Static Dependency Analysis of Recursive Structures for Parallelisation

Tim Lewis

Doctor of Philosophy
Institute of Computing Systems Architecture
Division of Informatics
University of Edinburgh
2004

Abstract

A new approach is presented for the analysis of dependencies in complex recursive data structures with pointer linkages that can be determined statically, at compile time.

The pointer linkages in a data structure are initially described in the form of two-variable Finite State Automata (2FSA) which supplements the code that operates over the data structure. Some flexibility is possible in that the linkages can specify a number of possible target nodes for a pointer.

An analysis method is described to extract data dependency information, also in the form of 2FSAs, from the recursive code that operates over these structures. For restricted, simple forms of recursion these data dependencies are exact; but, in general, heuristics are presented which provide approximate information. This method uses a novel technique that approximates the transitive closure of these 2FSA relations. In the context of dependency analysis, approximations must be *safe* in that they are permitted to overestimate dependencies thereby including spurious ones, but none must be missed. A test is developed that permits the safety of these approximate solutions to be validated.

When a recursive program is partitioned into a number of separate threads by the programmer this dependency information can be used to synchronise the access to the recursive structure. This ensures that the threads execute correctly in parallel, enabling a multithreaded version of the code to be constructed.

A multithreaded Micronet processor architecture was chosen as a target for this approach. Front- and back-ends of a compiler were developed to compile these multithreaded programs into executables for this architecture, which were then run on a simulation of the processor. The timing results for selected benchmarks are used to demonstrate that useful parallelism can be extracted.

Acknowledgements

This thesis would not have been possible without assistance from:

- My supervisor, Arvind.
- My wife, Ruth, who always inspires me.
- Alan Stanley, who spent much time getting the microthreaded processor simulator working with my code.
- Shi Zhong, who helped with SUIF compiler related stuff.
- Also the following, who have made a significant, albeit unwitting, contribution.
  - The designers of the 'daVinci' [1] graph visualisation package, which typeset all of the 2FSA diagrams in this thesis. This package is now available from `http://www.davinci-presenter.de/`.
  - Derek Holt, writer of the *kbmag* package, which my 2FSA implementation is built upon.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise, and that this work has not been submitted for any other degree or professional qualification.

Early versions of the material in Chapters 3, 4 and 5 appears in the publications [2], [3] and [4].

(*T. A. Lewis*)

Table of Contents

Chapter 1

Introduction

Parallelism is a factor which influences the performance of computer systems ranging from distributed systems of computers linked over the Internet to thread level parallelism in single-chip multiprocessors. Examples include:

- Thousands of loosely-coupled machines linked by the Internet, such as the SETI@home project [5], which searches for extraterrestrial intelligence by analysing radio telescope data, and distributed.net [6] which concentrates on solving mathematical and cryptographic problems.
- Dedicated supercomputers such as the Intel Paragon or Cray T3D. These machines have multiple processors connected via high performance communication interfaces.
- Individual CPUs that allow many different instructions to be executed concurrently. These include multithreaded Micronet [7] designs that we target in this thesis, and HyperThreaded Intel Xeon architectures.

In a single threaded multi-processor machine there are a number of processors, each running a single threaded computation. There will be an interconnect to shared memory, and perhaps some local memory for each processor.

On the other hand a multi-threaded, single processor consists of one piece of silicon on which a number of processor cores are placed. The coupling between these processors (*Thread Processing Units*) is much tighter. The time to create new threads will be less than the time to instantiate a fresh process on a multi-processor machine. Thus dynamic computations where threads are created and destroyed will be much better supported. Communications between threads will also be much faster than between processes on separate processors. This will enable concurrent tasks that have much greater cooperation requirements to be run efficiently. Thus multi-processor machines tend to favour static numbers of concurrent tasks with little communication, whereas multi-threaded processors cope better with irregular numbers of tasks that communicate more frequently.

This work is oriented towards multithreaded processors. The goal will be to test our parallelism on a simulator of a Micronet architecture, which is described more fully in Chapter 6. However, the kinds of program analysis techniques outlined in this thesis have much wider applicability.

Parallelism is chiefly pursued to enable programs to run faster. As a side-effect it may also increase the size of problems that can be solved in a reasonable time. A parallelised version of a sequential program splits it into a number of parallel tasks which are very likely to require communication with each other.

The ratio of the execution times for the parallel version of a program and the sequential one is known as the *speedup*. In an ideal world, from the perspective of the programmer, a program split into two concurrent sub tasks should exhibit a speedup of two; however this is unlikely in practice. In general two issues will conspire to prevent this; overhead associated with communication and the unbalanced splitting of tasks.

The amount of communication between tasks will depend on the ingenuity with which the program has be sub divided. Also, the communication delay between the concurrent elements of these architectures (PCs, processors or thread execution units) varies hugely. Matching the communication load of a parallel algorithm with the communication delay of the target architecture is necessary to make the parallelisation efficient. This is a difficult problem.

1.1 Achieving Parallelism

There are many different approaches to producing parallel programs. One perspective divides them into the following two basic ones:

- Designing parallel languages that allow the programmer to express the parallelism in the problem in a natural and flexible way.
- Designing automatic parallelising compilers/code analysers that take existing sequential codes and produce a parallel version by exposing the available parallelism.

This thesis generally follows the second approach, but there is an element of the first. We will be concerned mainly in the automatic parallelisation of sequential programs, but there are two aspects in which we require the bare sequential code to be supplemented by extra information.

Firstly the sequential program will be enhanced with additional descriptions of the pointer data structures that it uses. This additional information is part of the specification of the problem, and normally exist only implicitly in the code written to initialise pointer relationships within the structures. This information does not require any explicit consideration of potential parallelisation in the program.

