or MODIFY, look in the STORE table) to the same location:

Eliminate the registration for this access.

Else

Enlarge the registration of the instruction by including as many other adjacent registrations with a compatible access type that have not yet been eliminated as possible. For LOAD accesses, shrink the registered memory region so that it is delimited by a LOAD access at both sides.

The instrumented code for the instruction that combines the memory access registrations must now include an instruction that adjusts the address of the memory region. For example, in the optimized instrumented version of Code fragment 6.2 in Code fragment 6.3, the instruction 'lea -16(%esp,1),%eax' adjusts the address by -16 before registering the access (every 'push' instruction implicitly decrements %esp by 4 bytes, see the previous section).

<u>Address</u>	<u>instruction</u>	<u>optimized instrumented version</u>
1:	push %ebp	push %eax ; combined registration for instructions 1,6,7,8 lea -16(%esp,1),%eax ; adjusted access address push \$16 ; access length push \$1 ; instruction address call diota_trace_S_ ; register the STORE accesses push %ebp
2:	mov %esp,%ebp	mov %esp,%ebp
3:	sub \$4,%esp	sub \$4,%esp
4:	movl 8(%ebp),%eax	push %eax ; combined registration for instructions 4,5 lea 8(%ebp),%eax ; adjusted access address push \$8 ; access length push \$4 ; instruction address call diota_trace_L_ ; register the LOAD accesses movl 8(%ebp),%eax
5:	addl 12(%ebp),%eax	addl 12(%ebp),%eax
6:	mov %eax,-4(%ebp)	mov %eax,-4(%ebp)
7:	push -4(%ebp)	push -4(%ebp)
8:	call Bfunction	call Bfunction

Code fragment 6.3: optimized instrumented version of Code fragment 6.2 with combined memory access registrations

The data race detector now receives the address of the first instruction involved in each the combined access registration. This means that if a data race is detected for part of the accessed combined memory region, the address of the instruction that accessed that specific part needs to be deducted. This is very well possible, by re-analyzing the instruction block that contains the first instruction involved in the combined access registration. Data races occur infrequently, so even though this is a bit time-consuming, it the effect on the performance is minimal.

6.2 Inter-node optimization of indirectly addressed memory accesses

We can optimize more if we do not just look at the instructions per node, but look more globally at all instructions of all nodes of the control flow graph. This means that we need to spend more time to analyze the control flow, so it may not pay off. We distinguish between the following phases to statically analyze the memory accesses performed by the function, using its entire control flow graph.

1. For each node in the control flow graph, determine which relations between register values are guaranteed to exist, every time the first instruction of the node is executed.
2. For each node in the control flow graph, determine which memory locations are guaranteed to have been accessed (and what the type of the access was for each location), before the first instruction of the node is executed.
3. For each memory access in the control flow graph, determine if the registration of the access is redundant with respect to a preceding instruction and can be eliminated.

These phases are discussed in Sections 6.2.1, 6.2.2 and 6.2.3 respectively.

6.2.1 Determining the relations between registers at the begin of every node

We wish to use register-relations to optimize memory access registrations across the entire control flow graph. To be able to do this, we need to determine which relations between register values are guaranteed to exist at the begin of every node, so that we can determine the relative locations of indirect memory accesses using different but related registers inside every node. We can do this by using the technique of Section 6.1.1 embedded in a different algorithm.

Assuming that there are no loops and no nodes with multiple incoming edges, it is straightforward to determine the relations at the begin of each node:

- Assign a fresh begin-state consisting of a fresh register_relation for register EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI to every successor of the entry-node of the graph. This reflects the fact that we do not know anything about the registers at those points.
- Determine the end-state (again consisting of a register_relation for register EAX, ..., EDI) of each successor of the entry-node by analyzing the changes to registers by all its instructions.
- Use each newly determined end-state of a node as begin-state for its successors. Again determine the end-states for these successors. And so on and so forth, until we have determined the begin-state of all nodes of the graph.

If there are nodes with multiple incoming edges, it becomes slightly more complex: we then need to determine which relations hold irrespective of via which incoming edge such a node is entered. We can copy the relations that hold into the begin-state of the node, and use a newly initialized register_relation for registers that do not have the same register_relation in the end-state of all predecessors. For example, in the end-states of node A and B in Figure 6.1, the same relation holds for all registers except for EAX and EBX. Therefore, in the begin-state of node C that has both node A and B as predecessor, we need to use a fresh relation for EAX and EBX, as if they were changed unpredictably just before the first instruction of node C with address 15 is executed.

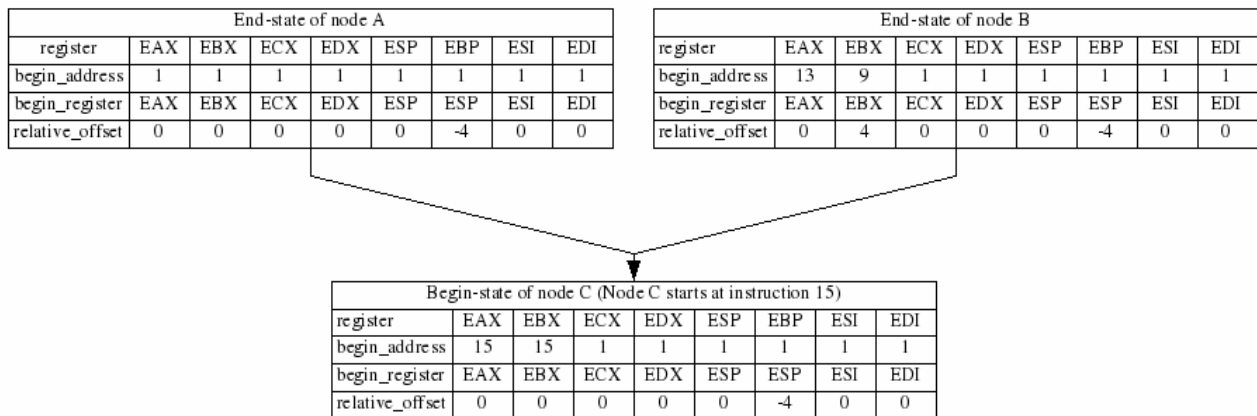


Figure 6.1: The begin-state of node C is deduced from the end-states of predecessors A and B.

If there is a loop, there is bound to be a node N with at least one predecessor of which the end-state will not be known until the node N itself has been processed. This is a cyclic dependence relation. Figure 6.2 presents an example of such a relation, where the begin-state of node N depends on the end-state of node N itself.

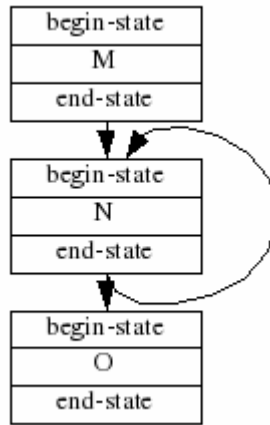


Figure 6.2: Cyclic dependence relation for node *N*

To handle the cyclic dependence relations, we need to perform a number of iterations over all nodes until we reach a stable situation. We start with an 'uninitialized' state as end-state for every node, and we ignore uninitialized end-states of predecessors when determining the initial state of a successor node. When in a later iteration the end-state of a predecessor becomes available or changes, we will re-determine the begin-state of its successors by using a fresh register relation for each register for which the register relations in the end-states of the predecessors do not match. Then we will propagate any changes through the other nodes of the graph during the same iteration and subsequent iterations until we reach a stable situation in which nothing changes anymore. At that point, we have distilled which register relations in the begin-states of the nodes hold irrespective of the predecessor via which the node is entered.

For the example of Figure 6.2, processing the nodes by preorder traversal of the graph during every iteration for high efficiency: first M, then N and finally O, this algorithm means that:

- During the first iteration, first determine the end-state of entry node M using its fresh begin-state. Then using the end-state of node M, and ignoring the missing end-state of node N, we initialize the begin-state of node N by making it equal to the end-state of node M. We use the begin-state of node N to determine the end-state of node N, which is then also used as the begin-state of node O.
- During the second iteration, we notice that the end-state of node N is now available from the first iteration. Just like we combined the end-states of node A and B in Figure 6.1, we now combine the end-states of node M and N from the first iteration to obtain a new begin-state for node N. This may lead to some new 'fresh' register relations in the new begin-state of N that were not in the previous begin-state of node N, if the instructions of node N change the value of one or more registers. If the new begin-state of node N has changed with respect to that of the previous iteration, we also re-determine the end-state of N and the begin-state of O.
- We perform additional iterations until there were no changes during the last iteration.

