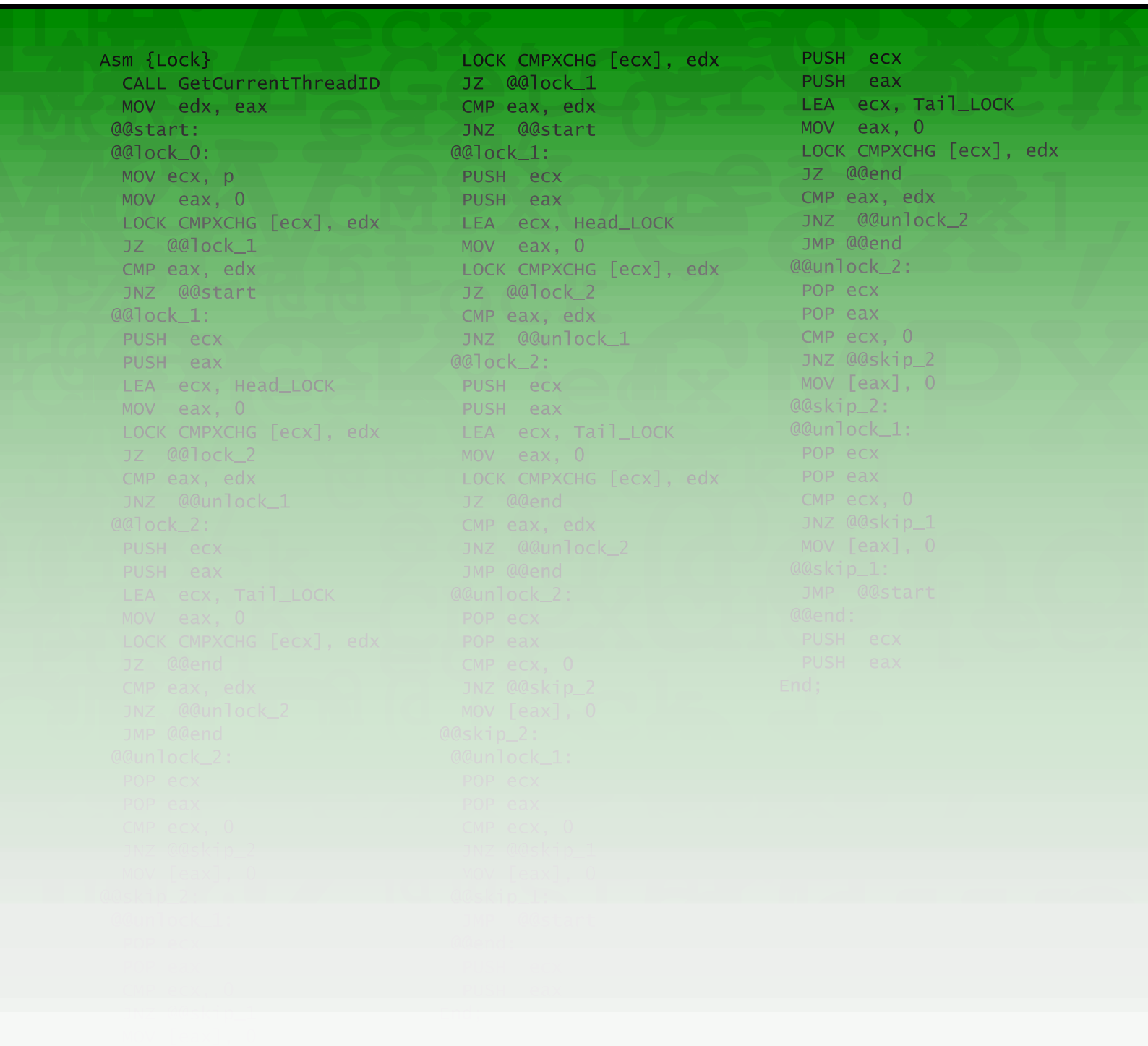
Avoiding Race Conditions With Micro Transactions



```
Asm {Lock}
  CALL GetCurrentThreadID
  MOV  edx, eax
 @@start:
 @@lock_0:
  MOV ecx, p
  MOV  eax, 0
  LOCK CMPXCHG [ecx], edx
  JZ  @@lock_1
  CMP eax, edx
  JNZ  @@start
 @@lock_1:
  PUSH  ecx
  PUSH  eax
  LEA  ecx, Head_LOCK
  MOV  eax, 0
  LOCK CMPXCHG [ecx], edx
  JZ  @@lock_2
  CMP eax, edx
  JNZ  @@unlock_1
 @@lock_2:
  PUSH  ecx
  PUSH  eax
  LEA  ecx, Tail_LOCK
  MOV  eax, 0
  LOCK CMPXCHG [ecx], edx
  JZ  @@end
  CMP eax, edx
  JNZ  @@unlock_2
  JMP @@end
 @@unlock_2:
  POP ecx
  POP eax
  CMP ecx, 0
  JNZ @@skip_2
  MOV [eax], 0
 @@skip_2:
 @@unlock_1:
  POP ecx
  POP eax
  CMP ecx, 0
  JNZ @@skip_1
  MOV [eax], 0
 @@skip_1:
  JMP  @@start
 @@end:
  PUSH  ecx
  PUSH  eax
End;
```

Jesper Bødker Christensen

AALBORG UNIVERSITET

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
Telephone: (45) 9635 8080
Telefax: (45) 9635 9798
http://www.cs.aau.dk

Title:
Avoiding Race Conditions with Micro Transactions

Topic:
Programming Technology

Project period:
DAT6, spring 2008
February 2008 - June 2008

Project group:
d621a

Members of the group:
Jesper Bødker Christensen

Supervisor:
Bent Thomsen

Number of copies: 4

Number of pages: 74 (64)

Abstract:

This report proposes a solution to race conditions that is a variation of locking and transactional memory. We introduce a small language that uses implicit statement level locking to make micro transactions. This yields new semantics for the assignment statement in the programming language. The locks are implemented using busy-waiting and an atomic compare-and-swap instruction. A simple flow-sensitive and field-sensitive points-to analysis is introduced to deduce which locks are required to perform the transactions atomically. Some experiments are performed which show that the performance of micro transactions is comparatively acceptable. However, it is concluded that, due to lack of compositionality, micro transactions are not a practical solution to race conditions in modern languages.

Preface

This report is my master thesis in programming technology. My previous work includes co-authoring a report on concurrency models where we explored the differences and similarities between the thread model and the process model. My master thesis can be viewed as a continuation of this work.

The previous report, 'Concurrency Models - Processes as an Alternative to Threads' [14], was predominantly a theoretical work. This report is based on a more experimental practical hands-on approach. Much time was spent on implementing and experimenting with a compiler to try different approaches. Emphasis has not been on developing formal semantics or an elaborate type system but, rather, on experimenting with a programming language and the compiler for this language.

The report is primarily intended for people with theoretical knowledge of programming languages, language theory and concurrent programming. In order to fully benefit from this report some theoretical knowledge on these subjects is required, prior to reading this report.

Jesper Bødker Christensen

Contents

Chapter 1

Introduction

In a few years the performance of the processors in our computers will no longer be measured in MHz. We will also have take the number of cores, synchronization measures and memory architecture into account. Limitations in size and heat will prevent further increase in clock frequency and promote the use of multiple cores in processors even more than today.

This means that the free lunch is over [15]. No more will our programs run faster with each new generation of processors. To utilize the new multicore processors we will have to change the way we design and write programs.

There have been many attempts to solve this problem in the last five decades by providing new paradigms, languages and libraries but none have proved to be 'The next big thing'. Semaphore, monitor, mutex, message passing, functional languages, process oriented languages, transactional memory... the list goes on. This report does not promise a solution to all the challenges associated with multicore processors. But it might serve as a small step towards a solution.

In my previous work, 'Concurrency Models - Processes as an Alternative to Threads' [14], we analyzed the dominant concurrency model today, the thread model, and another concurrency model, the process model. We assessed the compositionality potential of the two concurrency models and used abstract computer architectures to compare the two models on different hardware, in anticipation of distributed memory architectures in future processors. We showed that the two models are equally expressive and that they have similar deficiencies. We concluded that neither model is adequate and that new abstractions are needed to provide better concurrency mechanisms in future programming languages.

In this report we are only concerned with shared memory architectures. This memory architecture is simple to understand and use. And even though there are signs that suggest that the industry is becoming increasingly interested in distributed memory architectures we will not consider those in this report. For the purpose of this report a shared memory architecture gives a simple foundation to work from.

1.1 Locking

Semaphores, monitors and mutexes are very effective ways of ensuring mutual exclusion and preventing race conditions. Unfortunately these mechanisms require the programmer to enforce correct locking protocols and are therefore prone to errors.

A typical pitfall is forgetting to lock or unlock at the correct points in the program. A related problem with this approach is deadlocking caused by incorrect or conflicting locking protocols. Although this approach is error-prone it is the de facto way of ensuring mutual exclusion today.

Locking is even more problematic in the object oriented paradigm. The inheritance anomaly is a term used to describe the problem that :. synchronization code cannot be effectively inherited without non-trivial class re-definitions" [9]. This problem arises when a subclass is unable to properly honor or follow the locking protocols of a superclass - possibly resulting in a deadlock.

1.2 Transactional Memory

Transactional memory is a more sophisticated method of avoiding race conditions and ensuring consistency. This approach performs updates in shared memory by executing a series of read and writes as a single *atomic* operation. This gives a guarantee to the programmer that either all of the updates in the transaction were successfully executed or that none of the updates were executed. Software transactional memory is a method that does not require any special hardware while hardware transactional memory is based on hardware support. Examples of these are [8] and [11], respectively. Hybrid models that combine the use of hardware and software also exist.

If the transaction conflicts with other transactions then the transaction is aborted and the changes are rolled back. The transaction may be subsequently retried under the assumption that it no longer conflicts with other transactions. This enforces *consistency* in the program by always leaving the data in a consistent state.

The intermediate states of a transaction are not visible to other parts of the program - either all of the updates in the transaction are observable or none at all. This property of a transaction is called *isolation*.

Once a transaction has completed the changes made in the transaction are committed. A transaction which has completed and been committed will not be rolled back or undone at a later time - this is called *durability*.

The four properties of transactions described above, Atomicity, Consistency,

Isolation and Durability, are abbreviated ACID. The ACID properties, along with the transaction concept, are analogous to those in database systems. These properties give a programmer a very powerful programming model where much of the error handling and synchronization is automatically performed by the compiler or a runtime system.

Transactions can be implemented using locks but are often implemented by lock-free mechanisms. Logs are used to keep a record of the changes made during a transaction to detect conflicts and to roll back a transaction in case of a conflict. This means that although transactions are optimistic - making changes 'in-place' - there is an overhead associated with maintaining logs. Some early experiments have shown a slowdown of a factor of two as a result of this overhead [3].

A significant challenge in transactional memory is handling operations that cannot be rolled back. Almost any I/O operation is difficult or impossible to undo and is therefore problematic inside transactions.

Many of the proposed transactional memory frameworks require that the programmer identifies where transactions are needed - which statements that must be within a transaction and which statements that can be safely executed outside of a transaction.

Transactional memory is a comparatively new mechanism and much research is still being done on the subject. This means that solutions to the challenges in transactional memory may be available in the foreseeable future and that the overhead from logging may be reduced to an insignificant factor - making transactional memory a viable and feasible mechanism.

1.3 Micro Transactions

The approach described in this report, micro transactions, is a compromise between traditional locking and transactional memory. Micro transactions are based on compiler-generated locks and locking. No logs are used and transactions are never rolled back. Static analysis is used to determine what, when and where to lock.

A micro transaction consists of a sequence of one or more assignment statements. In addition to assignment statements, a transaction may be enclosed in an if-then-else like conditional statement. There is at most one branching and no iteration within a micro transaction. No function calls are allowed within micro transactions. These restrictions prevent deadlocking and the use of I/O operations.

Micro transactions guarantee atomicity, consistency and durability. They do not, however, provide complete isolation. Because only assignments are protected

by locks it is possible to observe inconsistent states in a program. This is a design choice which is discussed further in Chapter 3.

The main design principle behind micro transactions is simplicity. Limiting transactions to assignments only means that locks are held only for short periods of time and that a simple static analysis can be used to ascertain locking protocols. All locks are acquired before a transaction is executed - ensuring that the transaction can run to completion. All locks are released after the transaction has completed so other transactions may proceed. Deadlocks are prevented because all locks are acquired before starting a transaction. The locking order is determined by static analysis to avoid livelocks.

If a lock is already acquired by another transaction then busy waiting is used to retry. Busy waiting is preferable to a context switch because locks are presumed to be held only for a very short period of time.