Secondly, the techniques developed will require a predefined partitioning of the program into separate threads. The methods developed in this thesis will automate the process of synchronising those threads.

Techniques for automatic parallelisation can be performed either at compile time or at runtime:

- *Compile Time or Static Analysis.* These techniques are run at compilation time and only rely on information directly available from the source code of the application.
- *Runtime or Dynamic Approaches* These aim to use information that can be collected at run-time to improve its parallel performance. Such techniques include load balancing, dynamic process creation and dynamic scheduling.

This thesis will be concerned with static analysis techniques, which can only rely on information provided to the compiler by the programmer. No run-time information can be assumed such as the size of the data structures or the values of the data within them. Such analysis can be improved if some restrictions can be placed on such values, and in some senses static and dynamic techniques can be considered independently. Static techniques can always be assisted by the use of additional information available at run-time, but they have the corresponding benefit that any information derived will apply to any specific run of the code, irrespective of the input data.

Nearly all static analyses for the purposes of parallelisation will require *dependency analysis*. One point in a program is dependent on another if they both access the same memory location, and at least one of them writes to it. Dependency analysis aims to find out which parts of a program are dependent on each other. Although dependencies are between program points and not parts of a data structure, we will often discuss the dependency analysis of a particular structure. This is just convenient shorthand for the analysis of linkages within a structure and the programs that access it.

1.2 Recursive Structures

A *recursive structure* is a data structure that contains pointers (or references) to other instances of itself. In other words, its *definition* is recursive. As an example, a tree structure in which each node contains two pointers to left and right subtrees is recursive. Such structures are created dynamically, each node is allocated separately and the whole structure can vary in size. To terminate the structure at the foot or *leaf* of the tree, the left and right pointers do not point to allocated trees but instead they are set to the *null* value. This description has been given in terms of languages such as C with pointers and dynamically allocated memory,[1] but their equivalent

[1] The terms *pointer-based structure* or just *pointer structure* are common. We will use both.

exist in most programming languages. Recursive structures also exist in languages without pointers (such a Java), where the sub trees are stored as references. In this case references are really being used as a more disciplined form of pointers. Recursive structures are also common in functional languages, such as ML or Lisp.

This thesis will be concerned with analysing programs that use recursive structures. All of the analysis will be relevant to any imperative languages that has support for recursive structures. However, we will generally use a 'C-like' terminology and framework for discussing and implementing our ideas. This is not meant to imply that these techniques are only relevant for these languages. Indeed, analogous techniques could be applied to functional languages, but this idea is not developed any further.

We will develop a system of structure specifications termed *2FSA descriptions* to store and manipulate the linkages between pointers in a structure. A 2FSA is an automaton that accepts pairs of strings of symbols, and is used to define a relationship between strings. This system gives an unique string of symbols (a *pathname*) to each node in a data structure. Determining whether a particular pointer links two nodes is reduced to asking whether the two pathnames are accepted by a particular 2FSA. These automata can be manipulated to answer a variety of questions about linkage properties and can also be extended to store dependency information for programs.

1.3 Automatic Parallelisation

Automatic parallelisation is a practical idea as there exists a large corpus of existing sequential code, and good sequential algorithms exist for many problems. Also sequential algorithms can be more natural to design and simpler to debug.

It has traditionally been the rôle of the compiler to hide the details of a particular computer architecture from the programmer. The 'Dragon book' [8] is a good introduction to the tasks that compilers have traditionally automated. For example, consider the issue of register usage. Some types of CPU have many more registers than others and compilers can take account of this and use whatever number is available without the programmer having to manually decide which data values in the programs are best stored in registers.

Hiding the details of the parallelism in an architecture is just an extension of this. However, automatic parallelisation is far more difficult because it requires the compiler to extract much more information from the code than is usually present. Restricting the algorithms in one way or another allows partial solutions to this problem to be explored.

Some other difficulties of this approach are:

- Decomposing a program into equal-sized tasks is difficult as it is hard to estimate the relative execution times of blocks of program code statically. These execution times

may vary dynamically and will be dependent on the actual hardware architecture.

- These tasks will (unless the programmer is particularly fortunate) have to communicate to synchronise concurrent sections of programs which are automatically synchronised in the sequential version. Deciding when and where to synchronise tasks to ensure correctness is one problem, while ensuring that this communication overhead does not kill any performance benefit. This is, of course, a factor influencing the original task selection.

- The synchronisation between parts of a sequential program is normally implicit and even the algorithm designer may be unaware that it is occurring. Automating the discovery of these points of synchronisation is hard. Even worse, sequential execution imposes dependencies that may not be necessary, which impedes the parallelisation process.

1.4 Dependency Analysis: Arrays versus Pointers

Parallelisation and dependency analysis of array codes has been a successful research area. Analysis of pointer based code is a very different proposition, dependency analysis can be solved tar more fully for array structures than pointer-based ones. The array problem is still open, however, as most techniques heavily restrict the way that arrays are used.

There is much overlap between array and pointer analysis. For example indexing arrays by using the values of another array (e.g. '`a[i] = b[c[i]]`') is generally explicitly prohibited in these approaches. These kind of constructs can be used to implement general pointer semantics, and therefore could be used to set up the kinds of structures that are considered in this thesis.

For loops iterating over arrays, there are many techniques that can find independent iterations and transform the loop into some parallel form. For code that uses dynamically-allocated, pointer-based structures, the situation is quite different, and more complex. The reason relates closely to the point made above about using arrays to index into another array. When used normally, an array is well understood statically, with the only run time variable being its size. The values are always laid out in the same order, and we know which value will be accessed when we look at index '`i+1`' or '`10-i`'.