6.2.2 Determining which accesses are guaranteed to precede and follow each node

Now that we know which register relations hold irrespective of via which predecessor a node is entered, we can determine which memory locations are guaranteed to have been accessed at the begin of each node. Again this is straightforward if there are no nodes with more than one predecessor. Every node then 'inherits' the memory accesses of its predecessor (again STORE and

ANY tables are used, see Section 6.1.2) and every node adds the memory locations that are accessed by its own instructions to the set of accesses.

Nodes with multiple predecessors inherit from their predecessors only the memory accesses that the predecessors have in common, just like they inherited only the register relations that their predecessors had in common, as we have seen in the previous section. To deal with loops, we again use begin-states, end-states, and we start with 'uninitialized' states and perform multiple iterations until nothing changes anymore.

When comparing the accesses of predecessor nodes A and B of successor node C to determine which accesses they have in common, we compare the 'STORE' tables in the end-state of node A with the corresponding tables of node B to determine which STORE/MODIFY accesses they have in common, resulting in tables with all STORE/MODIFY accesses that are inherited by the begin-state of node C. We do the same with the ANY tables. An example for the ANY tables is given in Figure 6.3. The STORE access at address 2 by instruction 2 of node A does not re-occur in the begin-state, because the other end-state does not have an overlapping access.

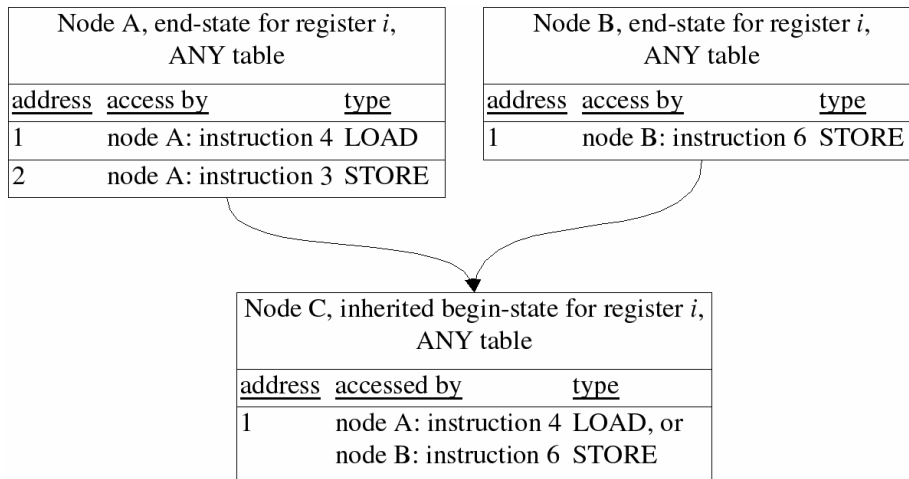


Figure 6.3: Example of the intersection of the 'ANY' tables for register i

We need the ANY table instead of a LOAD table to be able to eliminate LOAD access registrations that have been made redundant by a LOAD access in one or more predecessors and by a STORE access in one or more other predecessors. With a LOAD table instead, these preceding accesses would not have been inherited from the predecessors, because the predecessors do not have the LOAD access in common, nor do they have the STORE access in common. This can also be seen in Figure 6.3: the begin-state of successor node C inherits the access to address 1 from predecessor nodes A and B even though node A loads data from it and node B stores data into it.

6.2.3 Identifying candidates for elimination

In the previous section we described how we can determine the memory accesses that are guaranteed to have been performed on entry of a node using ANY tables and STORE tables.

In principle, every access registration that is guaranteed to be followed by another overlapping access registration of the correct type is a candidate for elimination. We can use the ANY tables and the STORE tables to identify the candidates.

Theoretically, every access registration A that is guaranteed to be preceded or followed by another adjacent access registration B of the correct type is a candidate for combination, if and only if

registration B is also guaranteed to be preceded or followed by registration A. However, we would have to analyze the graph in both directions to identify candidates for combination. If we did this, we would also need an optimization strategy, because it would not be straight-forward to choose which candidates to eliminate and/or combine and which to keep (this is an NP-hard problem). Additionally we have the problem of exceptions and/or interrupts as described in Section 5.1. Forward analysis suffices to be able to optimize loops by unrolling them (Section 6.4).

6.3 Eliminating registrations of stack memory accesses

If a multi-threaded program only shares global and/or allocated data, or the programmer is only interested to check for data races among global and allocated data, the data race detector does not need to check for data races on the stack. This means that DIOTA does not have to provide it with information about memory accesses on the stack, so we can eliminate all their registrations.

Using the register relations of Section 6.1.1, it is easy to tell in a large number of cases if an access is on the stack: if the 'begin_register' in the 'register_relation' of a register that is used in an indirect address calculation is register ESP, the access is on the stack and we can eliminate the registration.

It is simple to use a policy to only share global and/or allocated data in a multi-threaded program. Programs that also share data on the stack can often easily be converted to share only global and/or allocated data. This can be a great advantage for fast data race detection using DIOTA. Some performance results are presented in Section 7.5.

6.4 Loop unrolling

Sometimes a loop that does not contain synchronization operations accesses the same memory location during every iteration. Such accesses need to be registered only once. If we unroll the first iteration of each loop, we can instrument such memory accesses only in the first iteration. This may eliminate a great number of repeated registrations.

To be able to do this, we first need to identify loops, then determine which nodes of the control flow graph are part of each loop, and finally copy these nodes and insert them properly into the existing control flow graph. Then we need to perform intra-node optimization of the access registrations.

Because loop unrolling is computationally expensive, we only perform this when we are fairly certain that we are dealing with the control flow graph that represents exactly one function. This makes it less likely that the graph is incomplete and that the instrumented code needs to be discarded later, wasting our effort.

In the subsections, we first explain how we can identify 'natural' loops in a control flow graph using domination analysis. Then we show how we unroll the first iteration of a natural loop. Finally we discuss why loop unrolling can unfortunately not be applied to natural loops that contain call instructions.

Note that although loop unrolling violates DIOTA's policy 'one instrumented version for each instruction of the original program' for self-modifying code support, this is not a problem because loop unrolling is only applied to entire control flow graphs, and the complete control flow graph will be deleted if at least one byte of the original program code covered by the graph is self-modified. Thus self-modification is supported.

6.4.1 Identifying loops

There is a class of loops called 'natural loops' that can be identified at limited cost. A natural loop is a loop with only one starting point, and in structured programming languages all 'for', 'while', 'do ...

while', 'until' and 'repeat' statements form natural loops, because they have only one entry point (unless a separate 'goto' is used to also enter the loop). So the vast majority of all loops are natural. The node that is the starting point of the loop is called the 'loop header'. This class of loops can be identified by performing dominator analysis on the control flow graph.

A node V 'dominates' a node W iff V is on every path from any entry point of the graph to W . The dominator V of W that is the closest to W is the 'immediate dominator' of W . Cooper, Harvey and Kennedy [COO01] have published a simple, fast algorithm for dominator analysis. This algorithm determines the immediate dominator of every node. By using the property that if a node V dominates a node W , and a node U dominates V , U also dominates W , we can use the immediate dominators to determine all dominators of a node W very efficiently. We have implemented this algorithm.

Loops can be identified by scanning for 'backedges'. A backedge is an edge from a node B to a node A , where A dominates B [AHO86]. This makes A a loop header. All nodes in the loop with loop header A can be determined by first determining all immediate predecessors B_i of A that are dominated by A . All predecessors of every B_i up to and including the loop header are part of the loop.

6.4.2 Unrolling the first iteration of a loop

We can unroll the first iteration of a loop by doing the following:

- Copy all nodes of the loop and all edges that start from them.
- Reroute all original backedges to the copied loop header. The original loop header is no longer a loop header after that.

Figure 6.4 shows a loop consisting of nodes A , B , and C with header A on the left. On the right, loop nodes A , B and C have been copied to nodes A' , B' and C' , respectively, along with the edges that start from them. The backedge from C to A has been rerouted to A' . Nodes A , B and C have become the unrolled first iteration of the loop.

After unrolling a loop L that contains nested loops, both the unrolled first iteration of L and the copied loop L' contain the nested loops. Since both the unrolled iteration and the remaining loop are dominated by their (former) loop header, we can apply the algorithm recursively to unroll nested loops, to unroll each loop exactly once in the following way: traverse in pre-order both the nodes of the original control flow graph, starting from the entry node, and the copied nodes, starting with the successors of the copied loop header. Visit every node exactly once, except the copied loop headers, to prevent that we unroll a loop more than once. Pre-order means that we visit each node itself first, before visiting its successors. Thus we apply loop unrolling to each loop header node we encounter immediately, before applying it to its successor nodes and to the copied loop nodes to unroll nested loops and neighboring loops.