1.4 Thesis

Before writing this master thesis the author and Simon H. Thøgersen wrote the report 'Concurrency Models - Processes as an Alternative to Threads' [14]. This report looked at the differences and similarities in the thread model and the process model. Among the conclusions in this report was the need for better exclusive access mechanisms. Race conditions was identified as the primary problem with existing mechanisms. This master thesis explores a way to avoid race conditions using automatic locking.

The thesis that is subject of this report is:

> *A compiler can, through static analysis, generate locks and the necessary locking protocols to ensure mutual exclusion in micro transactions. Micro transactions can make concurrent programming easier and less error-prone. Race conditions can be avoided using micro transactions. Micro transactions provide a programming model that is easy to comprehend and use.*

1.5 Summary

In this chapter an overview of micro transactions as well as an overview of related mechanisms was presented. In the following chapters micro transactions will be explained in greater detail. An experimental language, Vex, that implements micro transactions will be introduced and the relevant parts of the Vex compiler will be presented. The results of the experiments performed with Vex will be

discussed. Finally, micro transactions are compared with related work and the report ends with a conclusion.

The Vex Language

A simple verbose experimental language, Vex, was constructed to experiment with micro transactions. Vex is a concurrent language with language constructs for declaring and spawning threads.

The language is derived from Pascal, but Vex uses some constructs from Delphi. Delphi is derived from Pascal and is a superset hereof. The syntax and semantics of some statements are different from those in Pascal and a few new statements have been added. These differences and additions are explained in this chapter. Figure 2.1 shows an abstract overview of how the languages are related.

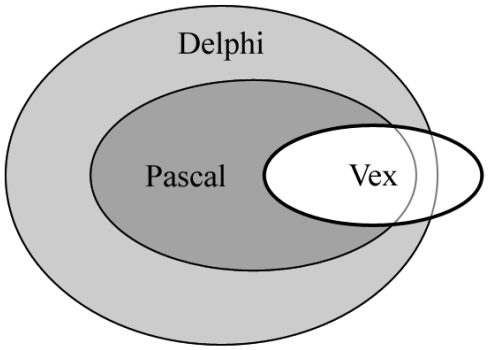


Figure 2.1: *The Vex language is derived from Pascal but Vex also uses some of Delphi's language constructs. Additionally, Vex has language constructs which are not found in Delphi or Pascal.*

The Vex compiler translates Vex programs into a Delphi unit. The Delphi unit is used by a runtime host program. This host program is compiled into an binary executable using the Delphi 2007 command line compiler. The compiling process is explained in greater detail in Chapter 4.

A short concurrent Vex program is listed in Figure 2.2 to give the reader an idea of what a Vex program looks like.

```
Program TestProgram;
Var
  Account1, Account2 : Integer;

Type
  Test = Thread ( Count : Integer )
  Var
    Amount, I : Integer;
  Begin
    For I := 1 to Count do
    Begin
      Amount := Random( 100 ) - 50;
      Account1 := Account1 + Amount & Account2 := Account2 - Amount;
    End;
  End;

Var
  I : Integer;
Begin
  Account1 := 1000 & Account2 := 1000;
  For I := 1 to 10 do
    Spawn Test( 1000000 );
  Join;
  Print( IntToStr( Account1 + Account2 ) );
end.
```

Figure 2.2: *A simple Vex program that starts 10 threads and waits for all threads to terminate. Each thread transfers an amount from one account to another 1,000,000 times. The printed result is always 2,000.*

2.1 Introduction to Delphi

Since Vex is related to Delphi a short introduction is in order. Delphi is based on Pascal but it contains many modern language features such as classes, objects, exceptions and support for Microsoft Windows GUI components.

The syntax in Delphi is a superset of Pascal with additional syntax for declaring classes, objects, forms and other added features. Delphi also supports the use of modules which are called units in Delphi. A Delphi program has one `program` file which may use any number of `unit` files.

Unlike C and C++, Delphi does not have a separate header file. A Delphi source file is divided into several sections, where the `interface` section corresponds to a `.h` file and the `implementation` section corresponds to a `.c` file.

Delphi does not have garbage collection which means that it is the responsibility of the programmer to allocate and deallocate memory.

2.2 Reused Syntax

Much of the syntax in Vex is identical to the corresponding counterparts in Pascal. Vex can be considered a modified subset of Pascal. Comments, scope levels, types, expressions, operator precedence and coercion in Vex is identical to that of Pascal. However, Vex only has support for a limited number of types, namely: Boolean, Integer, Char, String and Float.

Variable, procedure and function declarations are identical to those of Pascal. However, Vex only allows by-value parameters and does not allow by-reference parameters. Call statements, block statements (begin-end), for, repeat-until and while statements are identical to those in Pascal. The semantics for these syntactical elements described are identical to the semantics for the corresponding counterparts in Delphi.

Type declarations are identical to those in Pascal except for the declaration of threads. Threads are not a part of the Delphi language but are used through classes and libraries. Vex has language constructs for thread support. The declaration of a thread is elaborated in Section 2.3.2.

The scope rules in Vex follow the standard Pascal and Delphi convention of using a nested block structure with static binding and typing as defined in [6]. Thread declarations are treated in the same way as procedure declarations with regards to scope.

The complete syntax definition in Backus-Naur Form (BNF) is available in Appendix A.

2.3 Modified Syntax

There are four modifications to the grammar: the assignment statement, the type declaration of threads, the if-then-else statement and a modified version of the if-then-else statement. These modifications are described in the following subsections.

2.3.1 Assignment Statement

The assignment statement is the most important modification. An assignment can have one of four different structures: A general assignment, a pointer assignment, an exchange assignment or an allocation assignment.

Moreover, several assignments can be joined together in one statement. This feature is an essential part of micro transactions. The updating operations in a micro transaction are, in fact, a sequence of joined assignments.

Any assignment statement is atomic with regards to other assignments. Any variable in an assignment statement which might be referenced from another scope is called a shared variable. Shared variables are locked during the execution of the assignment statement s.t. other assignments using that variable must wait until the assignment has completed.

Assignments using function calls may not contain shared variables. This means that an assignment can contain either function calls *or* shared variables. This is explained in greater detail in chapter 3.

The variations of the assignment syntax are explained in the following sections.

General Assignment

The general assignment is defined by the syntax:

```
<Assignment> ::=
    <Qualified identifier> ':=' <Expression>
```

This is the typical assignment which allows for statements such as:

```
A := 123 + Round(77.54);
B := 'Hello' + ' ' + 'World!';
C := 345.5 - (0.125 + C);
D := not D;
```

The syntax of the general assignment is similar to the assignment statement in Pascal. Functions may be called in a general assignment - but only when no shared variables are used in the assignment. This restriction is explained in Chapter 3.

Pointer Assignment

Pointer variables may be assigned the value of other pointer variables in the general assignment. When assigning an address of a variable to a pointer the following syntax is used:

```
<Assignment> ::=
    <Qualified identifier> ':=' '@' <Qualified identifier>
```

An example of this follows here:

```
RecPtr := @Rec;
RecPtr := @Rec.Next;
RecPtr := @Rec.Prev;
RecPtr := @Nil;
```

The last assignment assigns a special value, Nil, to a pointer variable, `RecPtr`. The Nil value carries a special meaning: A pointer variable that points to Nil is explicitly unassigned - pointing to address 0 in memory. In Pascal and Delphi the value of pointers are undefined after declaration, until being assigned a value for the first time. When the value of a pointer is undefined it can point to any memory address and, as such, it cannot be tested for the Nil value.

In Vex pointers are initialized to Nil after declaration by the compiler. This makes it possible to test if pointers are unassigned by testing whether the pointer is equal to Nil or not. *Note that this does not remove the risk of dangling pointers!*

Exchange Assignment

The third assignment syntax, the exchange, was added to allow a shorthand notation for a common programming pattern: Swapping the contents of two variables. This assignment is defined by the following syntax:

```
<Assignment> ::=
    <Qualified identifier> '<=>' <Qualified identifier>
```

This syntax allows for the following statements:

```
A <=> B;
B <=> C;
RecPtr1 <=> RecPtr2;
RecPtr^.Next <=> RecPtr1^.Prev;
```

Note that the '^' symbol is used to dereference a pointer (as in Delphi and Pascal).

Allocation Assignment

The allocation assignment is used to assign a pointer to a new dynamically allocated record. The `new` keyword must be followed by a type identifier. A typed pointer to the newly allocated record is constructed and assigned to the identifier on the left hand side of the assignment. This assignment is defined by the following syntax:

```
<Assignment> ::=
    <Qualified identifier> ':=' 'new' <Type>
```

This syntax allows for the following statements:

```
RecPtr1 := new MyRecord;
RecPtr^.Next := new MyRecord;
```

This assignment syntax is particularly useful in a sequence of assignments. An example of this is shown in the following section.

A Sequence of Assignments

The assignment statement is defined as:

```
<Assign stm> ::=
    <Assignment> | <Assignment> '&' <Assign stm>
```

This means that any of the previous four assignment structures can be used interchangeably in sequence, constituting a single statement. Examples of this used is shown below:

```
A := (100 + B) / C &
B := B + 1 &
C <=> D;

Tail^.Next := New Element &
Tail^.Next^.Next := @Nil &
Tail^.Next^.Prev := Tail &
Tail := Tail^.Next;
```

The last statement is an excerpt from a program that uses a doubly-linked list of integers. The statement inserts a new element in the end of the list and updates the necessary pointers in one statement. This is useful since the statement is atomically executed and other statements cannot change the same pointers during the execution of that statement.

2.3.2 Thread Declarations

Threads are declared as types in Vex. A thread type defines the behavior of a spawned thread (see Section 2.4.2). The declaration of a thread type resembles the declaration of a procedure, allowing the thread to be parameterized. A thread has a declaration section before the body of the thread which allows for thread-local variables. The body of a thread is declared within a block statement (begin-end).

Here is an example of a thread declaration:

```
Type
  MyThread = Thread (InputValue : Integer)
Var
  LocalVar : String;
Begin
  LocalVar := 'HelloWorld';
  While InputValue > 0 do
  Begin
    Print(LocalVar);
    InputValue := InputValue - 1;
  End;
End;
```

This declares a new type, `MyThread`. A `MyThread` thread can be spawned after the declaration of MyThread using the `spawn` statement.

2.3.3 If-Then-Else Statement

The if-then-else statement has been modified to remove ambiguity. By requiring a begin/end block statement after the if-then part the ambiguity of Pascal's if-then-else statement is removed. The modification is necessary because the GOLD Parser System is used for parsing and it does not allow ambiguous grammar.

The following code is legal Delphi code but illegal in Vex:

```
If Condition1 Then
If Condition2 Then
    DoSomething
Else If Condition3 Then
```

```
    DoSomethingDifferent;
Else
    DoSomethingElse;
```

This code is ambiguous because it is not clear which if statements the else 'statements' are associated with. Delphi deals with this issue by ruling that any else statement is associated with the innermost preceding if statement, thereby disambiguating the syntax.

The following code is the correct Vex if-then-else statement corresponding to the previous code example:

```
If Condition1 Then Begin
  If Condition2 Then
  Begin
    DoSomething;
  End
  Else If Condition3 then
  Begin
    DoSomethingDifferent;
  End
  Else
    DoSomethingElse;
End;
```

Note that this is legal syntax in both Pascal, Delphi and Vex.

2.3.4 If-Do-Else-Do Statement

This statement is a high level version of a compare-and-swap operation that allows for conditional assignments. The statement can be considered a simplified version of an If-Then-Else statement. The are two syntax definitions for an If-Do statement. The simplest If-Do statement contains no else part and is defined as:

```
<Ifdo stm> ::=
    'if' <Expression> 'do' <Assign stm>
```

If `<Expression>` evaluates to true then the `<Assign stm>` is executed. If `<Expression>` evaluates to false then no assignments are executed.

The second syntax definition is:

```
<Ifdo stm> ::=
    'if' <Expression> 'do' <Assign stm> 'else' 'do' <Assign stm>
```

If `<Expression>` evaluates to true then the first `<Assign stm>` is executed. If `<Expression>` evaluates to false then the second `<Assign stm>` is executed.

The execution of an If-Do statement can be somewhat more complicated in practice because of the locking involved. In practice, `<Expression>` may be evaluated several times before any `<Assign stm>` is executed. The details of executing If-Do statements are described in Chapter 3.

2.4 New Syntax

The syntactical additions to Vex which are not present in Pascal are related to the dynamic deallocation of memory and thread spawning. These additions are described in the following subsections.

2.4.1 Free Statement

The free statement is used to deallocate dynamically allocated memory. The `free` keyword must be followed by a typed pointer that points to a dynamically allocated record. The syntax for the free statement is defined as:

```
<Free stm> ::=
    'free' <Qualified identifier>
```

An example is listed below:

```
PtrToFree := Tail &
Tail := Tail^.Prev &
Tail^.Next := @Nil;
Free PtrToFree;
```

The example above removes the last element in a doubly-linked list by updating the necessary pointers before deallocating the element.