However, as soon as a structure has a pointer to another structure there is a run-time dependent component to the layout of the values. When pointer structures are declared, there are no restrictions on which memory location certain pointers can be linked to. Thus a structure such as a binary tree with nodes in each level linked in a list (which we shall meet later in much more detail), cannot be differentiated from a graph structure that has nodes with three edges. The binary tree has a lot of *regularity*, *structure* or *pattern* within it, and this can be useful in any static analysis. In contrast a general graph may have almost none, and nothing can be done statically to determine data dependencies within it. Deciding whether any given two pointers

point to the same memory location is known as the *alias* problem. This is the same Pointer Data Structure Dependence Problem as described in [9].

Here, the fact that the structure may be understood statically is not captured by the description. The main theme of this thesis will be to tackle the resistance of pointer structures to static analysis, and how the analysis can be improved by utilising additional information about the structure of pointer linkages. One question which will be answered in this thesis is the following: by restricting the scope of pointer-based structures in some of the same manners as for arrays, can the potential for automatic parallelisation be improved?

1.5 Aims

To summarise, the aims of this thesis are as follows:

- Pursue the approach of using detailed pointer descriptions of data structures to permit more precise static dependency analysis of programs that update them.
- Use this dependency information in automating the parallelisation of these programs.
- Demonstrate the parallelism extracted by simulating the execution of these programs on a multithreaded machine.

1.6 Contributions

The main contributions of this thesis are:

- A 2FSA-based specification method that allows the description of complex linkage patterns in recursive structures.
- A method for extracting approximate, yet safe dependency information from a class of simple programs that recurse over these structures.
- A method of using such information to ensure that the structure accesses in a particular threaded version of the program will produce the same computation as the sequential version.
- Execution of these threaded programs on a simulated multi-threaded processor architecture. This includes compilation and object code generation for this architecture.

1.7 Publications

Sections of this work appears in the following publications:

- D. K. Arvind and T. Lewis, "Static analysis of recursive data structures," in *Languages and Compilers for Parallel Computing, 10th International Workshop LCPC'97* (Z. L. et al, ed.), no. 1366 in Lecture Notes in Computer Science, (Minneapolis, Minnesota, USA), pp. 423–426, Springer-Verlag, Aug. 1997.
- D. K. Arvind and T. Lewis, "Dependency analysis of recursive data structures using automatic groups," in *Languages and Compilers for Parallel Computing, 11th International Workshop LCPC'98* (S. C. et al, ed.), Lecture Notes in Computer Science, (Chapel Hill, North Carolina, USA), Springer-Verlag, Aug. 1998.

- D. K. Arvind and T. Lewis, "Safe approximation of data dependencies in pointer-based structures," in *Languages and Compilers for Parallel Computing, 13th International Workshop LCPC'00* (S. M. et al, ed.), no. 2017 in Lecture Notes in Computer Science, (Yorktown Heights, NY, USA), Springer-Verlag, aug 2001.

1.8 Thesis Structure

The rest of the thesis proceeds as follows.

Chapter 2 looks at the background to this problem and reviews some of the existing work in this area.

Chapter 3 describes our approach for the programmer to explicitly describe the linkages within the structures that the programs utilise. This chapter introduces the 2FSA approach which is the basis of this work. We introduce certain example structures used throughout this thesis to illustrate these ideas and show how to create the 2FSA descriptions for their pointers.

Chapter 4 takes a simple recursive language expressing the updating of such structures and shows how to generate dependency information for programs using 2FSA structures.

In Chapter 5 once the raw dependency information of a program has been extracted, we want to use this to simplify the process of constructing a parallel version. We follow the approach of requiring the programmer to partition the program into a suitable set of threads, and then automating the process of ensuring that they run correctly by updating the structure in the same, correct order as the sequential version. This chapter looks at how to simplify the complex dependency information that is generated, to produce the locking schemes required to synchronise the threads.

Chapter 6 reports the performance speedups by comparing the parallel implementation of our example programs with their sequential versions. This chapter describes the process of compiling these codes for a research multithreaded architecture and running them on an instruction level simulator. We also follow a tracing process that instruments these sequential programs to produce traces that document the dependencies. This estimates the potential parallelism within the particular partitioning scheme, and is used to act as a yardstick to compare the speedups we obtain from simulation.

In Chapter 7 we conclude, also relating some possible future research directions.

Chapter 2

Background and Related Work

Tackling dependency analysis in the presence of pointer based structures is a difficult problem. There are a number of different relevant research fields, this chapter aims to provide the background and cover related work.

2.1 Applications

Pointer-based data structures are an important implementation technique. There are many applications that use the kind of pointer structures we consider in this thesis. For many of these applications the problem size can scale arbitrarily depending on the spatial or temporal resolution of the simulations. The size of the data structures used to implement them and the amount of computation required to manipulate them are likewise unbounded. This means that improving performance by exploiting any parallelism within them is an essential consideration.

The literature contains a wide variety of problems that fall into this group. The simulation of semiconductor devices can use structured three dimensional meshes [10] and variable resolution rectangular meshes [11]. Neural networks are implemented as complex linked structures wth the aim of being parallelised [12]. The ICASE project [13] uses multithreaded architectures and mesh structures in aeronautics simulations. These examples all use sophisticated data structures with complex linkages. It is the linkages in the structures used by these applications that we will aim to capture and exploit in this thesis.