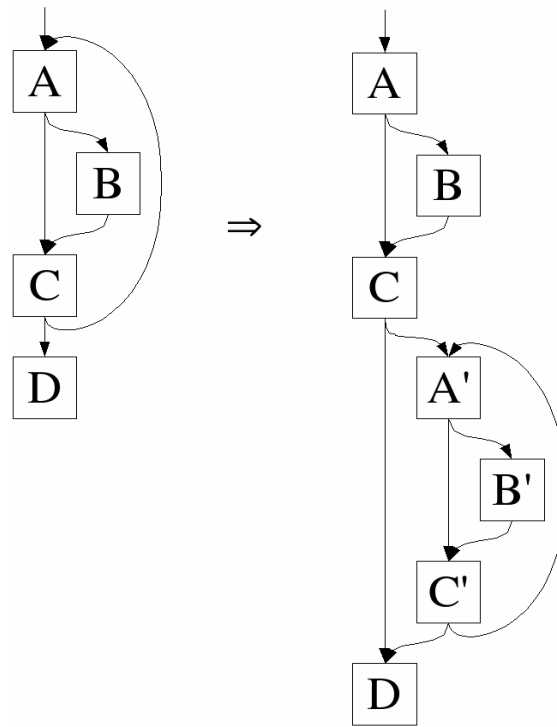


Figure 6.4: At the left the original loop, at the right the same loop with the first iteration unrolled.

6.4.3 Loops containing call instructions

Calls to subroutines inside loops are problematic. In addition to the general problem that we do not know if the call ever returns, that we do not know if the call executes a synchronization operation, and that we do not know if the subroutine changes registers or not, we now have an additional problem.

The instruction after the call is a return point, where the calling function normally resumes execution after the called function returns. If a subroutine is called by instrumented code, the instrumented code for the `call` instruction puts the original return address, that is the address of the instruction after the original un-instrumented `call` instruction, on the stack and then jumps to the called subroutine. When the called subroutine returns, its instrumented `ret` instruction translates the original return address to the address of the instrumented code associated with it, and returns to that translated address. All this is necessary to guarantee the correct operation of the program: the program may take conditional action by for example comparing the original return address it expects to find on the stack to another address, which would produce an incorrect result if the return address on the stack was an address of instrumented code.

However, if we have unrolled the first iteration of a loop, we have two candidate addresses of instrumented code associated with the original return address: an address in the unrolled first iteration and an address in the copied loop nodes. For nested loops this gets even worse: these are copied several times as part of the loops they are nested in, and then each copy is unrolled, resulting in many candidate return addresses. The candidate node that was part of the original loop is the only one we make known globally as entry point. Its copies are kept hidden from the outside world. This node is safe to return to, because nothing is assumed about relations between the registers when it is entered, so nothing has been optimized away incorrectly if we return from the called subroutine to the instrumented code of the unrolled first iteration of the loop. A disadvantage of this way of working is that part of the copied loop nodes will never be reached and we waste time and space instrumenting them. An advantage is that we can sometimes combine or eliminate memory access

registrations in the first nodes of the copied loop based on the memory access registrations in the last nodes of the first iteration, even though call instructions inside the loop limit the opportunities.

6.5 Why inlining is not practical to optimize data race detection

As we briefly mentioned in Section 2.1, DIOTA contains 'profile' functionality that analyzes which paths through the code are executed most often, and an 'inline' module that can write optimized instrumented code for such code paths.

The initial approach to this project was that we would optimize only the code generated by the inline module for data race detection. This attempt failed because of various problems. Most importantly, it was not possible to throw away any generated inlined code, because it was shared by all threads. If one thread would throw away inlined code, another thread could crash immediately, because the instructions it was about to execute were overwritten. This means that the amount of inlined code grows infinitely, or fills a limited amount of memory very soon, and may become outdated and unused very soon because other paths through the program become more popular.

Since then, DIOTA has been modified to generate separate instrumented code for every thread. Even though this means that DIOTA can now theoretically delete or replace the inlined code of a thread safely, several problems prevent us from being able to do so efficiently:

- There are references to the inlined code everywhere in the normal instrumented code that are impossible to administer and replace efficiently when throwing away inlined code. This would mean that we would have to replace the normal inlined code as well, and re-instrumenting code is very expensive.
- It is hard to tell what is more efficient when the maximum amount of inline code has been generated and we wish to inline some more code: should we throw away the old inlined code (expensive, as we have just explained), or should we keep the current inlined code, which could become outdated very soon, and stop inlining?

In addition, inlining is only practical for small multi-threaded programs with a limited number of threads because it costs much memory that can be used better to store the bitmaps of the data race detector. Mozilla [MOZ02] is an example of a bigger program for which the inlined code grows very fast and re-instrumentation is very expensive. This problem has taken bigger proportions now that DIOTA has been modified to generate separate instrumented code for every thread. An other disadvantage is that inlining is not compatible with optimizations such as the optimized `diota_ret_handler_`.

Using control flow analysis as basis for optimizations has the advantage that the policy 'one instrumented version for every byte of original program code' can be upheld, so control flow analysis is fully compatible with self-modifying code support.

This all appeared after some development effort, but not all effort was wasted because we could use the instruction decode module later on. Therefore we have not modified the inline module for optimization of the data race detector but we have focused on the optimization of normal instrumented code as presented in the previous sections of this chapter.

6.6 Building and using DIOTA and diota-dr with the optimizations

To build DIOTA with intra-node optimization, simply use 'make' without extra options. Inter-node optimization requires extra resources compared to intra-node optimization, and to save these resources per default, we have chosen for a build-time option. To build DIOTA with inter-node optimization, use

```
make O=-DOPTIMIZE_MEM_INTER_NODE
```

To also build DIOTA with loop unrolling (only applicable when using inter-node optimization), use

```
make O='-DOPTIMIZE_MEM_INTER_NODE -DOPTIMIZE_MEM_UNROLL_LOOPS'
```

The data race detector backend normally does not ask DIOTA to use memory access registration optimization, because this is not fully compatible with all other DIOTA backends. For example, the `diota-malloc` backend may fail to detect all invalid stack access (an access to an address on the stack below the current value of the ESP register) due to the combination of memory access registrations or the elimination of stack access registrations.

To enable memory access registration optimization during data race detection, define the environment variable `DIOTA_DR='optimize_mem'` aside from `LD_PRELOAD=<path>/diota-dr.so`. You can check that it has been enabled in the main logfile for the application (`/tmp/diota.log-<application>-<username>`): the message

```
diota-dr: memory access registration optimization enabled
```

should be present.

To enable memory access registration optimization and also eliminate memory access registrations for accesses to the stack as much as possible (Section 6.3), define `DIOTA_DR='optimize_mem_no_stack'` instead. This produces the message

```
diota-dr: memory access registration optimization enabled, stack accesses ignored when possible
```

in the main logfile.

If DIOTA has been built with inter-node-optimization and loop unrolling, loop unrolling is enabled automatically if one of the options `optimize_mem` and `optimize_mem_no_stack` is specified in the environment variable `DIOTA_DR`.

The data race detector is built together with DIOTA. It supports several types of bitmaps. The 10-level combined bitmap is currently compiled per default. To use the 3-level bitmap, specify `-DBITMAP_3LEVEL`, or to use the 9-level bitmap, use `-DBITMAP_9LEVEL`. Normally, the data race detector prints information about data races it detects to the terminal. To prevent this and only output the data races to a file, specify `-DQUIET`. For example:

```
make O='-DOPTIMIZE_MEM_INTER_NODE -DBITMAP_9LEVEL -DQUIET'
```

7 Evaluation

In this chapter, we will evaluate the effectiveness of the different optimizations proposed in the previous chapter. To determine this effectiveness we execute performance tests. We cannot compare the current version of the data race detector and DIOTA with all its performance optimizations to the original version of DIOTA as it was at the moment we started the work on this project, for two reasons:

- it is infeasible because the benchmark programs would not be able to finish with the original version of the data race detector without segment combination due to the much higher memory requirements
- it is unfair to compare them because the current version of DIOTA has evolved too much: there are too many changes with respect to the original version, e.g., bug fixes, functionality changes, other optimizations that do not directly relate to our optimizations but may influence performance.

Therefore we will take the reverse approach and turn some optimizations off if possible to see the negative impact on the performance. Often benchmark programs cannot even be run without these optimizations, because the data race detector will run out of memory immediately.

7.1 Benchmark setup

We have used several applications of the second release of the Stanford Parallel Applications for Shared Memory (SPLASH-2, they probably did not like the acronym 'SPASM') programs as benchmark set [WOO95], with m4 macros that use the pthreads library (see Appendix B). For each program we have used the default input specified in the documentation, unless noted otherwise. These programs use the SPMD execution model: all threads (sometimes except the main thread) execute the same algorithm on a part of the data. Often each thread performs one computation step then synchronizes with all other threads. This means that the segments can be combined very well. Because in addition they create threads only once and they allocate memory only once, these programs can run indefinitely with the data race detector enabled: they will never run out of memory for bitmaps, and will never run out of space for thread segments.