2.4.2 Spawn Statement

The spawn statement initializes and starts a new thread. A spawn statement resembles a procedure call preceded by the `spawn` keyword. The actual parameters to the spawned thread must correspond with the formal parameters in the declaration of the thread type. The syntax is defined as:

```
<Spawn stm> ::=
    'spawn' <Type> |
    'spawn' <Type> '(' <Expression List> ')'
```

An example of usage is illustrated in Figure 2.2.

2.5 The Runtime Host Program

A Vex program is compiled to a Delphi unit. This unit is used by a Delphi program file - the host program. The host program also uses other units with utility functions and classes. The use of a host program makes it easy to add functionality such as text output and timing capabilities to a Vex program.

The host program keeps track of threads which makes it easy to both join and to not terminate before all threads have terminated.

The host program also has functionality for timing the execution of a Vex program. The CPU timer is used for precision timing.

The output is updated at most every 250 ms. to keep timing interference at a minimum. The print function is thread safe and uses a lock and busy waiting to assure mutual exclusion.

2.6 Summary

The syntax of Vex is very close to that of Delphi and Pascal. The modifications and additions are were made to adhere to the Pascal syntax tradition. The exception to this is the use of the '`&`' symbol to construct a sequence of assignments. The use of a symbol (instead of an English word) is somewhat untraditional in the Pascal language tradition. An obvious replacement is the word "AND" but this is already used in expressions and would cause ambiguity. The choice of using the symbol '`&`' can be considered a lack of imagination on the part of the author.

Chapter 3

Micro Transactions

A micro transaction is a an atomic assignment statement (consisting of one or more assignments in sequence) or an atomic If-Do-Else-Do statement (enclosing one or two assignment statements) that contain references to at least one protected variable. A protected variable is a variable that potentially requires protection from race conditions. Each protected variable is associated with a specific lock and each lock is associated with only one variable. The compiler determines which variables are protected by static analysis (see Section 3.1).

A micro transaction has three steps:

1. Locking phase: Protected variables are locked
2. Execution phase: Execute assignments
3. Unlocking phase: Protected variables are unlocked

The locking protocol uses conservative two phase locking (C2PL) [4]. The C2PL protocol acquires all locks before commencing a transaction and releases all locks after a transaction has committed or aborted. If all locks cannot be acquired in the locking phase then the locks that were acquired are released and the locking phase restarts immediately in a busy-wait fashion.

The atomicity of micro transactions is provided by locking all protected variables before executing the assignments in the transaction, ensuring mutual exclusive access to those variables. This ensures consistency in the programming model. When all assignments in a micro transaction have been executed all locks are released and the transaction has committed. A committed transaction cannot be undone or rolled back at a later time which ensures durability. Isolation in micro transactions is only guaranteed with respect to other transactions. That is, an inconsistent view of the system can be observed since only assignments are protected by locks.

No logs are used to implement micro transactions. Updates are made directly to variables 'in-place'. Since an exclusive lock is held for all protected variables

during transactions there is no need for keeping logs. The exception to this is when exceptions or runtime errors occur during transactions. This situation is not handled by the current implementation (see Section 3.5).

Since a micro transaction only contains assignments the locks are only held for a short period of time. The time complexity of the execution phase in a transaction is $O(n)$ where n is the number of assignments in the statement. It was a design goal to have short locking periods and use busy waiting to avoid expensive context switching.

Several issues must be addressed to ensure that transactions work properly. Since micro transactions are based on locks there is an inherent risk of deadlocking and livelocking. Deadlocks are prevented by removing one of the conditions necessary for deadlock to occur. Deadlocks and livelocks are described in Section 3.2.5 and Section 3.2.6.

3.1 Protected Variables

It would be easy to assume that all variables are protected and simply lock every variable when necessary. This, however, would be very inefficient because an overhead is associated with locking each variable. The challenge is to determine a minimum set of variables that needs locking. Initially, we shall assume that all variables are protected.

First, we can exclude any variable that is declared in a scope S and are never referenced from a child scope of S. Secondly, any variable that is declared within a scope S and only read in a child scope of S can be excluded. The group of variables that are declared within a scope S and are written to inside a child scope of S are included in the set of protected variables. This group of variables is referred to as *shared variables* henceforth. To determine whether a variable is written to in a child scope of the scope in which it was declared only requires a look-up in the symbol table for every applied occurrence of the symbol. In a more expressive language where, for example, by-reference parameters were available this determination would require a more complicated analysis.

The last set of variables are records that are dynamically allocated using the `new` keyword. Pointers to these records may be used in other scopes through parameters or other pointers. Instead of using complicated analysis to determine which records requires locking and which do not, the principle of simplicity is applied and all records are included in the set of protected variables. Ideally, some of the records could be excluded from the set of protected variables but the overhead of locking some 'false-positives' was deemed acceptable.

Pointers may be assigned new values during the execution phase of a micro

transaction so a points-to analysis is used to determine the set of locks that must be acquired in the locking phase. The points-to analysis is described in Section 3.3.

3.2 Locking Protocols

There are two syntactical elements which are used for transactions: the assignment statement and the If-Do-Else-Do statement. The syntax of these statements is described in Section 2.3. The If-Do-Else-Do statement contains assignment statements and allows for conditional assignments. When an assignment statement is located within an If-Do-Else-Do statement it is treated slightly different than when it is not. The following sections explain the locking protocols associated with these two statements.

3.2.1 Assignment Statement

This section describes the locking protocol of an assignment statement that is not located inside an If-Do-Else-Do statement. Consider the following assignment statement:

$$V_1 \texttt{ := } E_1 \texttt{ \& } V_2 \texttt{ := } E_2 \texttt{ \& } \ldots \texttt{ \& } V_n \texttt{ := } E_m \texttt{;}$$

A set of protected variables, $protected = \{P_1, P_2, \ldots, P_k\}$, is constructed through the points-to analysis s.t. all the protected variables in $\{V_1, V_2, \ldots, V_n\}$ and in expressions $E_1, E_2, ..., E_m$ are in *protected*.

We introduce a function, *Lock*, that tries to acquire the locks for a set of variables. A return value of *true* signals that all locks were successfully acquired and a value of *false* means that no locks were acquired. The *lock* function is listed in Algorithm 1.

We introduce a procedure, *Unlock*, that releases the locks for a set of variables. The function is defined in Algorithm 2.

The locking protocol for assignment statements is listed in Algorithm 3. The risk of livelocking is easy to see in Algorithm 3. If any lock cannot be acquired then *Lock* returns $false$ and the while loop runs again. This means that there is a risk that the loop will never terminate. The livelock risk is described in Section 3.2.6.

If no protected variables are referenced in an assignment statement then the generated code is simply a sequence of Pascal assignment statements.

Algorithm 1 The Lock function.

```
Function Lock( {L_1, L_2, ..., L_x} )
i = 1
while i ≤ x do
  Try to acquire lock for variable L_i
  if lock was acquired then
    i = i + 1
  else
    while i > 1 do
      i = i − 1
      Release lock for variable L_i
    end while
    Return false
  end if
end while
Return true
```

Algorithm 2 The Unlock function.

```
Procedure Unlock( {L_1, L_2, ..., L_x} )
i = 1
while i ≤ x do
  Release lock for variable L_i
  i = i + 1
end while
```

Algorithm 3 Locking protocol for assignment statements.

```
{Locking phase}
while not Lock( protected ) do
  {Do nothing}
end while
{Execution phase}
V_1 := E_1 &
...
V_n := E_m;
{Unlocking phase}
Unlock( protected )
```

3.2.2 If-Do Statement

This section describes the locking protocol of the If-Do-Else-Do statement. Consider the following statement:

```
If C1 and C2 and ... and Cw Do
   V1 := E1 & V2 := E2 & ... & Vs := Es;
Else Do
   Vs+1 := Es+1 & Vs+2 := Es+2 & ... & Vn := En;
```

Three sets of protected variables are determined through three separate points-to analyses: $protected_{exp}$, $protected_A$ and $protected_B$. We exclude all the variables in $protected_{exp}$ from $protected_B$ and $protected_B$ s.t. these variables are not locked twice.

$protected_{exp} = \{P_1, P_2, \ldots, P_l\}$ s.t. all protected variables in conditions $C_1, C_2, \ldots, C_n$ are in $protected_{exp}$.

$protected_A = \{P_{l+1}, P_{l+2}, \ldots, P_{l+k}\} \setminus protected_{exp}$ s.t. all protected variables in $\{V_1, V_2, \ldots, V_s\}$ and in expressions $E_1, E_2, ..., E_s$ are in $protected_A$, while none of the protected variables in $protected_{exp}$ are in $protected_A$.

$protected_B = \{P_{l+k+1}, P_{l+k+2}, \ldots, P_l + k + r\} \setminus protected_{exp}$ s.t. all the protected variables in $\{V_{s+1}, V_{s+2}, \ldots, V_n\}$ and in expressions $E_{s+1}, E_{s+2}, ..., E_n$ are in $protected_B$ while none of the protected variables in $protected_{exp}$ are in $protected_B$.

Algorithm 4 describes the locking protocol for the If-Do-Else-Do statement.

If no protected variables are referenced in an If-Do-Else-Do statement or in the assignment statements herein then the generated code is simply a Pascal If-Then-Else statement.

3.2.3 Acquiring Locks

Locks are implemented using integers. For each protected variable an additional integer variable is allocated by the compiler which is used to lock the protected variable. The lock variable is initialized with a value of 0 to indicate an unlocked status. A nonzero value indicates a locked status.

When a lock is acquired the value of the lock variable is set to the current thread id. This indicates not only that the lock has been acquired but also which thread that has acquired the lock. This is necessary to correctly lock dynamically allocated records. The static points-to analysis cannot determine if multiple pointers refer to the same record or not. Instead, a runtime check inspects the lock variable and determines whether it is already locked by the current thread

Algorithm 4 The locking protocol for If-Do-Else-Do statements.

```
1: {Locking phase}
2: while not Lock( protected_exp ) do
3:    {Do nothing}
4: end while
5: Abort = False
6: if C_1 and C_2 and ... and C_w then
7:    {Locking phase - protected_A}
8:    Abort = not Lock( protected_A )
9:    if Abort then
10:      Unlock( protected_exp )
11:      Goto 2
12:   end if
13:   {Execution phase A}
14:   V_1 := E_1 &
15:   ...
16:   V_s := E_s;
17:   Unlock( protected_A )
18: else
19:   {Locking phase - protected_B}
20:   Abort = not Lock( protected_B )
21:   if Abort then
22:      Unlock( protected_exp )
23:      Goto 2
24:   end if
25:   {Execution phase B}
26:   V_s+1 := E_s+1 &
27:   ...
28:   V_n := E_n;
29:   Unlock( protected_B )
30: end if
31: Unlock( protected_exp )
```

or by another thread. If the variable is already locked by the current thread then it is disregarded and the locking phase continues. If the variable is locked by another thread then all locks are released and the locking phase restarts.

3.2.4 Releasing Locks

The memory address of an acquired lock is pushed onto the stack during the locking phase. We do this because the assignments in a transaction can change pointers which means that we might 'loose' references to locks as a result. Consider the following transaction:

```
ptr^.value := 42 &
ptr := b;
```

After this transaction the reference to the record that `ptr` pointed to at the beginning of the transaction is lost. Saving a reference on the stack before executing the assignments solves this problem. During the unlocking phase the address of each lock is popped from the stack and a value of 0 is written to the lock.

3.2.5 Deadlock Prevention

There are four necessary conditions for deadlocks to occur [7]. By removing one of these conditions the risk of deadlocking is prevented. The conditions for deadlock are:

1. The 'mutual' exclusion condition: Tasks claim exclusive control of the resources they require.
2. The 'wait for' condition: Tasks hold resources already allocated to them while waiting for additional resources.
3. The 'no preemption' condition: Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion.
4. The 'circular wait' condition: A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain.

Deadlocks are prevented in Vex by removing the second condition: All necessary locks are acquired before beginning a transaction. If any of the locks are

already acquired by another transaction then all locks are released and the locking protocol restarts.

A consequence of this is that function calls are prohibited in transactions and must be made outside of a transaction. This restriction is necessary because functions may contain transactions that will try to acquire additional locks and thereby voiding the deadlock prevention.