A large class of related applications are *n-body problems*, these deal with the situation where a large number of particles occupies a location, with interaction forces between each pair of particles. This occurs in astronomy, where stars and planets exert gravitational forces on each other, and electrodynamics, where fundamental particles move under the force of electromagnetic charge. These areas pose a problem for solving them computationally, since the forces act between each pair of particles, the algorithms scale as $O(n^2)$, where n is the number of particles. For large simulations this is impractical. Barnes-Hut [14] is an approach for solu-

tion of N-body interaction codes that gives an $O(n.\log(n))$ complexity. It does this by building up a tree of particles, with the particle data stored at the leaves, which are linked. Nodes internal to the structure hold summary information for particles grouped in nearby neighbourhoods. This is a compound data structure with complex linkages that is not specified as either tree or list structure. Barnes-Hut is an approximate technique, each particle need only consider interactions with local particles, which are easily found by traversing the tree. These traversals are often best expressed as recursive functions over the tree.

Traditional fluid flow simulations store and manipulate the flow as a field of vectors indicating the direction and strength of flow at each point. These approaches would use a multidimensional array of vectors to store and manipulate the vector field. Existing array based parallelisation techniques are ideal for these problems. Vortex methods [15], however, treat the flow as if it were a composition of a finite set of vortices at discrete positions in the space. Flow simulations using vortex methods such as the Fast Parallel Particle Algorithms are studied in [16].

These applications suggest that there is a strong need for automatic parallelisation techniques that can cope with complex variable-sized pointer structures and recursively expressed algorithms. We will next introduce the area of dependency analysis which is one of the techniques necessary to achieve this.

2.2 Dependency Analysis

When a program in a high-level language such as C is compiled into executable code a simple statement in the language, such as a assignment to a variable or a function call is converted into a number of machine language instructions. Each of these may be executed a number of times when the program is run. Different instructions in an executing program access (read or write) different parts of the memory of the machine.

Dependency analysis of programs aims to find when two sections of code access the same memory location and at least one writes to it. Such sections are called *dependent* or are said to be in *conflict*.

Extracting parallelism is a common goal of dependency analysis. At one extreme massively parallel supercomputing can benefit from this kind of analysis in [17]. Since parallelising techniques typically split sequential programs and the data they access into separate parts, it is essential to know which sections of program access the same sections of data. This information can be used either to verify that the concurrent sub programs access completely disjoint data, or insert communication between them to ensure that the shared data is kept consistent with the original intentions of the sequential program. The focus of this thesis is in deriving dependency information to assist parallelisation techniques.

However, there are other applications for dependency information that do not involve parallelisation. The term *dead code* is used for sections of programs (typically function bodies, or conditional blocks) that are never executed. Although these are typically the result of programmer error, some code transformation techniques can introduce such anomalies. Similarly *dead data* is data that is never read, although it may have been created or written to. Dependency analysis can be used to eliminate dead data in the presence of recursive structures [18], [19]. Thus the dependency derivation techniques of this thesis have wider application than just for automatic parallelisation.

We will next have a general overview of the kind of dependency analysis techniques that have been developed.

2.2.1 Dataflow Analysis

Dataflow analysis [20] aims to locate, for each read from a memory location the corresponding write (the *source*), that produced that value. This builds up a picture of the flow of data from one part of program to another, and gives strong indications of where parallelism might occur. These techniques have been used in a number of places in general dependency analyses and for pointer-based structures.

Formal Language theory [21] permits the abstract manipulation of sets of words. Often language formalism is employed to handle the dataflow information. Algebraic transductions are used in [22, 23] to store and manipulate the data-flow information, which is applied to programs analysing tree structures. In [24] dependency analysis for trees and arrays is integrated, using transducers as a common mechanism. This work has not been extended to more complex pointer structures.

A large class of dataflow analysis problems have been solved by transforming them into graph reachability problems [25]. Dataflow analysis is extended from short statement blocks to that between procedures (or functions) in [26]. *Path profiles* are commonly executing paths through programs. These are used in [27] to improve data flow information.

Generally, static dependency analysis ignores the issue of conditionally executed blocks, since there is often little that can be done to predict the values of conditions evaluated at runtime. In [28] and [29] the problem is considered of how to extract dataflow information in the presence of conditional statements. Pushdown automata and context-free grammars are the formalisms used. This work uses the fact that the branching of some conditionals are related to the branching of others. For example, if the pointer $x->l$ is not null, then the pointer x must be non-null as well and this may be used to predict earlier branches. This prediction allows the set of potential dataflow sources of a statement to be reduced.

2.2.2 Other Dependency Analysis Techniques

Memory expansion is a typical technique for extracting parallelism. In programs one variable may be written to more than once, this adds extra dependencies in the form of write-after-write and write-after-read that must be respected. Memory expansion aims to reduce these by expanding one variable into many. This removes some of these dependencies. Additional restoration code must be generated so that later reads can be informed which of the many possible writes actually produced the value that they want. There is a continuum of expansion strategies. In the extreme the program is converted into Single Assignment Form: no memory cell is written to twice. Maximal Static Expansion [30], aims to avoid restoration code. The expansion is maximal in the sense that any further expansion would require dynamic restoration code.

Dependency analysis can work at different levels of granularity within programs. Often dependencies between function calls (known as *interprocedural* dependencies) are ignored in favour of finding fine-grained dependencies within single blocks of code. A *Lattice model* approach to handing interprocedural dependencies is described in [31]. Here the static analysis proceeds by building up a lattice of information about data dependency, which is iterated over the function calls until a fixed point is reached.

2.2.3 Array Dependency Analysis

As mentioned earlier, if one confines programs to those that only access arrays then the dependency problem is simplified since the problem of aliasing is greatly reduced. In fact these approaches frequently ignore overlapping arrays, so no aliasing can occur: two different array names cannot refer to the same memory location. Arrays are simpler than pointers since it is relatively simple to work out at compilation time whether two indices are the same value, rather than whether two pointers can reference the same location. Often index expressions are restricted to linear functions to ease this. Pointer-style tricks such as using the values of one array to index into another are nearly always excluded from these analyses.