We have also included the multi-threaded browser 'Mozilla' [MOZ02] as benchmark program. This is an example of a program with threads that each execute different code, and threads do not synchronize in such an orderly way as those of the SPLASH-2 benchmark. This means that eventually the data race detector may run out of memory.

The machine we have executed our benchmark programs on is a dual Intel Celeron 500 MHz with 768 MB memory. We have measured the 'real time' that was needed to execute the benchmark programs. We have executed the SPLASH-2 programs with 2, 4 and 8 threads. For mozilla (6 threads), we have waited until the process showed no CPU activity after the requested page (<http://www.elis.ugent.be>) was shown, then pressed ALT-F4 to close mozilla. Because mozilla has internal time-outs, we have measured execution time a number of times for each different data race detector configuration and taken the minimum execution time.

We would have liked to include the replay backend (Section 2.2) in our tests to show how the data race detector is used and performs in practice. Unfortunately, recent changes to DIOTA have temporarily rendered this backend inoperable, so we have not been able to collect results.

7.2 The cost of instrumentation

For reference, we present the execution times of the applications when run normally without data

race detection and without instrumentation by DIOTA in Figure 7.1.

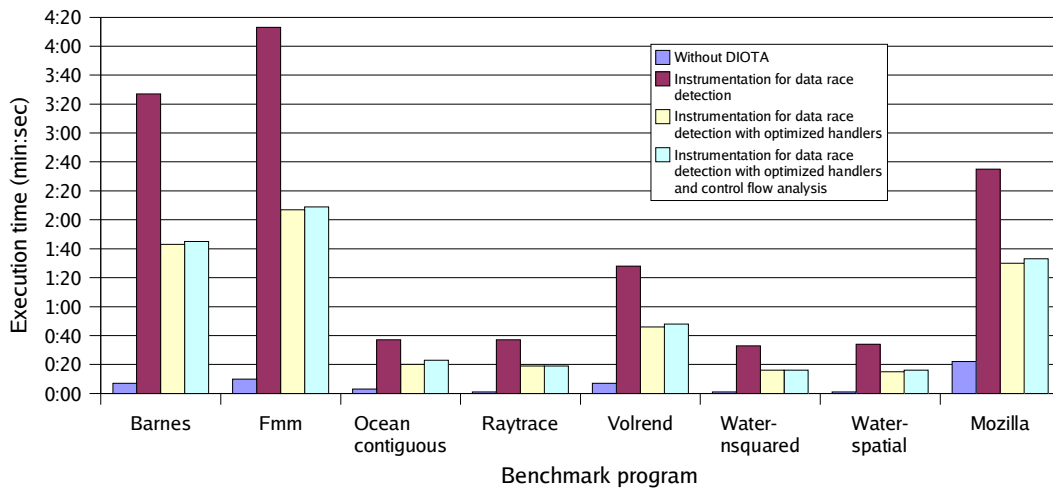


Figure 7.1: Basic cost of instrumentation and execution of the instrumented code including handler functions, but without actual data race detection. SPLASH-2 benchmarks executed with 2 threads. Mozilla has a fixed number of 6 threads.

We can determine an upper bound on the performance of the data race detector by making it request all information from DIOTA as normal, but letting the data race detector do nothing at all with this information. Then, in effect, we measure the sum of the time needed by DIOTA to instrument the application and the time needed by the instrumented code to provide the data race detector with the requested information. Every second column in Figure 7.1 shows the measured execution time for DIOTA without control flow analysis and without optimized handler functions (as the original version of DIOTA). Every third column shows the measured value for DIOTA without control flow analysis and with optimized handler functions (Section 4.6), and every fourth column shows the value for DIOTA with control flow analysis (Chapter 5) and with optimized handler functions.

We can conclude from these results that instrumentation slows down the execution considerably, that the optimized handler functions improve performance, and that by using control flow analysis to create opportunities for optimization we do not have to lose performance.

7.3 Segment combination settings

After the implementation of segment combination, we tested how frequent the data race detector should try and combine thread segments. Segment combination requires that a list of segment numbers is sorted. The segment numbers in this list are obtained from the vector clocks of the synchronization objects and of the current segment of each thread. We use the quicksort algorithm to sort these as fast as possible, but sorting them takes some time for every combination attempt. Therefore, trying to combine segments too often may not be optimal, because it may be impossible to combine segments and we may waste time sorting the segment numbers.

On the other hand, if we do not combine segments often enough we may spend too much time on compare operations. We would be comparing a segment with x other segments that could already have been combined into one segment. This would mean that we could have saved $x-1$ compare operations if only we would have combined segments earlier.

We currently allocate pools of 20 segments. Every time the pool has been used up, we first try to

combine segments. If this frees at least one segment, we continue without allocating a new pool. Only if we cannot combine any segments, we allocate a new pool of 20 segments. This number 20 has been determined experimentally. Setting it higher lowers performance, setting it lower does not improve performance. We have measured a very negative influence on the performance if set too high.

Unfortunately we have noticed this influence on the performance very late, and we have no time left to implement and experiment with such strategies. Therefore we will make do with the reasonably good number of 20 segments and the current allocation strategy. We will recommend this as a topic for further work and suggest some improvements in Section 8.2.6.

7.4 Bitmap optimizations

In Section 3.3 we have presented our optimizations to the bitmaps in the datarace backend. In this section we show the results. The most optimized versions of the multi-level bitmaps used in the data race detector for administration of all memory accesses performed by the program have:

We compare the performance of a 3-level bitmap (like the original data race detector had, but optimized) with the new 9-level and 10-level bitmaps.

<i>Property</i>	<i>3-level</i>	<i>9-level</i>	<i>10-level</i>
Bytes per block	2048	32	32
LOAD/STORE combined	No	No	Yes
Cache	No	Yes	Yes
Pool allocation	No	Yes, 256KB per pool	Yes, 256KB per pool

Figure 7.2 shows the difference when we fit the data race detector with the different bitmaps.

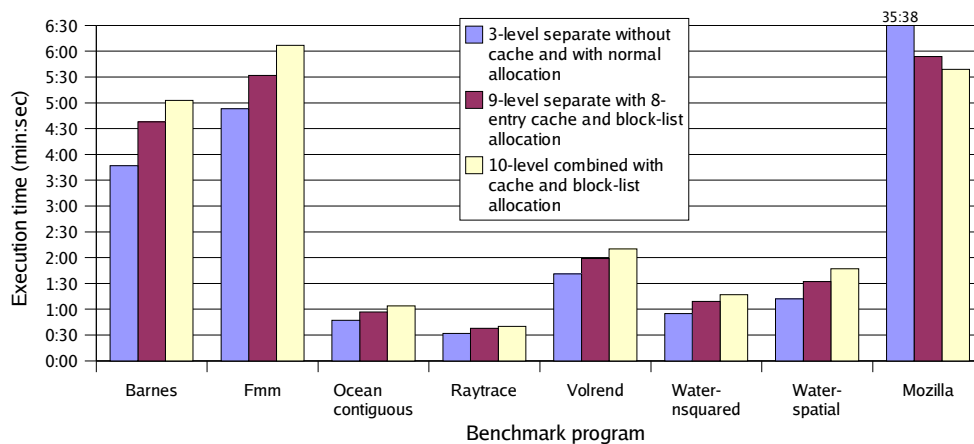


Figure 7.2: Performance of the 9-level and 10-level bitmap compared to that of the 3-level bitmap. SPLASH-2 programs executed with 2 threads.

The 3-level bitmap is the fastest for the SPLASH-2 benchmarks, but the 9-level bitmap is the fastest for Mozilla. This is explained as follows: the 3-level bitmap allows for faster registration of memory accesses because it has fewer levels of indirection, but the 9-level bitmap allows for faster comparison and combination of bitmaps. The 10-level bitmap may be slower than the 9-level bitmap because the 9-level bitmap has separate caches for LOAD accesses and for STORE accesses

and the 10-level bitmap does not. The results are for 2 threads. For more threads, comparison of segments will become more important, and may change the balance in favor of the 9-level bitmaps.