3.2.6 Livelock Avoidance

A livelock can occur if two or more transactions are repeatedly trying to lock the same two protected variables in reverse order. The following pseudo code illustrates this problem:

```
Thread 1:                    Thread 2:
  Lock A                       Lock B
  Lock B                       Lock A
  (Do assignments)             (Do assignments)
  Unlock B                     Unlock A
  Unlock A                     Unlock B
```

If each thread has acquired one lock at the same time then none of the threads can acquire their second lock. They will both release their locks and retry. This conflict can, theoretically, continue indefinitely, constituting a livelock. In practice, however, this scenario is not likely to continue indefinitely. This would require that the scheduler schedules and preempts the two threads at same the time indefinitely on a multi-core processor or that the scheduler preempts the threads between the first and the second lock indefinitely on a single-core processor. It should be noted, however, that the risk of livelock increases with the number of protected variables that are used in both of the conflicting transactions.

The livelock can be prevented if the set of locks is a totally ordered set and locks are always acquired in order. The dynamically allocated records (with associated locks) in Vex makes it impossible to determine a totally ordered set statically. A dynamic approach where locks are sorted at runtime could be attempted. But this approach was discarded because it would complicate the locking process and incur an overhead in the locking process.

Instead of implementing livelock prevention we try to avoid livelocks. This means that there is a risk of livelocking but that the compiler tries to reduce the risk statically. This requires no extra checking at runtime and the static analysis involved is simple.

After type checking the Vex source code the compiler has a list of protected variables. This list is sorted according to the name of the variables. In many

cases this will prevent livelocking but when pointers are involved in transactions livelocks can still occur.

3.3 Points-to Analysis

The purpose of the points-to analysis is to acquire the correct locks in the locking phase, to release the correct locks in the unlocking phase, to determine a minimal set of locks and to ensure that references to locks are preserved for the unlocking phase.

The value of pointer fields in records are tracked which means that the analysis is field-sensitive[12]. The analysis is flow-sensitive which means that the order of the assignments in a micro transaction is taken into account. This gives a precise result as opposed to a flow-insensitive analysis. A precise flow-sensitive analysis is possible because micro transactions have a very simple control flow with no iteration and one branching at most. A precise flow-sensitive analysis of a Turing-complete language is not possible, per Rice's Theorem[13], because it requires decidability of a non-trivial language property.

3.3.1 An Example

This section serves as an introduction to the points-to analysis. The analysis of an example transaction is explained step-by-step. The following declarations are assumed for the transaction:

```
Type
  PElement = ^TElement;
  TElement = Record
    value : Integer;
    next  : PElement;
  End;

Var
  p, head, newhead : PElement;
  value1, value2 : Integer;
```

The micro transaction we will be analyzing is the following:

```
p       := head &
value1  := p^.value &
newhead := p^.next &
value2  := newhead^.value &
head    := newhead &
p^.next := @nil &
p       := p^.next;
```

In this example the pointers (`p`, `head` and `newhead`) are protected variables and must be locked. All referenced records must also be locked. The points-to analysis determines which records must be locked in the locking phase and unlocked in the unlocking phase in order to lock correctly.

It is assumed that the pointers which are used in the assignments - prior to being assigned new values - are initialized before the transaction is started. In this example `head` and `head^.next` are assumed initialized. These assumptions are part of the points-to analysis, as the example will show.

To analyze the transaction we construct a labeled directed multigraph. The graph is initialized s.t. it contains one vertice, *nil*, and no edges, as illustrated in Figure 3.1. The set $lockset = \emptyset$ is constructed. *lockset* is used to identify variables and records that needs to be locked.

Figure 3.1: *The points-to graph after initialization.*

Assignments consist of a left-hand-side and a right-hand-side, separated by the assignment symbol: ':='. When analyzing assignments the right-hand-side is analyzed before the left-hand-side.

The first assignment, '`p := head`', is analyzed. The right-hand-side has a single identifier, `head`, which is a pointer so we construct a new vertex, *head*. We assume that the pointer is initialized and points to a valid record so we construct a vertex, $record_1$, and a directed edge from *head* to $record_1$, labeled *head*. Next we process the left-hand-side by creating a vertex, p, and an directed edge from p to the vertex that *head* has a edge to, $record_1$. We label this edge p. Both p and *head* are protected variables so we include p and *head* in *lockset* s.t. $lockset = \emptyset \cup \{p, head\} = \{p, head\}$. Now the assignment has been processed and the resulting graph is illustrated in Figure 3.2.

Figure 3.2: *The points-to graph after the first assignment.*

In the second assignment, '`value1 := p^.value`', the qualified identifier on the right-hand-side is processed. p exists and has an edge to $record_1$ and `p^.value` is not a pointer so we do not add any new vertices or edges. But since `p` is dereferenced it needs to be locked so it is marked with the *lock* function s.t. $lock(record_1) = true$. The left-hand-side is processed and because `value1` is not pointer no further changes are made to the graph. After processing the second assignment the graph is unchanged.

The right-hand-side of the third assignment, '`newhead := p^.next`', is the pointer, `p^.next`. The p vertex exists and it has an edge to $record_1$ but no *next* edge from $record_1$ exists so we create a new vertex, $record_2$ and a directed edge from $record_1$ to $record_2$, labeled *next*. This is under the assumption, again, that `p^.next` is an initialized pointer to an existing record. The left-hand-side contains the pointer `newhead` so we create a vertex *newhead* and a directed edge from *newhead* to $record_2$, labeled *newhead*. *newhead* is a protected variable so we include it in *lockset* s.t. $lockset = \{p, head\} \cup \{newhead\} = \{p, head, newhead\}$. The graph now looks shown in Figure 3.3.

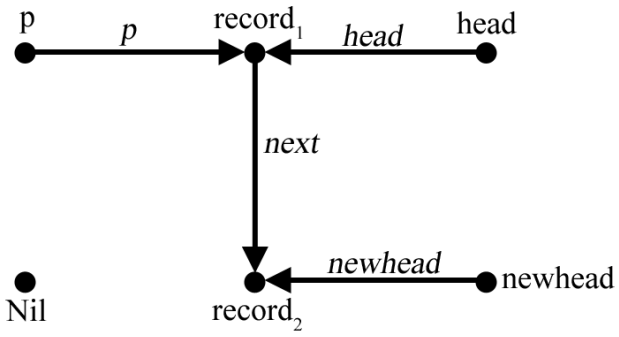


Figure 3.3: *The points-to graph after the third assignment.*

The fourth assignment, '`value2 := newhead^.value`', is analyzed in the same manor as the second assignment and leaves the graph unchanged. Since `newhead` is dereferenced the corresponding vertex, $record_2$, needs locking and it is marked with the *lock* function s.t. $lock(record_2) = true$.

The pointers used in the fifth statement, '`head := newhead`', cause no new vertices. However, because an edge labeled *head* already exists from *head* to another vertex we create a new edge labeled *head′*. This edge is oriented from *head* to $record_2$ because the *newhead* edge ends at $record_2$. The resulting graph is illustrated in 3.4

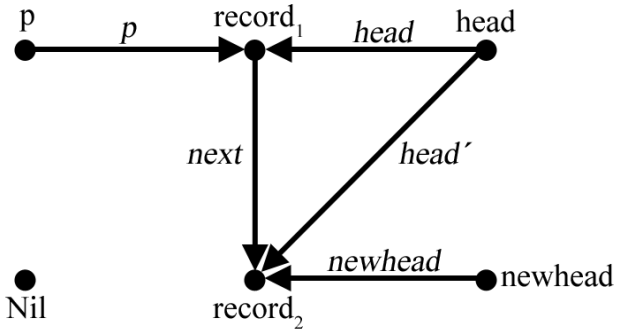


Figure 3.4: *The points-to graph after the fifth assignment.*

In the sixth assignment, '`p^.next := @nil`', the right-hand-side causes no new vertices. A vertex p corresponding to the pointer `p` on the left-hand-side already exists and the edge labeled p ends in $record_1$. A *next* edge from $record_1$ already exists so a new edge, labeled $next'$, is constructed from $record_1$ to nil. The graphs now looks as illustrated in Figure 3.5.

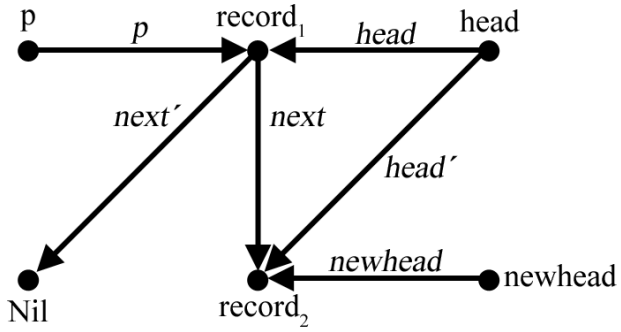


Figure 3.5: *The points-to graph after the sixth assignment.*

In the seventh and last assignment, '`p := p^.next`', the right-hand-side causes no new vertices. The edge labeled p points to $record_1$ which has a edge labeled $next'$ to nil. The vertex p corresponding to the left-hand-side already exists, as does the edge labeled p so a new edge, labeled p', is constructed from p to nil. This completes the graph which now looks as illustrated in Figure 3.6.

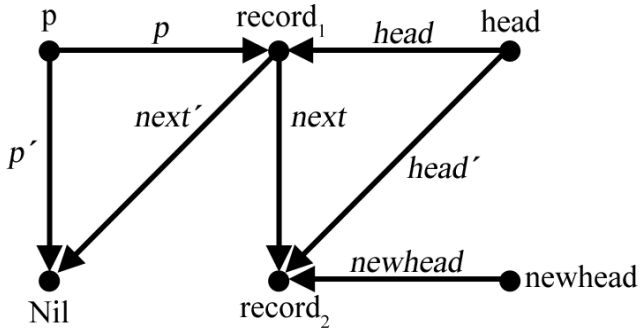


Figure 3.6: *The points-to graph after the seventh and last assignment.*

By observing the graph we can easily see that only two records are referenced in the transaction, $record_1$ and $record_2$. Both records need to be locked since $lock(record_1) = lock(record_2) = true$. We cannot use the pointers that set the *lock* value to true, `p^` and `newhead^`, to lock the records because these pointers do not point to the records before the transaction.

We need to use the graph to determine which pointers to use for locking the records. Also, we need another function, *owner*, to determine the pointers. The *owner* function is defined s.t. $owner(V) = W$ where W is the vertex from

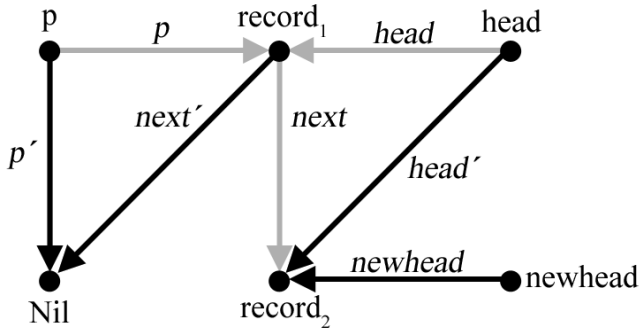


Figure 3.7: *The points-to graph after removing edges s.t. the graph reflects the values of pointers after the transaction. 'Removed' edges are drawn in gray and the remaining edges in black.*

which the first edge to V originated. $owner(V) = nil$ for any vertice V to which there are no edges. For the example this means that $owner(record_1) = head$, $owner(record_2) = record_1$ and for any other vertice V, $owner(V) = nil$. Now we can easily find the correct pointers to $record_1$ and $record_2$ by using the *owner* function and the edges in the graph.

For $record_1$ we use $owner(record_1) = head$ and find the edge from *head* to $record_1$, labeled *head*. Since $owner(head) = nil$ we are done. The pointer is simply *head^* (the dereference symbol is added to differentiate from the variable *head*). We add *head^* to *lockset* s.t. $lockset = \{p, head, newhead\} \cup \{head\hat{}\} = \{p, head, newhead, head\hat{}\}$.

For $record_2$ we use $owner(record_2) = record_1$ and find the edge from $record_1$ to $record_2$, labeled *next*. Next we use $owner(record_1) = head$ to find the edge from *head* to $record_1$, labeled *head*. And since $owner(head) = nil$ we are done. So the pointer to $record_2$ is $head\hat{}.next\hat{}$ and we include this in *lockset* s.t. $lockset = \{p, head, newhead, head\hat{}\} \cup \{head\hat{}.next\hat{}\} = \{p, head, newhead, head\hat{}, head\hat{}.next\hat{}\}$.

By removing any edge e where a corresponding edge e' exists we can see, as illustrated in Figure 3.7, that after the transaction completes we no longer have a reference to $record_1$. This is why we push the address of the pointers on the stack during the locking phase.

To acquire locks to the correct variables and records we use *lockset* and push the addresses of the locks onto the stack. To release the locks we pop the addresses from the stack. If we simply used *lockset* to release the locks - without using the stack - we would quickly run into problems since `head^.next^` is undefined after the transaction.