Since this is is a simpler problem, much more progress has been made in automatic array program analysis and it is instructive to review what is possible to achieve.

Array Dataflow Analysis is a technique, described in [32] to extract parallelism in data-parallel and control-parallel programs. In [33] dataflow analysis for scalars and arrays has been investigated. The technique of Array Privatization splits arrays up into independent sections which is useful for partitioning arrays between a number of independent processors. This has been done using an array dataflow analysis as in [34] .

In [35], dependencies between array references are handled by considering the monotonic evolution of induction variables, rather than finding closed form solutions. This can be ex-

tended towards handling pointer based structures, but aliasing is not treated. Properties of the references found include 'listness', 'treeness' and 'combness', so this touches on the shape analysis mentioned later in Section 2.4. Combness is the property that the map from array positions to elements is injective *i.e.* no two array positions map to the same element.

The Polytope model [36] is useful for handling dependencies in subsections of arrays which are defined by affine expressions in array subscripts. The automatic paralleliser LooPo [37], uses the polytope model to extract parallelism from `for` and `while` loops. The paper [38] uses the polytope model and summarises the use of a raft of techniques including Array Dataflow Analysis (ADA), Fuzzy ADA, single assignment, speculative execution, maximal static expansion to extract parallelism from array accessing code. The PIPS automatic paralleliser [39] handles FORTRAN code and arrays.

These analyses often require fairly intensive manipulations with complexity in the order of exponential in the size of the expressions. Presburger Arithmetic is such a technique, in recent work [40] a sub-domain of this is used to permit faster polynomial time analysis of array dependencies.

There is some crossover of techniques with pointer-based dependency analysis: Integer Linear Programming and formal language theory are both used as a framework for analysing recursive functions over arrays and trees in [41]. Again, only strict tree pointer structures are supported.

2.2.4 Pointer Dependency Analysis

We now turn to the main problem of interest in this thesis: dependency analysis in the presence of pointers. A recent and fairly exhaustive look at the current state of the art of pointer analysis is given in [42], its conclusion being that this is a complex problem with no complete solution. As mentioned earlier, in contrast to array analysis, analysing programs with pointers has the additional problem of aliasing. Alias graphs are used to find access conflicts in data structures in [43].

That this is a hard problem is given greater credibility by the fact that deliberately introducing pointer usage into code has been suggested in [44] as a method of code *obfuscation*. This obfuscation, which aims to hide the mechanism of the algorithms, can be used to permit programs to execute on untrusted machines without revealing the purpose of the program. By artificially introducing pointers into the algorithms, standard dataflow techniques of analysing code can be thwarted.

The *may-alias* problem aims to find out when two distinct pointers in a program *may* point to the same memory location. In [45] an equivalence is found between the pointer aliasing problem and the program dependency problem. Code is annotated to store when a variable is last written, thus converting dataflow into an extraction of may-alias information.

The concept of may-alias is given a more quantitative rigour in [46], where the *probability* of aliasing is introduced. This probability is computed statically for pointers within the program. This information can be used to refine transformations. If pointers can be aliases, then knowing how likely this is can permit code transformations that optimise for whichever is the most common case.

Dynamic pointer structures are considered in [47], where structures called *link graphs* are use to store the alias information. In [48] the problem of estimating where pointers may be pointing is tackled for multithreaded programs. In [49] points-to analysis and connection analysis are used to parallelise a number of pointer based benchmarks.

So far, these techniques make no assumption or prediction about there being any pattern to pointer linkages, and work for any general graph structure. This makes them artificially weaker when applied to structures where there is a known strict pattern to the linkages. The following try to make use of such additional linkage structure.

The work [50] considers shapes as free, commutative and almost free, using a toy recursive language (LEGS) and performs dataflow analysis with Numbered Occurrence Languages as the underlying formal model. Tree grammars are used to produce liveness patterns, and sub trees of used (or 'dead' data) can be pruned.

Escape analysis [51], decides when dynamically allocated objects in one part of a program can 'escape' to be accessed from other parts of the code. In [52], coarse-grained parallelism such as *function call* parallelism, *forall* parallelism and *foreach* parallelism is extracted from C code. Again, this uses information on whether it is a tree, DAG or cyclic data structure.

Traversal-Pattern-Sensitive pointer analysis [53] makes the assumption that each unique path through a data structure leads to a distinct node. This property does not hold for the kinds of structures we want to be able to deal with, since a number of distinct paths of pointers could lead to the same node. The reason that this is a useful valid approach is that when this finds aliases, these must be aliased in practice. The imprecision comes from the fact that it may mark pointers as independent (*i.e.* , not aliased) when they can point to the same memory location.

In the technique known as *k-limiting* [54], aliases are manipulated up to a depth of k (a chosen constant) pointer dereferences. Beyond that depth all pointers are assumed to be aliases. Thus, if applied to a binary tree, pointers to values below level k would all be considered potential mutual aliases, greatly reducing the resolution of dependency information that can be extracted. Indeed, for recursively defined data structures this approach is unsuitable. In [55], a symbolic access path method is used to tackle the problem of aliasing. These symbolic access paths are a finite representation of a potentially infinite set of aliases, and give a much greater resolution to the information. In [56] right regular relations are used to represent the aliasing information. A lattice of information is built up and manipulated.

The work [57] looks at loop transformations of pointer programs. The structures must have

a couple of properties for this analysis to work:

- *Structural inductivity*: they are non-cyclic and each node has at most one parent.
- *Speculative traversability*: All leaf pointer point back to themselves, so traversals never generate null pointers.