Segment combination and removal limits the maximum number of bitmaps in memory, and allows most of the SPLASH-2 benchmarks to run forever. The number of segments needed remains constant, because either all threads synchronize with each other at fixed points, or all threads except for the main thread synchronize with each other at fixed points and the main thread is inactive. In either case segments can be combined very well, because the active threads execute the same algorithm and each thread repeatedly performs a computation step and synchronizes with all other threads before the next computation step. This means that after each computation step, the segments of the completed computation step can be combined with those of the previous computation step. Other prerequisites for running indefinitely have also been fulfilled: the vector clocks have a sufficient width, because thread creation is not dynamic (a fixed number of threads is started once), and the combined bitmaps do not grow beyond a certain size, because no new memory is allocated by the benchmark program after the initialization phase. The allocated memory often consists of large blocks, and often each thread accesses a contiguous memory area. The 3-level bitmap with the big blocks is better suited for this.

Mozilla on the other hand does not have these nice properties. The number of uncombinable segments grows, and therefore the memory usage increases, and each new segment has to be compared to more and more other segments. Mozilla cannot run forever because eventually the data race detector will run out of memory for the bitmaps of new segments. But because the blocks of the 9-level and 10-level bitmap are smaller, they cost less memory and they can be compared at a lower cost than those of the 3-level bitmap. This more than makes up for the slower memory access registrations of the 9- and 10-level bitmaps.

The memory usages in Figure 7.3 also include the memory needed for the administration of segments and synchronization objects.

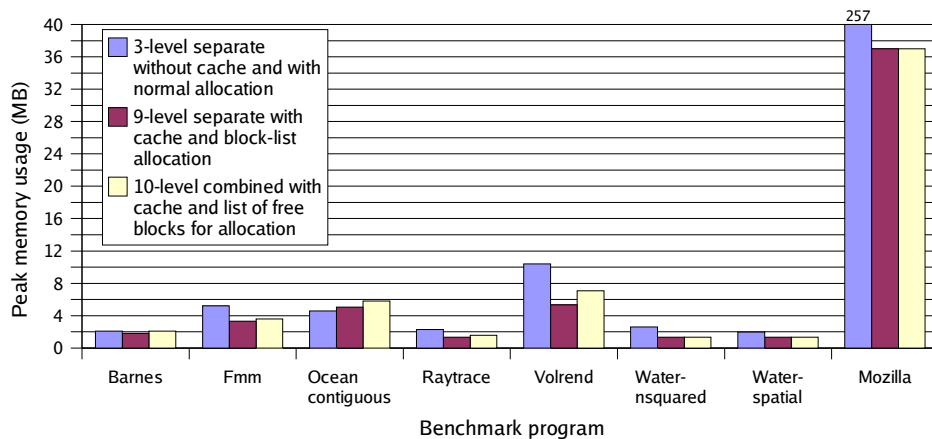


Figure 7.3: Memory allocation by the data race detector for segments, bitmaps and synchronization objects compared for the 3-level, 9-level and 10-level bitmap. SPLASH-2 programs executed with 2 threads.

The 9-level bitmap requires the least memory, and for the Mozilla benchmark this is very important. The 10-level bitmap has combined LOAD/STORE access at the lowest level, the 9-level and 3-level bitmaps do not. This means for example that if a large memory region is initialized with STORE operations, the 10-level bitmap needs twice as much memory at the lowest level (half of which is reserved, but unused, for LOAD accesses) compared to the 3-level and 9-level bitmap to register the STORE accesses.

We recommend the use of the 9-level bitmap in general, because it performs much better than the 3-level bitmap on the 'hard' applications for which the number of thread segments in memory keeps growing, and it does not perform much worse than the 3-level bitmap for the easy ones. If you know the thread segments of the application can be combined very well and the number of segments does not grow too much, you can use the 3-level bitmap for extra performance.

7.5 Memory access registration optimization by DIOTA

Figure 7.4 presents the result of the optimizations discussed in Chapter 6 used for data race detection. The columns represent:

1. instrumentation without memory access registration optimization (normal)
2. instrumentation with intra-node memory access registration optimization (Section 6.1)
3. instrumentation with intra-node memory access registration optimization and stack access elimination (Section 6.3)
4. instrumentation with inter-node memory access registration optimization (Section 6.2)
5. instrumentation with inter-node memory access registration optimization and stack access elimination
6. instrumentation with inter-node memory access registration optimization and loop unrolling (Section 6.4)
7. instrumentation with inter-node memory access registration optimization, loop unrolling and stack access elimination

For all tests we have used the fully optimized datarace detector backend, with a 3-level bitmap for the SPLASH-2 benchmarks and a 9-level bitmap for Mozilla.

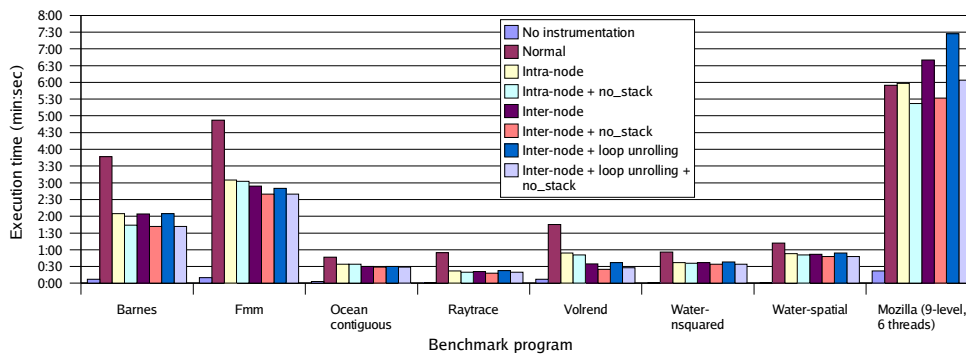


Figure 7.4: Performance results for optimizations by DIOTA. SPLASH-2 benchmarks executed with 2 threads and 3-level bitmap. Mozilla executed with 9-level bitmap.

For the SPLASH-2 benchmarks intra-node optimization is earned back quickly and improves performance. Inter-node optimization can in some cases perform significantly better. Mozilla is a big application that executes much more code than the relatively small SPLASH-2 benchmarks. This increases the initial analysis and optimization cost and this has not yet been earned back after rendering the requested web page. The test results prove that loop unrolling can improve upon inter-node optimization, but they show no significant improvement.

For elimination of stack access registrations (the 'no_stack' columns in the Figures) it is important to know if the benchmark programs share data on the stack or not, otherwise the results are misleading. Barnes and Fmm do not share data on the stack, so at least these results are correct performance improvements.

It is important to note that most of the benchmarks have an initialization phase, the execution period before the creation of the first thread. The effort put into the optimizations of the code for this phase is hard to earn back, because the code is often executed only once. See also our recommendation in Section 8.2.3. We have also collected results for longer executions of the same benchmarks to see if we relatively save more time in the long run. These results are presented in Figure 7.5. For the SPLASH-2 benchmarks we have done this by multiplying the number of time steps by four. We could not do this for Volrend and Raytrace, because these do not work with a fixed number of time steps. For Mozilla we have done this by specifying the web page www.google.be on the command line and when this page was rendered pressing Alt-Home to go to the home-page specified as www.elis.ugent.be, followed by Alt-F4 to close Mozilla.

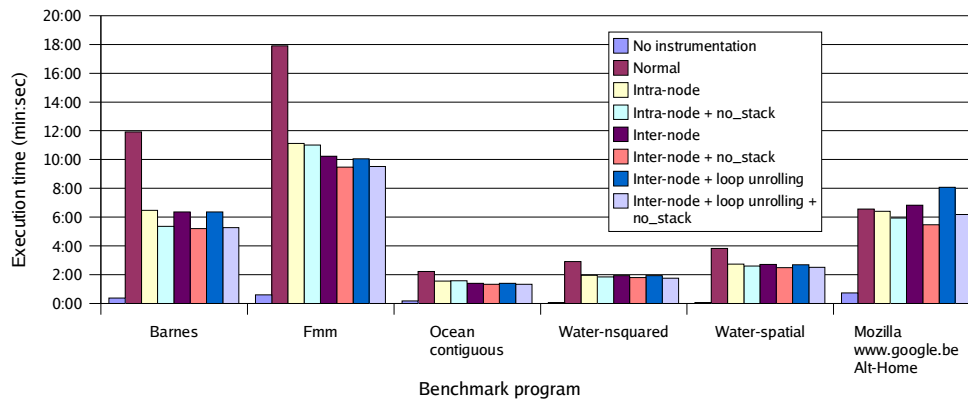


Figure 7.5: Performance results for optimizations by DIOTA. SPLASH-2 benchmarks executed with 2 threads and 3-level bitmap, with 4x time steps. Mozilla executed with 9-level bitmap and 2 web pages.

Relatively speaking, the time saved by optimizations is indeed higher for the results of Figure 7.5 than for the results of Figure 7.4, but only by about 2% for the SPLASH-2 benchmarks. For mozilla the relative gain is higher, about 9%.