This concludes the example. Before continuing to the actual algorithm it should be noted that we used edges with labels such as *head* and *head′* for sim-

plicity in this example. The actual algorithm uses one set of edges for the initial value of pointers, *edges*, and another set of edges for the value of pointers after the transaction, $edges'$. So in the actual algorithm, two edges which are both labeled *head* would exist; One in *edges* and one in $edges'$.

3.3.2 Algorithm for Assignment Statements

Qualified identifiers, denoted by a Q, are used extensively throughout the analysis. Qualified identifiers are a sequence of one or more identifiers, denoted by I. The syntax of Qualified identifiers is defined as:

```
<Qualified identifier> ::=
    Identifier |
    Identifier '^' |
    Identifier '.' <Qualified identifier> |
    Identifier '^' '.' <Qualified identifier>
```

The analysis constructs a directed labeled multigraph. The vertices of the graph is the set $\mathbb{V} = vertices$ and the edges of the graph is the set $\mathbb{E} = edges \cup edges'$. $\mathbb{V}$ initially contains one vertex, *nil*, which represents the `nil` value (an explicitly unassigned pointer). $\mathbb{E}$ is empty after initialization. The edges in *edges* represent the initial value of pointers to referenced records. The edges in $edges'$ represent the value of pointers to referenced records after the transaction has completed.

The set $\mathbb{I}$ represents the set of identifiers in a program and the set $\mathbb{Q}$ represents the set of qualified identifiers that can be constructed from $\mathbb{I}$.

A number of functions are used to hold information about vertices, edges and identifiers. These functions are described below:

lock: $\mathbb{V} \rightarrow \{true, false\}$ The *lock* function returns true if a record or variable that a vertex $V \in \mathbb{V}$ represents must be locked before a transaction is started. If V does not need to be locked then $lock(V) = false$.

protected: $\mathbb{I} \rightarrow \{true, false\}$ The *protected* function returns true if an identifier $I \in \mathbb{I}$ is a protected variable. If I is not a protected variable then $protected(I) = false$.

label: $\mathbb{E} \rightarrow string$ The *label* function returns the label of an edge $E \in \mathbb{E}$. The return value of *label* is a text string.

name: $\mathbb{V} \rightarrow string$ The *name* function returns a string with the name of a vertex $V \in \mathbb{V}$. If the function is undefined for a vertex V then *name* returns the empty string, s.t. $name(V) =$ ''. The *name* function is defined

for vertices that represent variables. It is used to ensure that duplicate vertices are not constructed.

identifiername: $\mathbb{I} \rightarrow string$ The $identifiername$ function returns a string with the name of an identifier $I \in \mathbb{I}$.

dereferenced: $\mathbb{I} \rightarrow \{true, false\}$ The $dereferenced$ function returns true if an applied occurrence of a pointer identifier $I \in \mathbb{I}$ is dereferenced, i.e. $dereferenced(\texttt{ptr\^{}}) = true$ and $dereferenced(\texttt{ptr}) = false$.

new: $\mathbb{V} \rightarrow \{true, false\}$ The *new* function returns true if the record associated a vertex $V \in \mathbb{V}$ is allocated within the transaction. Any record allocated within the transaction does not need locking because no other transaction can have a reference to it.

owner: $\mathbb{V} \rightarrow \mathbb{V}$ The *owner* function is used to construct a linked list of vertices which represent a qualified identifier and the records that are referenced through the qualified identifier. V is the first identifier in a qualified identifier when $owner(V) = nil$. For the special *nil* vertex, the function is defined as $owner(nil) = nil$.

lastidentifier: $\mathbb{Q} \rightarrow \mathbb{I}$ The $lastidentifier$ function returns the last identifier of a qualified identifer. The function returns I_n for a qualified identifier Q, where $Q = (I_1.I_2.\cdots.I_n)$.

The main algorithm for the points-to analysis of an assignment statement, *AnalyzeAssignment*, is listed in Algorithm 5. The edges and vertices of the graphs are initialized before the statements of the transaction are analyzed. Each type of assignment statement is processed by a separate function. After the statements have been analyzed, a list of qualified identifiers, $lockset_A$, is constructed. This set is used by the code generator to generate assembly code for the locking protocol of micro transaction A.

Two helper functions are used in the analysis of the statements. The fist helper function, *AddRecord*, is listed in Algorithm 6. This function adds a new vertex, W, that represents a record to the graph. The first parameter, V, identifies the vertex from which the record is first referenced. This vertex is saved as the owner of the new vertex. The second parameter, *LabelName*, is the name of the identifier that references W. If an edge labeled *LabelName* from the owner vertex, V, already exists in $edges'$ then this edge is removed. A new edge, E, is constructed from V to W with the label $LabelName$. This edge is included in *edges* to indicate that this is the initial reference to the record. The edge is also included in $edges'$ to indicate that after the transaction the record is also referenced by V. The edge is removed from $edges'$ later if a statement subsequently

Algorithm 5 The AnalyzeAssignment function.

Function AnalyzeAssignment ($A = (S_1 \& S_2 \& \ldots \& S_n)$)
$edges = \emptyset$
$edges' = \emptyset$
$vertices = \{nil\}$
for i = 1 to n **do**
  if S_i is an allocation assignment **then**
    Call AllocationAssignment(S_i)
  end if
  if S_i is an exchange assignment **then**
    Call ExchangeAssignment(S_i)
  end if
  if S_i is a pointer assignment **then**
    Call PointerAssignment(S_i)
  end if
  if S_i is a general assignment **then**
    Call GeneralAssignment(S_i)
  end if
end for
$lockset_A$ = MakeLockSet
return $lockset_A$

Algorithm 6 The AddRecord helper function.

Function AddRecord (V, LabelName)
if $E' = (V, W')$ exists *where* $E' \in edges' \wedge label(E') =$ LabelName **then**
  $edges' \Leftarrow edges' \setminus \{E'\}$
end if
$vertices \Leftarrow vertices \cup \{W\}$
$owner(W) = V$
$E = (V, W)$
$label(E) =$ LabelName
$edges \Leftarrow edges \cup \{E\}$
$edges' \Leftarrow edges' \cup \{E\}$
$new(W) =$ **false**
return W

assigns a new value to the identifier represented by V and *LabelName*. Finally, the new vertex is returned to the caller.

The second helper function, *AddQualifiedIdentifier*, is listed in Algorithm 7. The function adds vertices and edges to the graph to represent the qualifier identifier Q (the parameter). The first identifier in Q is a variable in the program and a vertex is constructed to represent this (if one does not already exist). The identifiers in Q are traversed adding vertices and edges for identifiers that are not already represented in the graph. If the last identifier is not a pointer then no edge or vertex is constructed for this identifier. If the first identifier is a protected variable then *lock* of the associated vertex defined as *true*. For any other identifier *lock* is always defined as *true* for the record wherein the identifier is a field. If the last identifier is a dereferenced pointer then *lock* of the 'referenced' vertex is defined as *true*. The vertex associated with the variable ($n = 1$) or the last referenced record ($n > 1$) is returned.

The *AllocationAssignment* function is listed in Algorithm 8. This function calls *AddQualifiedIdentifier* to ensure that the vertices and edges needed to represent the qualified identifier on the left-hand-side of the assignment, Q, are present in the graph. Then, a call to *AddRecord* adds a new vertex, W, to the graph to represent a newly allocated record. Finally, the value of the $new(W)$ is defined to *true* to indicate that the record was allocated within the transaction.

The *ExchangeAssignment* function, listed in Algorithm 9, calls *AddQualifiedIdentifier* to construct the vertices and edges for the qualified identifiers Q_1 and Q_2. The current edges from Q_1 and Q_2 in $edges'$ are found and used to construct new edges. Finally, the current edges in $edges'$ are replaced by the new edges.

The *PointerAssignment* function, listed in Algorithm 10, calls *AddQualifiedIdentifier* to construct the vertices and edges for the qualified identifier Q_1 on the left-hand-side of the assignment. The current edge from Q_1 in $edges'$ is found and removed from $edges'$. If the qualified identifier on the right-hand-side is '`nil`' then a new edge, E' is constructed from the vertex associated with the last identifier in Q_1, V_1, to the vertex *nil*. If the qualified identifier on the right-hand-side is not '`nil`' then we call *AddQualifiedIdentifier* to construct the vertices and edges for Q_2. We construct a new edge, E', from V_1 to W_2 where W_2 represents the record that Q_2 points to. Finally, the E' edge is included in $edges'$.

The *GeneralAssignment* function, listed in Algorithm 11, does one of two things: If the last identifier in the qualified identifier on the left-hand-side of the assignment, Q_1, is a pointer then the expression, *Exp*, on the right-hand-side contains exactly one qualified identifier, Q_2. In this case, the *AddQualifiedIdentifier* function is invoked twice to create vertices and edges for Q_1 and Q_2. A new edge, $E_1{}'$, is constructed from the vertex that represents the last identifier in Q_1, V_1, to the vertex that represents the record referenced by Q_2, W_2. $E_1{}'$ is included

Algorithm 7 The AddQualifiedIdentifier helper function.

Function AddQualifiedIdentifier ($Q = (I_1.I_2.\cdots.I_n)$)
if W exists *where* $W \in vertices \wedge name(W) = identifiername(I_1)$ **then**
  $V = W$
else
  $name(V) = identifiername(I_1)$
  $lock(V) = protected(I_1)$
  $owner(V) = nil$
  $vertices \Leftarrow vertices \cup \{V\}$
end if
for k := 1 to n **do**
  if $E = (V, W')$ exists *where* $E \in edges' \wedge label(E) = identifiername(I_k)$ **then**
    $W \Leftarrow W'$
  else if $k < n \vee I_n$ is a pointer **then**
    $W \Leftarrow$ AddRecord(V,$identifiername(I_k)$)
  end if
  if $k > 1$ **then**
    $lock(V) =$ **true**
  end if
  if $k < n$ **then**
    $V \Leftarrow W$
  else if I_n is a pointer $\wedge$ $dereferenced(I_n)$ **then**
    $lock(W) =$ **true**
  end if
end for
return V

Algorithm 8 The AllocationAssignment function.

Function AllocationAssignment ($S = (Q$ `:= new` $T)$)
$V =$ AddQualifiedIdentifier(Q)
$W =$ AddRecord(V, $identifiername$(lastidentifier(Q)))
$new(W) =$ **true**
return

Algorithm 9 The ExchangeAssignment function.

Function ExchangeAssignment ($S = (Q_1$ `<=>` $Q_2)$)
$V_1 =$ AddQualifiedIdentifier(Q_1)
$V_2 =$ AddQualifiedIdentifier(Q_2)
$I_1 = lastidentifier(Q_1)$
$I_2 = lastidentifier(Q_2)$
$E_1 = (V_1, W_1)$ *where* $E_1 \in edges' \wedge label(E_1) = identifiername(I_1)$
$E_2 = (V_2, W_2)$ *where* $E_2 \in edges' \wedge label(E_2) = identifiername(I_2)$
$E_1{}' = (V_1, W_2)$
$E_2{}' = (V_2, W_1)$
$label(E_1{}') = label(E_1)$
$label(E_2{}') = label(E_2)$
$edges' \Leftarrow edges' \setminus \{E_1, E_2\}$
$edges' \Leftarrow edges' \cup \{E_1{}', E_2{}'\}$
return

Algorithm 10 The PointerAssignment function.

Function PointerAssignment ($S = (Q_1$ `:= @` $Q_2)$)
$V_1 =$ AddQualifiedIdentifier(Q_1)
$I_1 = lastidentifier(Q_1)$
$E = (V_1, W_1)$ *where* $E \in edges' \wedge label(E) = identifiername(I_1)$
$edges' \Leftarrow edges' \setminus \{E\}$
if $Q_2 =$ '`nil`' **then**
  $E' = (V_1, nil)$
else
  $V_2 =$ AddQualifiedIdentifier(Q_2)
  $I_2 = lastidentifier(Q_2)$
  $E'' = (V_2, W_2)$ *where* $E'' \in edges' \wedge label(E'') = identifiername(I_2)$
  $E' = (V_1, W_2)$
end if
$edges' = edges' \cup \{E'\}$
$label(E') = label(E)$
return

Algorithm 11 The GeneralAssignment function.