It uses a system of path matrices to hold and manipulate the information. The first property is a genuine restriction, cyclic structures are common and many other structures have more than one pointer pointing to a particular node. However, speculative traversability can always be applied to an arbitrary pointer linked structure, so does not really restrict the class of structures.

In [58] finite state automata are used to analysis recursive tree programs, but the only structures considered are trees. Dependency analysis is used to detect dereferences of invalid pointers in [59], here cyclic structures are permitted.

Most analyses that use pointer pattern information place significant restrictions on the patterns of linkages that they permit in order to do this. There is room for a more permissive system that still permits alias questions to be answered.

2.2.4.1 Pointer Structure Correctness Analysis

Reasoning about pointer structures can be done without the goal of dependency analysis. In [60], questions about the correctness or intentions of a pointer program can be formalised. This can include such assertions as whether a pointer will always be non null or if a manipulation of a pointer structure leaks memory. This work explicitly handles lists, but it can be generalised to trees.

In [61] programmer supplied assertions about pointers can be statically verified by the compiler. In [62] these sort of decision procedures are defined for LISP list structures.

In [63] another formalism for reasoning about programs that manipulate pointer based data structures are developed. The model of the heap includes such update operations as pointer assignment, pointer arithmetic, allocation and deallocation. This formalises much of the informal reasoning used by programmers to justify their programs. Their system of reasoning works most cleanly with tree or list structures that are composed of disjoint parts, how well it can be applied to structures with shared sub trees that are more graph-like has only begun to be tested with DAG structures in [64]. This paper also begins to apply the formalism to reasoning about concurrent processes that access shared memory. This is an approach that could be developed to the point where it could be applied to assisting with automatic parallelisation.

2.3 Structure Description Languages

One approach to assist with creating dependency information for programs that use pointer data structures is in designing languages that allow the programmer to describe in detail the exact shape of structures. Often the language manipulating the structures is novel as well.

Newgen [65] allows data domains to be described in a high-level specification language. Code is generated to handle the datatype. Types can be built up by applying operations (products, union, list or array) to basic types. This research illustrates the kind of support it is useful to give to programmers in the form of generation of datatype manipulation code, but the permitted data structures are limited to lists of lists.

The formalism of Group Theory [66] is used to describe structures. Direct computation with groups as the main data structures is considered in [67]. The language $8\frac{1}{2}$ is a data parallel language that has structures that are based on groups [68], [69], [70], [71]. Shapes can be defined using abelian and non-abelian groups. In this context *abelian* groups give rise to array-like structures, and non-abelian groups, specifically *free* groups, map to tree like structures. This approach could be relevant to any structure that maps to a group, but more general structures beyond arrays and trees are not considered. This work does not make much use of the potential that the group approach could offer in terms of manipulation of sophisticated structures.

In [72] it is shown that generalised grammars are not sufficient to describe common data structures, such as binary trees with linked nodes at each level. A second-order logic with constraints is presented as a better model.

ADDS [73], [74], [75] is a system of describing pointer-linked structures. It can handle lists, trees, octrees [1] and sparse matrices, although the linkage information can be crude. ASAP [76], [77], extends the ADDS approach with sets of axioms that convey more precise linkage information. In [9], simple theorems about pointer dependency can be proved using the ASAP axioms. By keeping the structure descriptions simple many linkage types are not supported.

In [78] a logic language for describing regular languages and trees is described. This finds applications in manipulation and expression of FSAs on large alphabets. It is related to the formalism Graph Types [79], in which data structures have a tree like backbone and other pointers that cut across. These routing fields are described using a regular expression language, with additional conditional symbols that determine whether a location is a root or leaf of the tree. Structures such as cyclic lists, trees with linked leaves and red-black trees are handled. This is a sophisticated way of expressing the most useful pointer structures, but there is no application of these techniques to extracting dependency information.

Shape types [80] uses grammars to describe structures. Structure linkages are represented

[1] Octrees are trees with eight child nodes at each branch. They are used to represent the eight neighbours that each square has in a two dimensional grid

by a multi-set of field and node names. Each field name is followed by a the pair of nodes that it links. The structure of the multi-set is given by a context-free grammar. In addition, operations on these structures are presented as *transformers*, rewrite rules that update the multi-sets. This can handle even the most complex of pointer structures, but has not been shown to enable useful dependency analysis.

In [81] regular path descriptions are used to define graph languages, examples of applying this to trees with additional linkages and skip lists are given. A decidable logic for describing linked data structures is given in [82]. Reachability expressions describe the linkages within the structure. They are used to annotate sections of code. Structures which change dynamically can also be handled with this approach. Again, dependency analysis is not a goal of this approach.

In general these approaches tend either to fail to support sufficiently general pointer linkages or use formalisms that are unlikely to be amenable to handling dependency analysis.

2.4 Automatic Shape Analysis

Rather than require the programmer to explicitly describe the linkages in the structure, an alternative approach is to attempt to automatically derive some of this information from the code that initialises the structures and sets up the pointer linkages. What we term *automatic shape analysis* aims to extract shape information directly from the code (often C) that defines and uses the structure. What summarises this linkage information is often referred to as *shape*.

Many techniques and papers use the name 'shape' to describe some aspect of data structures. This has been applied to multi dimensional array structures in the 'FISh' language [83]. Here the shape information of a multi-dimensional array is the number of dimensions and size of each dimension. This enables static shape analysis to be performed on polymorphic programs to convert them into efficient imperative ones. Here the notion of polymorphism includes shape as part of the type. Algorithms can be customised based on the shape of data structures to which they are applied. This gives the kind of performance benefit that compiled imperative languages have to a polymorphic functional language. However, trees and other pointer based shapes are not supported in this notion of shape.