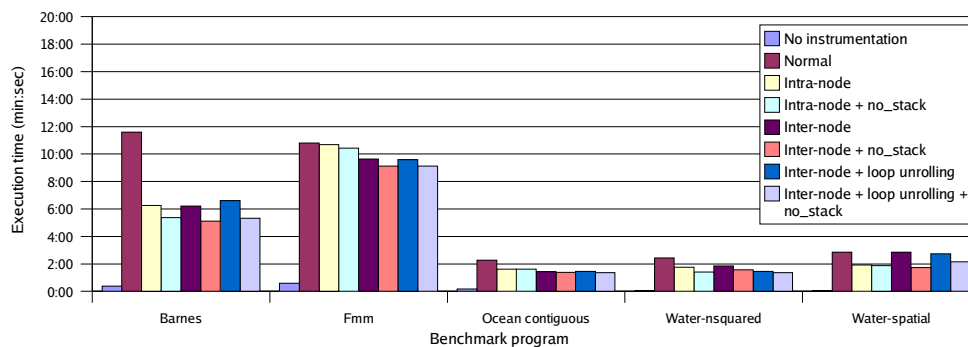


Figure 7.6: Performance results for optimizations by DIOTA. SPLASH-2 benchmarks executed with 4 threads and 3-level bitmap, with 4x time steps.

Figure 7.6 shows the results for four threads instead of two. Since the computer has only two CPUs, there is no additional speedup.

8 Conclusions and future work

In this report we have presented several methods to optimize data race detection for performance and necessarily also for memory requirements. In Section 8.1 we will summarize our accomplishments and draw our conclusions. Section 8.2 introduces some ideas for future work to improve the performance.

8.1 Conclusions

Data race detection with DIOTA and the data race detection backend `diota-dr` is detection at the machine code level. This means that all useful information that may be present in the source code of the computer program and that is available to the compiler has been lost. Thus many opportunities for optimization of data race detection that the compiler can do are impossible to exploit for DIOTA because at DIOTA's low level we can not tell if they are correct. It is therefore in general more efficient to use a higher level detector if available. Such a detector cannot always be used, `diota-dr` can therefore be useful, especially in the following cases:

- if there is no higher level detector for the programming language. `diota-dr` and DIOTA can be made very much programming language-independent, and this will improve once self-modifying code support and support for other thread libraries is implemented.
- if multiple languages are used for one binary program
- if the original source code is not available
- if a bug in the compiler or one of the libraries is suspected.

Even though we cannot optimize DIOTA and its data race detector backend as much as a higher level data race detector, we have managed to achieve significant improvements. Looking at the evaluation in the previous chapter, we have accomplished a significant performance improvement and reduced the memory requirements enormously, both by modifying the data race detector backend and by modifying DIOTA.

In the data race detector backend we have introduced segment combination and corrected segment removal to remove the low upper limit on the execution span of a program that could be analyzed. There are now important classes of programs that can run indefinitely and others that can run for a very long time. We have optimized segment allocation for speed, optimized vector clocks for speed, bugfixed and optimized synchronization object administration, and bugfixed and optimized the memory access registration storage by optimizing the bitmaps. Chapter 7 shows that the optimizations in the data race detector backend improve performance significantly.

In DIOTA we have introduced the more complex but much more powerful control flow analysis method that allows for all kinds of optimizations. Every thread now has its own separate instrumented code. This not only allows for optimization by inserting hard-coded thread-specific values in the instrumented code and the handlers, but also allows for other features such as the removal of instrumented code to free space for new instrumented code, and correct signal handling. The evaluation in Chapter 7 shows that this has improved performance

We have also introduced methods to eliminate and combine memory access registrations by analyzing register relations. The intra-node variant of this method has proven its worth in Section 7.5. Even though the initial analysis cost is higher for inter-node optimization, the extra benefits with respect to intra-node optimization can lead to a higher performance. The problem that limits the extra benefits of inter-node optimization the most is that we cannot optimize through call instructions without losing the correctness guarantee. Unlike intra-node optimization, inter-node

optimization can benefit from loop detection and loop optimizations. We have implemented loop detection. We have not exploited this very successfully yet, because our first loop optimization, loop unrolling, does not seem to improve performance, again because many loops contain call instructions (and the number of instruction cache misses may increase). We present an idea for another loop optimization based on the same loop detection algorithm in the next section.

All in all we are very satisfied with the results. Taking Mozilla as example, slowdown factors due to data race detection can still be high, but with our modifications and on a contemporary computer we can now test this browser by surfing the web with acceptable speed and without running out of memory for a long time.

8.2 Future work

In the subsections we present several ideas for further improvement of the performance of data race detection. Especially during the evaluation of our work in Chapter 7, we have noticed several opportunities for further improvement of the performance. The ones we have not been able to take advantage of before our deadline was reached are presented in the subsections.

8.2.1 Partial inlining of subroutines

We have encountered several limits that make it difficult to optimize instrumented code. The most serious limiting factor are call instructions. These are a big problem for several reasons:

- The halting problem. If the subroutine is not guaranteed to return, the registrations of the memory accesses preceding the call can not be combined with registrations of memory accesses after the call, because the latter accesses are not guaranteed to happen. Since the call puts the return address on the stack, and the data on the stack is volatile, the return address can be modified by a data race, which makes this problem unsolvable without sacrificing the correctness of the data race detection even for the simplest functions.
- If a synchronization operation can be executed by the subroutine (directly or indirectly), it is not correct to optimize 'through' the call. It is then not allowed to combine the registrations of the memory accesses preceding the call with the registrations of the memory accesses after the call. It is also not allowed to eliminate registrations of the memory accesses after the call based on the fact that they were registered before the call, because they may have to be registered in a different segment than those before the call. For simple subroutines it could be determined if they can perform a synchronization operation or not, but this is time-consuming. Another problem is that programs with self-modifying code can modify themselves in such a way that a subroutine includes a synchronization operation that it did not include before, making a seemingly correct optimization invalid after the self-modification.

A partial solution that allows for more optimization can be to include the initial part of each subroutine in the instrumented code of the caller (a different, limited version of inlining). This is not compatible with self-modifying code support, because then several instrumented versions of the same original program code would be created, and we would not be able to find all copies efficiently if we wanted to delete and/or replace them. It would also increase the size of the generated instrumented code enormously.

8.2.2 Moving memory access registrations out of loops

An idea for optimization that we have not explored yet is to take advantage of increments in loops. Often a loop increments (or decrements) an index variable (or pointer), for example named i , that is used to access an element i of an array. If a consecutive series of s elements of the array is accessed

by the loop in this manner, we may be able to register the entire series with s times the length of an element as one access outside the loop instead of performing s registrations of an access with the length of one element inside the loop. Our loop analysis can be used for this purpose:

1. Using the identified backedges to the loop header, we can see if the same increment of a register that is used to access memory inside the loop holds for all backedges (by comparing the end-states of the tail nodes of the backedges), and thus for the entire loop.
2. We can identify all exit points of the loop, after which the postponed registration should occur.

Some other things remain to be checked, for example if the accesses inside the loop are guaranteed to access a contiguous memory area (e.g., every array element and not every other array element). This can be done using the ANY and STORE tables in the end-state of the tail nodes of the backedges: if the entire region between the old value of the register and its new value is covered by accesses of the correct type, the registrations for these accesses can be postponed to after completion of the loop.

One side-effect of this method is that it will register accesses late, which may cause data races to be missed in case a signal or exception occurs after the accesses have been performed but before they have been registered.

8.2.3 Postponing memory access instrumentation

Most programs have an initialization phase during which there is only one thread. It is a waste of time to register all memory accesses during this phase, because they cannot be involved in data races. Currently, the memory access registration callback in the data race backend detects this and does not create a bitmap during this phase.

However, this can probably be improved. If we start instrumenting the program *without* memory access instrumentation, and then when we reach the first instance of `pthread_create()` remove all instrumented code and start re-instrumenting the code *with* memory access instrumentation, we may save a lot of time. Not only do we save a lot by not calling back the data race backend, but we also save a lot of analysis time by not analyzing the functions that are executed only once before the first instance of `pthread_create`. For example the browser mozilla has a long initialization phase before the first call to `pthread_create`, and it even locks and unlocks mutexes. Synchronization function calls such as `pthread_mutex_(un)lock` before the first `pthread_create` call need not be intercepted either.

It may even be possible to postpone the instrumentation of the program completely until `pthread_create` is executed for the first time, and to run the original code, by replacing the first bytes of the `pthread_create` function by a jump to DIOTA so that DIOTA can start instrumenting there.