Function GeneralAssignment ($S = (Q_1$:= $Exp)$)
V_1 = AddQualifiedIdentifier(Q_1)
if lastidentifier(Q_1) is a pointer **then**
{*Exp contains exactly one qualified identifier* (Q_2) *and nothing else*}
$Q_2 \Leftarrow$ The qualified identifier in Exp
V_2 = AddQualifiedIdentifier(Q_2)
$I_1 = lastidentifier(Q_1)$
$I_2 = lastidentifier(Q_2)$
$E_1 = (V_1, W_1)$ *where* $E_1 \in edges' \wedge label(E_1) = identifiername(I_1)$
$E_2 = (V_2, W_2)$ *where* $E_2 \in edges' \wedge label(E_2) = identifiername(I_2)$
$E_1{}' = (V_1, W_2)$
$label(E_1{}') = label(E_1)$
$edges' \Leftarrow edges' \setminus \{E_1\}$
$edges' \Leftarrow edges' \cup \{E_1{}'\}$
else
for every qualified identifier Q_2 in Exp **do**
V_2 = AddQualifiedIdentifier(Q_2)
end for
end if
return

in $edges'$ after the old edge, E_1, has been removed from $edges'$.

The other case - when the last identifier in Q_1 is not a pointer - is the assignment of simple types and statically allocated records. In this case Exp may contain any number of qualified identifiers. We start by calling *AddQualifiedIdentifier* to create vertices and edges for Q_1, the qualified identifier on the left-hand-side. Then we invoke *AddQualifiedIdentifier* for each Q_2 in Exp. No more edges need to be created or modified because no references can be changed in this case.

The last function in the points-to analysis, *MakeLockSet*, is listed in Algorithm 12. This function is called after all statements in a micro transaction have been analyzed. The function traverses all vertices and adds string representations of the qualified identifiers that must be locked to the set *lockset*. Any vertex, $V \in vertices$, where $owner(V) = nil$ and $lock(V) = true$ is a protected variable and is added to *lockset*. A vertex V where $owner(V) \neq nil$ and $lock(V) = true$ is a record. If $new(V) = true$ then V represents a record that was allocated within the transaction and therefore does not need to be locked. If $new(V) \neq true$ then we use the *owner* function to make a string representation of the first qualified identifier that referenced the record and include this string in *lockset*. Finally, *lockset* is returned as the result.

Algorithm 12 The MakeLockSet algorithm of the points-to analysis.

```
Function MakeLockSet
lockset = ∅
for every V ∈ vertices do
  if V ≠ nil ∧ V ≠ undefined ∧ lock(V) = true then
    if owner(V) = nil then
      lockset ⇐ lockset ∪ {name(V)}
    else if new(V) = false then
      N = ''
      while owner(V) ≠ nil do
        E = (W1, W2) where E ∈ edges ∧ W2 = V ∧ W1 = owner(V)
        N = label(E)+ '.' +N
        V ⇐ owner(V)
      end while
      lockset ⇐ lockset ∪ {N}
    end if
  end if
end for
return lockset
```

When the points-to analysis of the assignment statement has completed $lockset_A$ contains a list of all the qualified identifiers that must be locked before executing the assignments in the micro transaction.

3.3.3 Algorithm for If-Do Statements

The *AnalyzeIfDo* function, listed in Algorithm 13, performs the points-to analysis of If-Do statements. The vertices and edges for every qualified identifier, Q, in the conditional expression, Exp, are constructed by calling *AddQualifiedIdentifier* for every Q. Then *MakeLockList* is invoked to generate the locks for Exp, $lockset_{Exp}$. The locks required for assignment statement A, $lockset_A$, are determined by calling *AnalyzeAssignment* and excluding the locks already in $lockset_{Exp}$. The locks for assignment statement B, $lockset_B$, are also determined by invoking *AnalyzeAssignment*. The three sets of locks are returned and the analysis is complete.

The algorithm for If-Do statements without the Else-Do part is similar to Algorithm 13 and is omitted for brevity.

Algorithm 13 The AnalyzeIfDo function.

Function AnalyzeIfDo ($F = ($ `If` Exp `DO` A `Else DO` $B)$)
$edges = \emptyset$
$edges' = \emptyset$
$vertices = \{nil, undefined\}$
$lockset_{Exp} = \emptyset$
for every qualified identifier Q in Exp **do**
  AddQualifiedIdentifier(Q)
end for
$lockset_{Exp}$ = MakeLockSet
$lockset_A$ = AnalyzeAssignment(A) $\backslash lockset_{Exp}$
$lockset_B$ = AnalyzeAssignment(B) $\backslash lockset_{Exp}$
return ($lockset_{Exp}, lockset_A, lockset_B$)

3.4 Transaction Semantics

The semantics for micro transactions can be described informally as: Whenever a thread enters a micro transaction execution continues as if all other threads were suspended during the transaction. After the transaction has completed execution continues as if all threads are resumed. A program is executed as if no more than one transaction is executed at any given time. While this is the semantics of micro transactions, many micro transactions that do not interfere with each other may be executed at the same time in practice.

One cannot talk about transactions without discussing the ACID properties. The ACID properties are associated with certain guarantees that apply to transactions. The guarantees that apply to micro transactions are described in the following sections.

3.4.1 Atomicity

The guarantee of atomicity in micro transactions is that either all assignments in a micro transaction are executed or none are executed. Since the protected variables in the transaction are locked during the transaction, no other updates can be performed on the variables. This means that the transaction logically appears as a single atomic operation.

3.4.2 Consistency

The guarantee of consistency means that once a transaction has locked all of the necessary locks the transaction can execute uninterrupted. This means that

the programmer has a convenient programming model where a series of updates can be performed without regard to race conditions. This ensures that all of the assignments in a transaction are performed in the order specified, effectively guaranteeing consistency. Naturally, the compiler cannot guarantee that the updates specified by the programmer are correct and results in a consistent data model from the programmer's point of view. What *is* guaranteed is that the updates specified are performed correctly.

3.4.3 Isolation

The guarantee of isolation means that during a micro transaction there is a consistent view of the system *within* the transaction. The guarantee of isolation applies only within transactions which means that any expression using protected variables outside of a transaction may see an inconsistent view of the system at some point in the execution. Protected variables are only locked when written to and not when read from. This is a design decision which was made to reduce the overhead of locking.

3.4.4 Durability

The guarantee of durability means that after a transaction has completed all of the updates are committed and will not be undone or revoked at a later point in execution.

3.5 Limitations of the Current Implementation

Vex does not have exception handling or runtime error handling. If a runtime error or an exception occurs during a transaction then part of the transaction might be committed while another part is not. Furthermore none of the locks acquired are released which will most likely cause the program to deadlock or livelock. This guarantee of atomicity only holds when no exceptions or runtime errors occur.

By-reference parameters cannot be used. This is a minor limitation which can be remedied with a little extra work. For every by-reference parameter the compiler could add an extra parameter to pass a lock variable when a protected variable is passed by-reference. This would allow functions, procedures and threads to lock protected variables correctly when used as by-reference parameters.

Functions, nested 'if' statements and iteration cannot be used within transactions. This imposes some restrictions on the expressiveness of the programmer,

as some algorithms are not easily written with these restrictions.

There is a risk of livelock in Vex programs because the set of locks in a program is a not totally ordered set. The risk is reduced by sorting some of the locks. The risk can be further reduced by employing a truncated binary exponential backoff scheme like the CSMA/CD scheme used in the IEEE 802.3 ethernet protocol [2].

There is no transaction manager in Vex which means that when multiple threads are trying to acquire the same lock there is no guarantee that a given thread will acquire the lock within a given deadline (or within a given number of retries). This means that there is a risk of starvation.

You cannot write If-Do-***Else-If-Do*** statements. Removing this limitation is not complicated and the only reasons for not doing so is the time frame for this thesis and because it was not an essential feature.

If-Do statements cannot be nested. This limits the expressiveness of the programmer - but not the language. The lack of nested If-Do statements makes it more challenging, but not impossible, to write some algorithms.

3.6 Summary

This chapter has described the details about micro transactions. Protected variables were defined and a method of how to identify them was presented. A points-to analysis which identifies the protected variables that need to be locked in a given transaction was presented. Deadlock-free locking protocols for micro transactions were presented. And finally, some of the limitations of the implementation were discussed.

The Compiler

A two stage compiling process is used to compile a Vex program. Stage one translates a Vex program into a Delphi unit. The Delphi unit is used in a Delphi host program which is compiled into a binary executable by the Delphi 2007 compiler. The executable as well as the two compilers run on an Intel compatible processor with Windows XP. An overview of the compiling process is illustrated in Figure 4.1

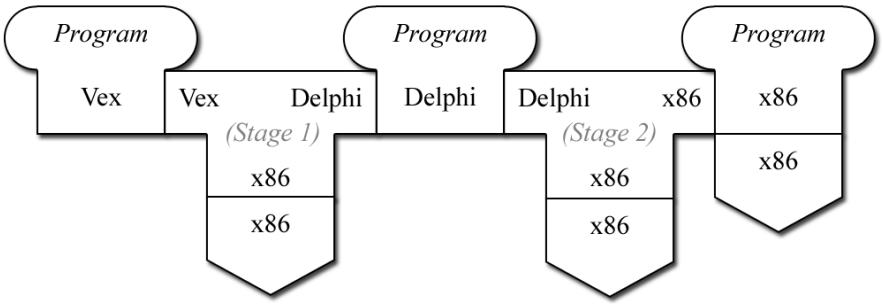


Figure 4.1: *The two stage compiling process used to compile a Vex program.*

A flow diagram of the two stage compiling process used to compile a Vex program is illustrated in Figure 4.2. Stage one translates a Vex program into a Delphi unit. First the GOLD parser is used to construct an abstract syntax tree (AST). The AST is used for type checking where the AST is decorated, adding type information to nodes in the AST. The decorated AST is used to generate Delphi code and a .Pas file is the output of stage 1. In stage two the .Pas file is compiled as a unit in a Delphi project (the .Dpr file). This Delphi project is the host program which is compiled into a binary executable by the Delphi 2007 compiler. The GOLD parser may report syntax errors if the program is not correct. The type checking process may also report errors if the program is not correct. The Delphi compiler may report an error if the .exe file cannot be overwritten.

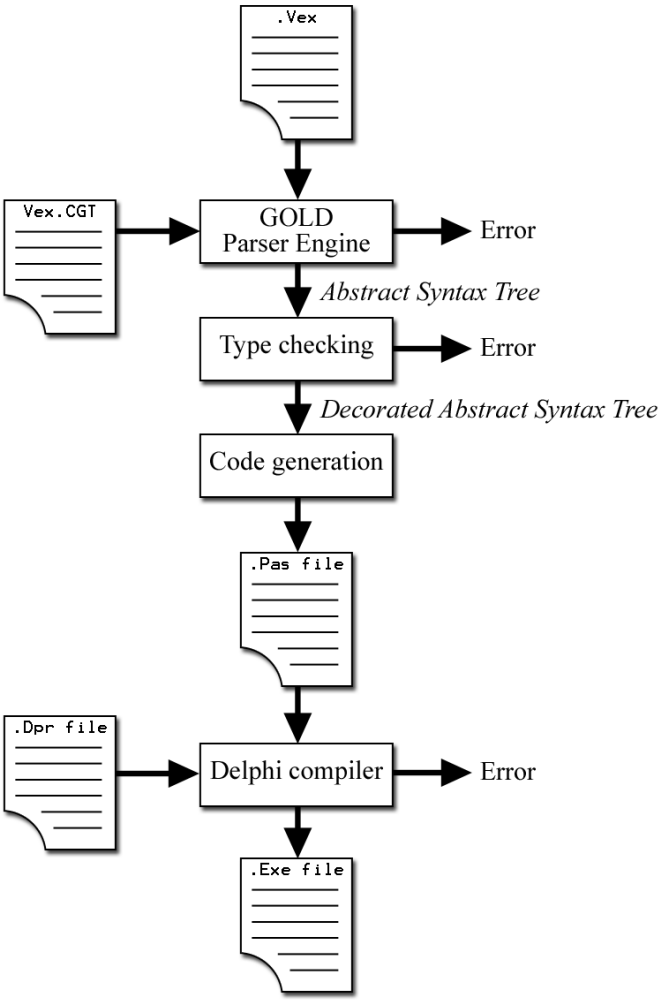


Figure 4.2: *Flow diagram of the two stage compiling process.*

An important Delphi feature is the ability to use inline assembler. This allows the Vex compiler to insert assembler instructions and use the atomic compare-and-swap instruction `CMPXCHG`[5]. When the `CMPXCHG` instruction is preceded by the `LOCK` directive the instruction is atomic across all cores in the CPU.

The target platform for the Delphi compiler is Windows XP on an Intel Architecture 32 bit (IA32) processor. IA32 processors are also popularly known as x86 processors.

4.1 Parsing

The GOLD Parsing System [1] is used for parsing in the Vex compiler. The Vex grammar was written in the 'GOLD Parser Builder' application. This application generates a compiled grammar table (CGT) file from a grammar file (GRM). This file contains LALR(1)[1] and DFA[2] parsing tables for parsing a Vex program. The CGT file needs only to be generated when the grammar is changed.