Loosely speaking, we will use the term shape to represent the pattern of the linkages in a data structure. Shapely types [84] abstract the notion of the shape of a data structure (tree, list, graph) from the usual type information (int, float). Thus trees, lists and meshes are examples of shapes.

The following are typical types of shape that are identified by these analyses:

- Lists: single or doubly linked (pointers join each node with the next or previous node).
- Tree: A number of independent sub trees at each node. The sub-nodes (of a *parent* node) are termed *child* nodes. Usually the number of child nodes is assumed to be fixed, a

binary tree has two, but a variable number can be implemented by utilising a linked list of child nodes from each parent.

- Directed Acyclic Graph (DAG): Tree-like, has no cycles, but sub trees may be linked, a child may have more than one parent.

- Cyclic: Graph or list with cycles (pointers can link around into a loop).

In [85], the shape *attribute* (here one of 'Tree', 'DAG' or 'Cyclic') can be extracted from programs by computing matrices of pointer aliases. Doubly-linked lists and trees with leaves pointing to additional lists can be spotted in code in [86], and [87]. In [88], these techniques are extended to include arrays of pointer structures, typically used in sparse array implementations.

In [89], the shape analysis shape graphs of [90] are improved to handle cyclic data structures.

Kleene's system of three-valued logic (true, false, unknown) is used in [91]. They locate *shape invariants* in sections of code that perform destructive updating on heap data, and use three-valued predicate logic to phrase questions about the possible structures that can arise.

In [92], a method of storage shape graphs is used to extract shape information from programs. Finite Static Shape graphs approximate the linkages within potentially unbounded data structures. They are used in [90] and [93], to approximate possible shapes that a pointers structure can take on. Shapes detected are lists, circular lists (lists that link their tail node back to their head node) and trees. Questions such as whether sections of code preserve these properties of structures can be posed in this formalism and proofs constructed to answer them.

In [52], techniques for detecting situations where parallelism can be inferred are discussed. This includes functions that access discrete sub-trees of tree-like structures, loops that traverse list like structures and array based loops that access pointers to separate data structures. Both the structure types and the code that uses them are to be automatically detected.

These techniques are quite restricted in the amount of information about the structure they can extract. The tree, DAG or cyclic properties that they can detect are all useful indicators that can give some information about the dependencies of programs that access them, and provide hints towards possible parallelisation strategies. Ultimately however, this is quite coarse-grained information, for example, once cyclic structures are detected it must be assumed that all pointers within them are aliased.

2.5 Conclusions

Current data dependency analysis techniques tend to restrict heavily the data structures that they handle. On the one hand analysis of array structures benefits heavily from the simplicity that

restriction to these linear structures provides, on the other general pointer aliasing techniques can assume little about where pointers are supposed to be targetted.

Automatic shape analysis approaches manage to extract some shape information from programs, but this information is often fairly simplistic. This suggests that automatically extracting this kind of knowledge from programs is far from solved. This thesis does not attempt to tackle this problem directly, but looks instead at what form such linkage information should take to permit useful manipulations with the goal of extracting dependency information.

Existing pointer dependency analysis techniques tend to handle rather basic pointer linkages patterns. In contrast formalisms for reasoning about data structures tend to be too general to permit useful alias analysis. There is a need for a method of specifying pointer linkages that is sophisticated enough to handle most of the ways that pointers are used to implement recursive data structures, yet can permit useful dependency information to be automatically extracted from programs that manipulate them.

In the next chapter we will introduce the 2FSA system of specifying linkage information.

Chapter 3

Structure Descriptions using 2FSA

In this chapter we detail a method we call the *2FSA approach* for describing the linkages in complex pointer based structures.

We specify the description language, and give some specifications of example data structures. We then consider one application unrelated to the parallelisation problem, that of using the description to automatically generate the code to join up the link pointers in a structure.

Finally, we compare this approach with other existing methods of pointer structure specification available in the literature.

3.1 Recursive Structures in C

We will first begin by looking at how dynamic data structures are handled in a language such as C, and then relate this the 2FSA approach.

Consider the binary tree data structure in Figure 3.2. The items `value`, `l`, `r` and `n` are the *fields* of the structure, and may contain items of data (such as the integer `value`) or pointers to other parts of the structure (such as `l`, `r` and `n`). Although this is not specified in the code, we assume that `l` and `r` point to two disjoint subtrees, and the `n` ('next') pointer links the nodes

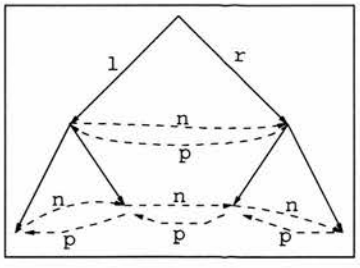


Figure 3.1: A binary tree with additional 'next' and 'previous' pointers.

```
struct Tree {
int value;
Tree * l;
Tree * r;
Tree * n;
};
```

Figure 3.2: The definition of a simple tree structure in C

of each level of the tree in a linked list. The p link the lists of nodes together in the reverse direction to the n. An example or such a binary tree structure is shown as a tree in Figure 3.1.

We will often refer to the pointers as *directions* which distinguishes them from the value fields and relates to the idea that there are a number of different routes that can be travelled from each node within the structure. We use the term *word* to denote any string of directions: we append words to *pathnames*, which are the unique names given to each node in the structure. If the word contains link directions, once the word is resolved this gives a new pathname (or pathnames).