8.2.4 Limiting the number of context switches for instrumentation

DIOTA now analyzes and instruments one function (or code fragment), then executes the instrumented code of that function. The instrumented code will probably call a subroutine that still has to be instrumented almost immediately, and thus returns control to DIOTA to instrument this subroutine. This costs many context switches between DIOTA and the instrumented code. We can save context switches by instrumenting subroutines that are certain to be called immediately without executing the instrumented code first.

8.2.5 Eliminating stack accesses dynamically

Section 6.3 describes a way to eliminate stack accesses during instrumentation. Another method to

eliminate stack access registrations is by determining the base address of the stack when the thread is started, and by testing at run-time if the memory address of an access is in between the base address and the current value of the stack pointer. If so, do not register the access. If a thread installs its own stack, which is not difficult to do, this test will lead to incorrect results, so it must be used with care.

8.2.6 Segment allocation, comparison and combination strategies

As we explained in Section 7.3, the segment allocation and combination strategies can have a high influence on performance.

To prevent worst-case scenarios and to obtain a more reliable high performance, it may be better to use an other allocation strategy: if our pool of segments has been used up, try and combine segments as we currently do. But after that, allocate additional segments as necessary to keep the size of the pool of free segments constant at 20 segments.

It may also be a good idea to give each thread its own segment pool, because each thread only combines its own segments. Currently the same thread may continuously try to combine its segments and fail, causing the allocation of additional segment pools even though segments of other threads may be combinable. If every thread has its own pool, every thread will be forced to combine its segments on a regular basis. It may also be advisable to build a mechanism to ensure that the segments of a thread that has become inactive are combined so that other threads can compare their segments with this one efficiently.

A completely different thought is that maybe threads should not compare their own new segments with segments of other threads themselves, but let one or more other threads do this. Since the thread is apparently active itself, it could better be executing its program code than comparing its segments. An other CPU than the CPU this thread is 'running' on may currently be doing nothing, wasting valuable time that could be used for segment comparison and combination. For example, Mozilla has one thread that does the most work. If we could take the comparison and combination work off its back and off the back of the CPU this thread is running on, we may be able to speed up the execution enormously. Too bad we did not think of this earlier, time's up.

8.2.7 Bitmap optimizations

The 3-level bitmap currently does not have optimized memory allocation, and does not use a cache. This may improve its speed, although it will probably not make as much difference as for the 9-level bitmap. The 10-level bitmap may be slower than the 9-level bitmap because the 9-level bitmap has separate caches for LOAD accesses and for STORE accesses and the 10-level bitmap does not. Therefore it may be possible to improve this for the 10-level bitmap. If this is possible, the cheaper segment comparisons when using the 10-level bitmap may tip the balance in favor of the 10-level bitmap instead of the 9-level bitmap.

References

- [ADV90] S.V. Adve and M.D. Hill, “Weak ordering – a new definition”. Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 2-14, May 1990.
- [ADV91] S.V. Adve, M.D. Hill, B.P. Miller, and R.H.B. Netzer, “Detecting data races on weak memory systems”. Proceedings of the 18th Annual International Symposium on Computer Architecture, pages 234-243, May 1991.
- [ADV92] S.V. Adve and M.D. Hill, “A unified formalization of four shared-memory models”. Technical report, University of Wisconsin, September 1992.
- [AHO86] Alfred V. Aho, Ravi Seti and Jeffrey D. Ullmann, “Compilers: Principles, Techniques and Tools”. Addison-Wesley Publishing Co.- Reading, Mass. ISBN: 0-201-10088-6.
- [AUD94] K. Audenaert and L. Levrouw, “Interrupt replay: A debugging method for parallel programs with interrupts”. *Microprocess. Microsyst.* 18, 10, pages 601-612, 1994.
- [AUD95] K. Audenaert and L. Levrouw, “Space efficient data race detection for parallel programs with series-parallel task graphs”. Proceedings of the third Euromicro Workshop on Parallel and Distributed Processing, pages 508 – 515, 1995.
- [BAL89] V. Balasundaram and K. Kennedy, “Compile-time detection of race conditions in a parallel program”. Proceedings of the 3rd International Conference on Supercomputing, pages 175-185, June 1989.
- [BER92] A. Beranek, “Data race detection based on execution replay for parallel applications”. Proceedings of Conference on Parallel Processing, pages 109-114, September 1992.
- [BOS97] K. De Bosschere and M. Ronsse, “Clock Snooping and its application in on-the-fly data race detection”. Proceedings of the 1997 IEEE International Symposium on Parallel Algorithms and Networks, pages 324-340, December 1997.
- [CHE98] G. Cheng, M. Feng, C.E. Leiserson, K.H. Randall, and A.F. Stark, “Detecting data races in Cilk programs that use locks”. Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, pages 298-309, June 1998.
- [CHO91] J.-D. Choi and S.L. Min, “Race frontier: Reproducing data races in parallel-program debugging”. Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming, pages 145-154, July 1991.
- [CHO98] J.-D. Choi and H. Srinivasan, “Deterministic replay of java multithreaded applications”. Proceedings of the 2nd ACM Symposium on Parallel and Distributed Tools, pages 48-59, August 1998.
- [COO01] Keith D. Cooper, Timothy J. Harvey and Ken Kennedy (Rice University, Houston, Texas), “A Simple, Fast Dominance Algorithm”. *Software – Practice and Experience* 2001, Vol. 4, Pages 1-10, John Wiley & Sons, Ltd.
- [DIN90] A. Dinning and E. Schonberg, “An empirical comparison of monitoring algorithms for access anomaly detection”. Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 1-10, March 1990.
- [DIN91] A. Dinning and E. Schonberg, “Detecting access anomalies in programs with critical sections”. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in *ACM SIGPLAN Notices*, 26(12), pages 85-96, December 1991.

- [DRE03] U. Drepper and I. Molnar, “The native POSIX thread library for Linux”. <http://people.redhat.com/drepper/nptl-design.pdf>, January 2003.
- [EMR88] P. Emrath and D. Padua, “Automatic detection of nondeterminacy in parallel programs”. Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pages 89-99, May 1988.
- [FID98] C. A. Fidge, “Partial order for parallel debugging”. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, pages 183-194, 1988.
- [FLA01] C. Flanagan and S. Freund, “Detecting race conditions in large programs”. Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), pages 90-96, June 2001.
- [HOO90] R. Hood, K. Kennedy, and J. Mellor-Crummey, “Parallel program debugging with on-the-fly anomaly detection”. Proceedings of the 1990 Conference on Supercomputing, November 1990.
- [INT02] Intel Corporation, “IA-32 Intel Architecture Software Developer's Manual”. Available from Intel Corporation, P.O. Box 7641, Mt. Prospect II, 60056-7641, and from <http://www.intel.com>.
- [ITZ99] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordechai, “Towards integration of data race detection in DSM systems”. Journal of Parallel and Distributed Computing, 59(2), pages 180-203, November 1999.
- [LER02] Xavier Leroy, The LinuxThreads library, POSIX 1003.1c kernel threads for Linux. <http://pauillac.inria.fr/~xleroy/linuxthreads/>, 1996.
- [MAE02] J. Maebe, M. Ronsse and K. De Bosschere, “Dynamic Instrumentation, Optimization and Transformation of Applications”. Compendium of Workshops and Tutorials Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques. September, 2002. Charlottesville, Va. Eds. Charney, M.; Kaeli, D. DIOTA website: <http://www.elis.ugent.be/~ronsse/diota/>.
- [MEL91] J. Mellor-Crummey, “On-the-fly detection of data races for programs with nested fork-join parallelism”. Supercomputer Debugging Workshop, pages 1-16, Nov. 1991.
- [MEL93] J. Mellor-Crummey, “Compile-time support for efficient data race detection in shared-memory parallel programs”. ACM/ONR Workshop on Parallel and Distributed Debugging, pages 129-139, May 1993.
- [MOZ02] Mozilla 1.1 Beta, Released 22 July 2002, (Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.1b) Gecko/20020722), Copyright © 1998-2002 by Contributors to the Mozilla codebase under the Mozilla Public License and Netscape Public License. Available from <http://www.mozilla.org/releases/>.
- [NET90a] R.H.B. Netzer and B.P. Miller, “On the complexity of event ordering for shared-memory parallel program executions”. 1990 International Conference on Parallel Processing, 2, pages 93-97, January 1990.
- [NET90b] R.H.B. Netzer and B.P. Miller, “Detecting data races in parallel program executions”. Technical report, University of Wisconsin, August 1990.
- [NET91a] R.H.B. Netzer and B. P. Miller, “Improving the accuracy of data race detection”. Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming, pages 133-144, April 1991.