The Vex compiler uses the 'GOLD Parser Engine' for Delphi by Alexandre Rai. The parser engine uses the CGT file to scan and parse a Vex program. The engine uses a DFA to scan the Vex program and generate tokens. The tokens are used by a LALR(1) parser to generate an AST for the Vex program. The dataflow for the GOLD Parsing System is illustrated in Figure 4.3.

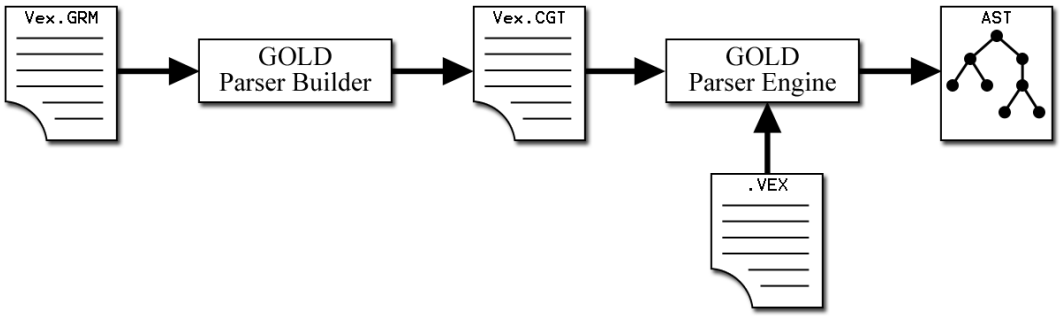


Figure 4.3: *The dataflow in the GOLD parsing System.*

4.2 Type Checking

The AST is traversed and type checked in a recursive descent. The AST is decorated with type information. Coercion is performed and the coerced nodes

[1]Look-Ahead, Left-to-right, Rightmost-derivation parser with a look-ahead of 1 token
[2]Deterministic Finite Automaton

in the AST are annotated with the appropriate type information. Protected variables are identified and marked as such in the symbol table.

Assignment nodes in the AST are decorated with a list of the identifiers on the right-hand-side of the assignment and with the identifier on the left-hand-side of the assignment. If-Do-Else-Do nodes are decorated with a list of the identifiers used in the conditional expression.

The list of identifiers used in an assignment node is searched for protected variables. If the list contains any protected variables and the assignment node references any functions then an error is reported and compilation is terminated.

Finally, a points-to analysis is performed in each of the If-Do-Else-Do and assignment nodes in the AST.

4.3 Code Generation

Since Vex is very similar to Delphi most of the code generation is straightforward. The most interesting aspect of the code generation is the inline assembler that acquires and releases the locks of protected variables.

4.3.1 Acquiring and Releasing Locks

Delphi allows writing assembler blocks within statement blocks. This feature makes is easy to use the CMPXCHG instruction in Delphi. Delphi symbols can be used within the assembler blocks which simplifies the task even more.

The CMPXCHG[3] assembler instruction is used to acquire a lock. It is a Compare-And-Swap (CAS) instruction in the Intel Architecture 32 processors. The details about the instruction can be found in 'Intel 64 and IA-32 Architectures Software Developers Manual - Volume 2A: Instruction Set Reference, A-M'[5]. After acquiring a lock the memory address of the lock is saved on the stack.

Since the memory address of the lock variable is stored on the stack, releasing a lock a simple matter of reading the address from the stack and writing a value of 0 to the memory address.

Jump instructions use labels as jump targets. A label is an alphanumeric symbol followed by a colon. A local label is prefixed with '@@' and is only reachable by jump instructions inside the same assembler block. Global labels must be declared prior to use in Delphi.

[3]CMPXCHG is a mnemonic for CoMPare-and-eXCHanGe

Algorithm 14 The Foo procedure. The declaration of a and b has been included for completeness.

```
Var
   a, b : Integer;
Procedure Foo;
Begin
   a := a + 100 &
   b := b -100;
End;
```

4.3.2 Assignment Statements

A simple Vex procedure `Foo` is shown in Algorithm 14. The code generated for this procedure is shown in Algorithm 15. Some assembler instructions have been compacted s.t. two instructions appear on the same line, separated by a semicolon, to conserve space[4].

In Algorithm 15 the two lock variables, `a_LOCK` and `b_LOCK`, have been added by the compiler. The micro transaction has been enclosed in a begin-end block. `GetCurrentThreadID` is called and the result is stored in the `edx` register. We try to acquire the first lock, `a_LOCK`, in lines 10 and 11. If successful then we jump to `@@lock_2`, if not then we jump to `@@start` and retry. After `@@lock_2` the address of the first lock is pushed onto the stack and we try to acquire the second lock, `b_LOCK`. If successful then we jump to `@@end` and push the address of the second lock onto the stack. If the second lock cannot be acquired then we jump to `@@unlock_1` where the first lock is released and we jump to `@@start` to retry.

When both locks have been acquired the execution phase of the transaction can start and lines 28 and 29 are executed. Finally, the two locks are released. Note that when an address is pushed onto the stack an additional value is pushed. The second value is used to signal whether the lock was already acquired by the current thread. If so, it will only be unlocked once during the unlocking phase. This is relevant when locking multiple records through pointers because the points-to analysis cannot always identify whether these records are in fact the same record.

4.3.3 If-Do Statements

A simple Vex procedure `Bar` is shown in Algorithm 16. The code generated for this procedure is shown in Algorithm 17. The assembler instructions which

[4]This is legal Delphi syntax

Algorithm 15 The code generated for the Foo procedure. Two assembler blocks have been inserted and two locks variables have been declared.

```
 1: Var
 2:     a, b : Integer;
 3:     a_LOCK, b_LOCK : Integer;
 4: Procedure Foo();
 5: Begin
 6:     Begin {Transaction}
 7:         Asm {Locking phase}
 8:             CALL GetCurrentThreadID; MOV edx, eax
 9:         @@start:
10:             LEA ecx, a_LOCK; MOV eax, 0
11:             LOCK CMPXCHG [ecx], edx; JZ @@lock_2
12:             CMP eax, edx; JNZ @@start
13:         @@lock_2:
14:             PUSH ecx; PUSH eax
15:             LEA ecx, b_LOCK; MOV eax, 0
16:             LOCK CMPXCHG [ecx], edx; JZ @@end
17:             CMP eax, edx; JNZ @@unlock_1
18:             JMP @@end
19:         @@unlock_1:
20:             POP ecx; POP eax
21:             CMP ecx, 0; JNZ @@skip_2
22:             MOV [eax], 0
23:         @@skip_2:
24:             JMP @@start
25:         @@end:
26:             PUSH ecx; PUSH eax
27:         End; {Locking phase}
28:         a := a + 100;
29:         b := b - 100; {Unlocking phase}
30:         Asm
31:             POP ecx; POP eax
32:             CMP ecx, 0; JNZ @@skip_2
33:             MOV [eax], 0
34:         @@skip_2:
35:             POP ecx; POP eax;
36:             CMP ecx, 0; JNZ @@skip_1
37:             MOV [eax], 0
38:         @@skip_1:
39:         End; {Unlocking phase}
40:     End; {Transaction}
41: End;
```

Algorithm 16 The Bar procedure.

```
Procedure Bar;
Begin
If b > 100 Do
   a := a + 100 &
   b := b -100
Else Do
   a := a + 100 &
   c := c -100;
End;
```

are used for locking and unlocking have been removed to conserve space. The removed locking and unlocking code is similar to that in Algorithm 15.

The first thing to notice in Algorithm 17 is the declaration of five labels. The names of the labels are postfixed with '_T0' because this code is Transaction 0 (the first transaction in the program) and because all label names must be globally unique. These labels are jump targets which the program jumps to, for example, when a lock cannot be acquired.

Execution of the transaction is as follows:

The locks for the conditional expression are acquired (line 6). If the locks cannot be acquired then execution retries immediately (busy-wait). In this example there is only one lock that must be acquired - the lock for variable *b*.

When the lock for variable *b* has been acquired, the conditional expression is evaluated (line 7). If the expression evaluated to *true* then execution continues with line 9 where the lock for the assignment statement is acquired. The assignment statement only requires one additional lock (on variable *a*) since *b* is already locked. If the lock on *a* is not successfully acquired then a jump to *fail1_T0:* is executed where the lock on *b* is released before jumping to *restart_T0:* (line 6) and restarting (busy-wait). If the lock on *a* is successfully acquired then the assignments in line 10 and 11 are executed and the lock on *a* is released (line 13) before jumping to *done_T0:* in line 14. After jumping to *done_T0:* the lock on *b* is released (line 35) and the transaction in completed.

If the expression in line 7 evaluated to *false* then execution continues with line 22, where the locks for *a* and *c* are acquired. If unsuccessful then a jump to *fail2_T0:* is executed and the locks on *a* (if acquired) and *b* are released (line 29) before jumping to *restart_T0:* (line 6) and restarting (busy-wait). If the locks are successfully acquired in line 22 then the assignments in line 23 and 24 are executed before releasing the locks on *a* and *c* (line 26) and jumping to *done_T0:* (line 27) where the lock on *b* is released.

By looking at Algorithm 16 and Algorithm 17 it looks as though the code

Algorithm 17 The compiled Bar procedure.

```
 1: Procedure Bar();
 2: Label
 3:     restart_T0, fail1_T0, fail2_T0, done_T0;
 4: Begin
 5:     Begin {Transaction}
 6:     Asm restart_T0: ... {Lock b} ... End;
 7:     If b > 100 Then
 8:     Begin
 9:         Asm {Lock a} ... JMP fail1_T0; ... End;
10:         a := a + 100;
11:         b := b - 100;
12:         Asm
13:             {Unlock a}
14:             JMP done_T0
15:           fail1_T0:
16:             {Unlock b}
17:             JMP restart_T0
18:         end;
19:     End
20:     Else
21:     Begin
22:         Asm {Lock a and c} ... JMP fail2_T0; ... End;
23:         a := a + 100;
24:         c := c - 100;
25:         Asm
26:             {Unlock a and c}
27:             JMP done_T0 ;
28:           fail2_T0:
29:             {Unlock a and b}
30:             JMP restart_T0 ;
31:         end;
32:     End;
33:     Asm
34:       done_T0:
35:         {Unlock b}
36:     End;
37:     End; {Transaction}
38: End;
```

required for a simple transaction is somewhat extensive - even with the assembler instructions removed. But in practice the assembler statements do not account for many bytes in the final program file.

4.4 Summary

This chapter has elaborated on some of the more interesting parts of the compiler. From parsing, using the GOLD Parsing System, to type checking. The use of the CMPXCHG instruction and examples of locking protocols for micro transactions were presented.

Experiments

The experiments were performed on a PC with the following specifications: One 2.4 GHz Intel Core 2 Duo E6600 Pentium processor with 2 GB of DDR2 RAM on an ASUS P5B-Deluxe motherboard with a 1,066 MHz FSB[1]. The experiments were performed in Microsoft Windows XP with service pack 2.

The tested programs were compiled with the Codegear Delphi 2007 compiler (version 11) using the 'Release' settings. The release settings enable optimizations while disabling debug information and runtime checking in the compiled program.

High resolution timing is achieved by using the `QueryPerformanceCounter()` function (from 'kernel32.dll').

5.1 Test Scenarios

Two different producer/consumer scenarios involving high contention were tested using three different methods. In the first scenario one thread inserts elements into a queue while another removes elements from the queue. In the second scenario one thread is pushing elements onto a stack while another pops the elements from the stack. In each scenario 10,000,000 elements are inserted/pushed and 10,000,000 elements are removed/popped. Both the queue and the stack is implemented as a doubly-linked list of records with a maximum size of 10,000 elements.

The first method uses Vex to construct the two scenarios, the second uses a critical section with busy-wait synchronization and the third method uses critical sections with blocking/preempting synchronization. The Windows API[2] function `TryEnterCriticalSection()` is used to enter a critical section using busy-wait synchronization. `EnterCriticalSection()` from the Windows API enters a critical section with blocking synchronization. Both functions are found in 'kernel32.dll'.

[1] Front Side Bus
[2] Application Programming Interface

Two threads, one reader and one writer, are used in both scenarios. Only two threads are used because a dual core processor was used in the test environment. More than two threads would increase the frequency of context switching because the processor only has two cores.

The priority of the threads was set to `REALTIME_PRIORITY_CLASS` to reduce context switching. Context switching incurs an overhead and is undesirable when employing busy-waiting - it is the very thing we are trying to avoid. Each thread was bound to a separate core using the `SetThreadAffinityMask()` function ('kernel32.dll'). This ensures that the threads do not change core (or run on the same core) when the threads are preempted.