- [NET91b] R.H.B. Netzer, “Race Condition Detection for Debugging Shared-Memory Parallel Programs”. PhD thesis, University of Wisconsin, 1991.
- [NET92a] R.H.B. Netzer and B.P. Miller, “What are race conditions? Some issues and formalizations”. *ACM Letters on Programming Languages and Systems*, 1, pages 74-88, March 1992.
- [NET92b] R.H.B. Netzer and S. Ghosh, “Efficient race condition detection for shared-memory programs with post/wait synchronization”. *International Conference on Parallel Processing*, St. Charles, IL, August 1992.
- [PER96] D. Perkovic and P. Keleher, “Online data-race detection via coherency guarantees”. *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 47-57, October 1996.
- [PER00] D. Perkovic and P. Keleher, “A protocol-centric approach to on-the-fly race detection”. *IEEE Transactions on Parallel and Distributed Systems*, 11(10), pages 1058-1072, October 2000.
- [POZ03] Eli Pozniansky and Assaf Schuster, “MultiRace: Efficient On-The-Fly Detection of Data-Races in C++ Programs”. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003, San Diego.
- [PRA01] C. von Praun and T. R. Gross, “Object race detection”. In *ACM Conference on Object-Oriented Programming Systems Languages, and Applications*, Oct. 2001.
- [RIC98] B. Richards and J. R. Larus, “Protocol-based data-race detection”. In *Second SIGETRICS Symposium on Parallel and Distributed Tools*, August 1998.
- [RON95] M. Ronsse, L. Levrouw and K. Bastiaens, “Efficient coding of execution-traces of parallel programs”. *Proceedings of the ProRISC/IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, 251-258, March 1995.
- [RON99] M. Ronsse and K. De Bosschere, “RecPlay: A Fully Integrated Practical Record/Replay System”. *ACM Transactions on Computer Systems*, Vol. 17, No. 2, pages 133-152, May 1999.
- [RON03] M. Ronsse, B. Stougie, J. Maebe and K. De Bosschere, “An efficient data race detector backend for DIOTA”, Parco 2003.
- [SAV97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs”. *ACM Transactions on Computer Systems*, 15(4), pages 391-411, October 1997.
- [WOO95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations”. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24-36, Santa Margherita Ligure, Italy, June 1995. Website: <http://www-flash.stanford.edu/apps/SPLASH>.

Appendix A: format of a decoded instruction

The general format for an IA-32 instruction is:

- up to 4 prefixes (optional)
- 1, 2 or 3 opcode bytes
- a ModR/M byte (if required by the opcode)
- a SIB byte (if required by the ModR/M byte)
- a displacement (optional, 1, 2, or 4 bytes)
- an immediate value (optional, 1, 2 or 4 bytes) or AMD 3DNow! suffix byte.

The instruction length thus varies from 1 to 17 bytes. Figure A.1 shows the data structure we use to decode IA-32 instructions. It basically follows the above instruction format. The `access_*` and `index_*` fields are needed to decode the ModR/M and SIB bytes. The `changed_register_set` field is a bitset with one bit for each general purpose register that is set if after the execution of the instruction the contents of that register may have been modified. The `flags` field contains handy information for DIOTA about the general type of the decoded instruction and the presence and properties of the ModR/M and SIB bytes.

```
struct s_t_decoded_instruction
{
    unsigned char  flags;           // see FLAG_* definitions below
    unsigned char  prefix[4];      // see PREFIX_* definitions below
    unsigned char  opcode[3];      // 1st,2nd,3rd byte at index 0,1,2. See opcode_length for #bytes.
                                    // if flags & FLAG_Opcode see also the Reg/Opcode field of ModRM
                                    // for opcode 0x0F 0x0F, see immediate field for 3DNow! suffix

    unsigned char  ModRM;          // valid iff (flags & FLAG_ModRM), even if value == 0, else 0.
    unsigned char  SIB;           // valid iff (flags & FLAG_SIB), even if value == 0, else 0.
    signed   int   displacement;   // value 0 if not present. see displacement_size for presence.
    signed   int   immediate;      // value 0 if not present. see immediate_size field for presence.

    unsigned char  num_prefixes;   // number of prefixes (0..4)
    unsigned char  opcode_length;  // length of opcode in bytes
    unsigned char  displacement_size; // in bytes (0 if no displacement present)
    unsigned char  immediate_size; // in bytes (0 if no immediate present)
    unsigned char  length;         // of total instruction, including prefixes, in bytes

    // indexes for access_* fields:
    // ModR/M, SIB : always index 0, maybe in combination with index register (see index_scale)
    // MOVS/STOS/... : ESI (if applicable) always index 0, no index register (index_scale = 0)
    // MOVS/STOS/... : EDI (if applicable) always index 1
    // PUSH*/POP* : ESP always index 1 (ESP may also be used for addressing for PUSH mem)
    // index_scale and index_register associate with index 0 only.
    unsigned char  access_type[2]; // see ACC_* definitions below
    unsigned char  access_size[2]; // in bytes (0 if access_type[*] == ACC_N)
    unsigned char  access_register[2]; // see REG_* definitions
    unsigned char  index_scale;      // 1/2/4/8, or 0 = no index register
    unsigned char  index_register;   // see REG_* definitions below

    // a change to AX/AH/AL means a change to EAX. Analogous for EBX, ECX, EDX!!!
    unsigned char  changed_register_set; // see MASK_* definitions below
};
```

Figure A.1: The data structure for decoded instructions.

Appendix B: m4 macros used for the SPLASH-2 benchmark set

```
divert(-1)
define(NEWPROC,) dnl

define(BARRIER, `{
  pthread_mutex_lock(&(($1).bar_mutex));
  ($1).bar_teller++;
  if (($1).bar_teller == ($2)) {
    ($1).bar_teller = 0;
    pthread_cond_broadcast(&(($1).bar_cond));
  } else
    pthread_cond_wait(&(($1).bar_cond), &(($1).bar_mutex));
  pthread_mutex_unlock(&(($1).bar_mutex));
}')

define(BARDEC, `{
  struct { pthread_mutex_t bar_mutex; pthread_cond_t bar_cond; unsigned bar_teller; } $1;
}')

define(BARINIT, `{
  pthread_mutex_init(&(($1).bar_mutex), NULL);
  pthread_cond_init(&(($1).bar_cond), NULL);
  ($1).bar_teller=0;
}')

define(LOCKDEC, `pthread_mutex_t $1;')
define(LOCKINIT, `{pthread_mutex_init(&($1),NULL);}')
define(LOCK, `{pthread_mutex_lock(&($1));}')
define(UNLOCK, `{pthread_mutex_unlock(&($1));}')

define(ALOCKDEC, `pthread_mutex_t ($1)[$2];')
define(ALOCKINIT, `{ int i; for(i = 0; i < ($2); i++) pthread_mutex_init(&(($1)[i]), NULL); }')
define(ALOCK, `{pthread_mutex_lock(&(($1)[($2)]);}')
define(AUNLOCK, `{pthread_mutex_unlock(&(($1)[($2)]);}')

define(PAUSEDEC, `sem_t $1;')
define(PAUSEINIT, `{sem_init(&($1),0,0);}')
define(CLEARPAUSE, `{;}')
define(SETPAUSE, `{sem_post(&($1));}')
define(WAITPAUSE, `{sem_wait(&($1));}')

define(WAIT_FOR_END, `{int aantal=$1; while (aantal--) pthread_join(__tid__[aantal], NULL);}')

define(CREATE, `{
  pthread_mutex_lock(&__intern__);
  assert(__threads__ < __MAX_THREADS__);
  pthread_create(&(__tid__[__threads__++]), NULL, (void*)(*)(void *)($1), NULL);
  printf("==> created thread %d\n", __threads__-1);
  pthread_mutex_unlock(&__intern__);
}')
define(MAIN_INITENV, `{__tid__[__threads__++] = pthread_self();}')
define(MAIN_END, `{exit(0);}')

define(INCLUDES, `{
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
#include <assert.h>
#define PAGE_SIZE 4096
#define __MAX_THREADS__ 256
}')

define(MAIN_ENV, `{
INCLUDES
pthread_t __tid__[__MAX_THREADS__];
unsigned __threads__=0;
pthread_mutex_t __intern__;
void *our_malloc(size_t size, char * file, unsigned line) { return malloc(size); }
}')

define(EXTERN_ENV, `{
INCLUDES
extern pthread_t __tid__[__MAX_THREADS__];
extern unsigned __threads__;
extern pthread_mutex_t __intern__;
void *our_malloc(size_t size, char * file, unsigned line);
}')

define(G_MALLOC, `our_malloc($1, __FILE__, __LINE__)')
define(NU_MALLOC, `our_malloc($1, __FILE__, __LINE__);')

define(CLOCK, `{long time(); ($1) = time(0);}')
divert(0)
```

Figure B.1: The m4 macros used for the SPLASH-2 benchmarks in Chapter 7