5.2 Results

The performance of each method in both of the two scenarios is listed in Table 5.1. The performance of Vex is greater (6.8 seconds) when using a queue as opposed to using a stack (8.6 seconds). This is due to higher contention on the `Tail` pointer which is used as the top of the stack, as both threads continuously try to manipulate this pointer while pushing and popping. When using a queue one thread primarily manipulates the `Head` pointer while the other thread primarily manipulates the `Tail` pointer. The maximum size of the doubly-linked list (10,000 elements) limits the performance of the queue, however. Adding elements to the queue when the queue is full results in busy-waiting.

	Queue	Stack
Vex	6.8 s.	8.6 s.
TryEnterCriticalSection	7.0 s.	6.2 s.
EnterCriticalSection	61.8 s.	60.5 s.

Table 5.1: *results*

`TryEnterCriticalSection` outperforms Vex overall. We interpret this as a result of using only a single lock - as opposed to Vex. Acquiring a single lock is evidently 14.3% faster on average than acquiring multiple locks in this experiment. Vex seems to be a little faster than `TryEnterCriticalSection` in the Queue scenario and this is probably because of the overhead associated with the function call (`TryEnterCriticalSection()`) into 'kernel32.dll'.

We can see that using `EnterCriticalSection` is comparatively slow. This is because `EnterCriticalSection` blocks the calling thread if another thread is in the critical section, resulting in a context switch. Both test scenarios involve high contention on the critical section which results in a high frequency of context switching. The running time in the two scenarios, 61.8 seconds and 60.5 seconds respectively, is a direct result of frequent context switching.

5.3 Summary

The experiments show that the automatic locking in Vex does not yield better performance than using manual optimization and critical sections. In the authors opinion, a 16.7% performance loss is, however, an acceptable performance loss given that you no longer need to manually ensure mutual exclusion.

The experiments show that the overhead of context switching can have a significant performance impact on small critical sections with high contention. Busy-waiting is a more efficient alternative in this case.

The few experiments here are only an indication of the performance of micro transactions. The two test scenarios do in no way represent a varied or complete test suite and are only indicative of the performance under very specific circumstances. Although the time frame did not allow for it, more experiments should be performed to have a detailed comparison of micro transactions and other approaches.

Chapter 6

Conclusions

Before concluding on the report, this paragraph presents a short summary of the report up to this point. In Chapter 1 the problem setting was introduced and the thesis about micro transactions was stated. The syntax of Vex, the Pascal based language which was used to experiment with micro transactions, was introduced in Chapter 2. The analysis of identifying variables that require locking was presented in Chapter 3. Followed by a description of locking protocols, points-to analysis and the informal semantics of micro transactions. The limitations of the implementation were described in the end of Chapter 3. The compiler was presented in Chapter 4. Specifically, the most interesting aspects of parsing, type checking and code generation. Chapter 5 summarized the results of the experiments that were performed to compare the performance of micro transactions with other methods.

The performance of the spin locks (busy-wait) can probably be improved by following the approach outlined in [10]. This approach includes spinning on a volatile read instead of spinning on the CMPXCHG instruction. Another improvement is the use of exponential back-off which can reduce the contention on locks and reduce this risk of livelock.

Some of the limitations associated with micro transactions requires the programmer to write workarounds in order to write correct algorithms. Most notably, that a transaction can only contain assignments and a single conditional expression. Other approaches, such as the atomic blocks in Software Transactional Memory[8], do not have such stringent limitations and, therefore, allow the programmer to be much more expressive.

On the other hand, micro transactions are unaffected by the problem with operations that cannot be undone during a rollback. Micro transactions are never rolled back and do not require log keeping which plays a large part in the overhead associated with most transactional memory methods.

The greatest problem with micro transactions is, in the authors opinion, the lack of compositionality. It is not possible to have transactions within transactions which greatly reduces the possibility of code reuse. This means that micro

transactions are not suitable for use with object oriented programming. And since the de facto programming model today is the object oriented programming model this means that micro transactions are of little consequence in the big picture.

As described in the conclusions of my previous work[14] there is a need for better exclusive access methods. A composable method is needed to solve the problems associated with writing concurrent object oriented programs. Micro transactions are not composable and, as such, do not provide any help in this matter.

6.1 Thesis

My thesis was this:

> *A compiler can, through static analysis, generate locks and the necessary locking protocols to ensure mutual exclusion in micro transactions. Micro transactions can make concurrent programming easier and less error-prone. Race conditions can be avoided using micro transactions. Micro transactions provide a programming model that is easy to comprehend and use.*

The first part of the thesis holds. A compiler can use a simple static analysis to identify where locks are needed, generate correct locking protocols and ensure mutual exclusion in micro transactions. Micro transactions can ease the writing of small or simple programming problems but it fails to alleviate the larger problem of concurrent programming: Compositionality. The programming model is easy to understand but using it is not as easy as expected. The lack of nested If-Do statements forces the programmer to write non-intuitive code that can justifiably be called workarounds.

While the thesis did not hold completely there are lessons to be learned from it. We must often try many different strategies before finding one that works. And finding one that doesn't work brings us a little closer to finding one that does.

6.2 Future Work

The correctness of the points-to analysis could be proved by a formal verification. Although this seems somewhat futile, as the applications of micro transactions seems limited due to the lack of compositionality.

It would be much more interesting to investigate if a compromise between the simplicity of micro transactions and the compositionality of software transactional memory could be found. Some degree of compositionality is necessary for micro transactions in order to integrate these with a modern object oriented language in a useful way.

Bibliography

[1] Gold parsing system, May 2008.
http://www.devincook.com/goldparser/.

[2] IEEE Std 802.3-2005. Ieee std 802.3 - 2005 part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications - section five. Technical report, IEEE, 2005.

[3] Channel 9. Programming in the age of concurrency: Software transactional memory, September 2006.
http://channel9.msdn.com/Showpost.aspx?postid=231495.

[4] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[5] Intel Corporation. Intel 64 and ia-32 architectures software developers manual - volume 2a: Instruction set reference, a-m, April 2008.
http://download.intel.com/design/processor/manuals/253666.pdf.

[6] Deryck F. Brown David A. Watt. *Programming language processors in Java : compilers and interpreters*. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2000.

[7] A. Shoshani E. G. Coffman Jr., M. J. Elphick. System deadlocks. *Computing Survey*, 3(2):12, June 1971.

[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[9] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions*

in Concurrent Object-Oriented Programming, pages 107–150. MIT Press, 1993.

[10] Mary R. Lee Michael Chynoweth. Implementing scalable atomic locks for multi-core intel em64t and ia32 architectures, August 2007. `http://softwarecommunity.intel.com/articles/eng/2807.htm`.

[11] SUN Microsystems. Rock's transactional memory, September 2008. `http://blogs.sun.com/HPC/entry/video_transactional_memory_on_rock`.

[12] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[13] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2), march 1953.

[14] Jesper B. Christensen Simon H. Thøgersen. Concurrency models - processes as an alternative to threads. Technical report, Aalborg Universitet, January 2008. `https://services.cs.aau.dk/public/tools/library/files/rapbibfiles1/1199714915.pdf`.

[15] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30 (3), 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

Syntax Definition

The grammar of Vex was written in the GOLD Parser Builder, an application in the GOLD Parsing System. The syntax used in the definition of the grammar is a variation of BNF. The Vex grammar file is as follows:

```
"Name"     = 'Vex'
"Author"   = 'Jesper Christensen'
"Version"  = '1.0'
"About"    = 'Verbose Experimental Concurrent Language'

"Case Sensitive" = 'False'
"Start Symbol" = <Program>
! ------------------------------------------------- Sets

{ID Head}      = {Letter} + [_]
{ID Tail}      = {Alphanumeric} + [_]
{String Chars} = {Printable} + {HT} - ['']

! ------------------------------------------------- Terminals

Identifier     = {ID Head}{ID Tail}*
CharLiteral    = (''{Printable}'')
               | ('#'{Number}{Number}*)
StringLiteral  = '' {String Chars}* ''
IntegerLiteral = {Number}{Number}*
FloatLiteral   = {Number}{Number}*[.]{Number}{Number}*

Comment Line  =  '//'
Comment Start =  '{' | '(*'
Comment End   =  '}' | '*)'

! ------------------------------------------------- Rules

<Program>     ::= 'program' identifier ';' <Declarationlist> <Stm block> '.'

!!!!!!!!!!!!!!!!!!!!!!!! declarations !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

<Declarationlist> ::= <Declaration> <Declarationlist>
```

```
                  |

<Declaration> ::= 'Var' <Var decl list>
                |  'Type' <Type decl list>
                |  <ProcDecl>
                |  <FuncDecl>

<ProcDecl> ::= 'procedure' Identifier <formal parameters> ';' <Declarationlist> <Stm block> ';'

<FuncDecl> ::= 'function' Identifier <formal parameters> ':' <Type> ';' <Declarationlist> <Stm block>


<Var decl list> ::= <Var decl> ';'
                  |  <Var decl> ';' <Var decl list>

<Var decl> ::= <Identifierlist> ':' <Type>

<Identifierlist> ::= Identifier
                   |  Identifier ',' <Identifierlist>

<Type decl list> ::= <Type decl> ';'
                   |  <Type decl> ';' <Type decl list>

<Type decl> ::= Identifier '=' <Type>
              |  Identifier '=' <Type> '^'
              |  Identifier '=' 'record' <Var decl list> 'end'
              |  Identifier '=' 'thread' <formal parameters> <Declarationlist> <Stm block>

<Type> ::= Identifier

<Formal parameters> ::= '(' <Par decl list> ')'
                      |

<Par decl list> ::= <Var decl>
                  |  <Var decl> ';' <Var decl list>

!!!!!!!!!!!!!!!!!!!!!!!! Statements !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

<Stm list>    ::= <Statement> ';'
                |  <Statement> ';' <Stm list>

<Statement>   ::= <Assign stm>
                |  <Stm block>
                |  <Call stm>
                |  <Free stm>
                |  <If stm>
                |  <Ifdo stm>
                |  <Spawn stm>
                |  <For stm>
                |  <While stm>
                |  <Repeat stm>
```

```
<If stm>      ::= 'if' <Expression> 'then' <Stm block>
               |  'if' <Expression> 'then' <Stm block> 'else' <Statement>

<Ifdo stm>    ::= 'if' <Expression> 'do' <Assign stm>
               |  'if' <Expression> 'do' <Assign stm> 'else' 'do' <Assign stm>

<Stm block>   ::= 'begin' <Stm list> 'end'

<Assign stm>  ::= <Assignment>
               |  <Assignment> '&' <Assign stm>

<Assignment>  ::= <Qualified identifier> ':=' <Expression>
               |  <Qualified identifier> ':=' '@' <Qualified identifier>
               |  <Qualified identifier> ':=' 'new' <Type>
               |  <Qualified identifier> '<=>' <Qualified identifier>

<Free stm>    ::= 'free' <Qualified identifier>

<Call stm>    ::= Identifier
               |  Identifier '(' <Expression List> ')'

<For stm>    ::= 'for' Identifier ':=' <Expression> 'to' <Expression> 'do' <Statement>
               | 'for' Identifier ':=' <Expression> 'downto' <Expression> 'do' <Statement>

<While stm>    ::= 'while' <Expression> 'do' <Statement>

<Repeat stm>    ::= 'repeat' <Stm list> 'until' <Expression>

<Spawn stm>    ::= 'spawn' <Type>
               |   'spawn' <Type> '(' <Expression List> ')'


!!!!!!!!!!!!!!!!!!!!!! Expression !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

<Expression>  ::= <Expression> <Rel op> <Add Exp>
               |  <Add Exp>

<Rel op>      ::= '>' | '<' | '<=' | '>=' | '=' | '<>'

<Add Exp>     ::= <Add Exp> <Add Op> <Mult Exp>
               |  <Mult Exp>

<Add Op>      ::= '+' | '-' | 'OR' | 'XOR'

<Mult Exp>    ::= <Mult Exp> <Mult Op> <Negate Exp>
               |  <Negate Exp>

<Mult Op>      ::= '*' | '/' | 'DIV' | 'MOD' | 'AND'

<Negate Exp>  ::= '-' <Exp Exp>
```

```
               |  <Exp Exp>

<Exp Exp>      ::= <Value> '**' <Value>
               |  <Value>

<Func Call>    ::= Identifier '(' <Expression List> ')'

<Expression List>  ::= <Expression>
                    |  <Expression> ',' <Expression List>

<Qualified identifier> ::= Identifier
                         |  Identifier '^'
                         |  Identifier '.' <Qualified identifier>
                         |  Identifier '^' '.' <Qualified identifier>

<Value>        ::= <Qualified identifier>
               |  <Func Call>
               |  '(' <Expression> ')'
               |  CharLiteral
               |  StringLiteral
               |  IntegerLiteral
               |  FloatLiteral
               | 'NOT' <Value>
```