We will now explain how to represent such a structure using the formalism of 2FSAs. We have a fixed list of symbols, the *alphabet* A, that correspond to the fields in the structure. For the binary tree, $A = \{l,r,n\}$, we define a subset of these fields $G \subseteq A$, as being the *generator symbols* (or just *generators*) of the structure. These pointers form a tree skeleton for this structure. Each node in the structure can be identified by a unique string of the generator symbols, called the *pathname* (a member of the set G^*[1]), which is the path from the root of the tree to that particular node. For the tree, the set of generators contains the left and the right pointer fields, $G = \{l,r\}$.

This pathname strategy is inspired by the work in [94], where pathnames are allocated to every node a the structure, and ownership of nodes by processes is decided directly from the names.

The remaining non-generators (we call them *link* directions) L represent pointers that can link nodes in the structure. These pointers cut across between nodes. This binary tree has such one link direction: $L = \{n\}$.

In this approach we require a specification of which nodes may be linked by a particular link direction. The full description of the structure thus contains a set of relations, $\rho_i \subseteq G^* \times G^*$, one for each link direction i. This relation links pathnames of nodes that are joined by a particular pointer. A node may be related in this way to more than one target node via a particular link. In practice the link will only join two nodes in a particular example of this structure; this 'multiple

[1]This *Kleene star* notation refers to the set of all strings of symbols from the set G

value' property allows for the representation of approximate information. It is useful to think of each relation as a function from pathnames to the power set [2] of pathnames: $F_i : G^* \rightarrow \mathcal{P}(G^*)$; each pathname mapping to a set of pathnames that it may link to.

The are many possible ways of representing such a relation between pathnames. We draw our inspiration from *Automatic Groups* research. Automatic groups [95] are a type of group that is particularly suited to direct algorithmic manipulation. Their group members are defined using *Two-variable Finite State Automata* (2FSA) descriptions. We will use these 2FSAs representation to describe our link directions. Although we will not pursue it any further here, some other results from automatic groups could have relevance to the area of pointer structures.

2FSAs are an example of a more general formalism known as *synchronised rational relations*, a comparative study of this variety of relations is given in [96].

We use the Knuth-Bendix Package for Automatic Groups [97], as a starting point for the 2FSA implementation we require later.

In group theory, members of a group are combined to form other group members. Thus, if we represent each of a subset of group elements (or generators, analogous to our generating directions) as a separate symbol, strings of symbols (or *words*) can be combined to form other group members. A single group element that is not one of the generators may have a number of different representations. The *word problem* is about deciding whether two words represent the same group element. For a general (non-finite) group this is undecidable; there exists no algorithm that is guaranteed to terminate that solves this problem. The word problem is analogous to the aliasing problem we have in pointer structures. Knuth-Bendix is an approach for solving the word problem in certain cases by reducing all representation of each group member to a common form that permits direct comparison. More details on applying this approach to automatic groups is given in [98, 99], more general approaches that don't rely on automatic groups representation are covered in [100] and [101].

The only direct connection between this thesis and Knuth-Bendix is that we use the *kbmag* library to implement 2FSA manipulations. The 2FSA formalism is introduced in the next section.

3.2 Two Variable Finite State Automata (2FSAs)

We first provide some background on finite state machines, for more detail consult, for example, [102]. They are related to formal language theory in [103]. We will try to give enough background here to permit our use of them in this thesis to be understood.

First some basic background on state machines. A (deterministic) Finite State Automaton (FSA) reads a string of symbols, one at a time, and moves from one state to another. The string

[2] The *power set* of a particular set is the set of all subsets. This any element of the power set is a subset of the original set.

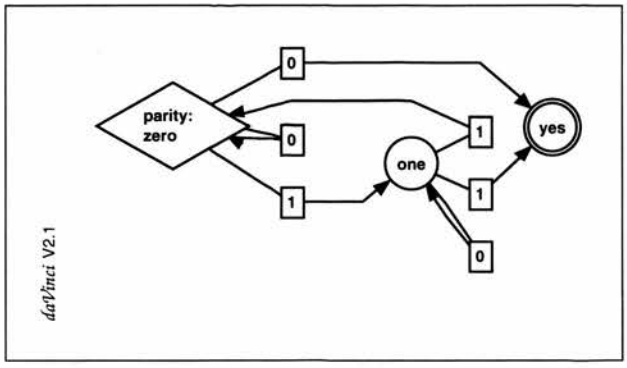


Figure 3.3: An example FSA in daVinci notation. This solves the parity checker problem.

is accepted, if it ends in one of the *accept* states when the string is exhausted. The automaton is specified by a finite set of states S, and a transition function $F : S \times A \rightarrow S$, which gives the next state for each of the possible input symbols in A. It can be best visualised by its state transition diagram, which shows each state and links them with labelled transitions.

An example FSA to illustrate the daVinci diagrams and notation used in this thesis is given in Figure 3.3. This FSA solves the *parity check* problem. In order to enable simple error-detection, strings of binary digits can be enhanced with an additional *parity bit*, which is zero if there are an even number of ones in the main string and one otherwise. This parity bit is appended to the original string. A parity check algorithm must therefore maintain a count of the ones and check to see if the final bit matches correctly. The FSA in Figure 3.3 analyses such a string symbol by symbol and accepts the string if the last bit checks correctly.

In this diagram the states are the diamond and the two circles. The diamond is the start or initial state (labelled *parity:zero* we call this the *zero* state here), the circle with a double border is an accept state, the single circle is a normal non-accepting state. These states are labelled with *zero*, *one* and *yes* to assist understanding of this example, most diagrams in this thesis number states in an arbitrary fashion. The string is consumed a symbol at a time, and transitions are made from state to state according to the values of the symbol (0 or 1). The squares with 1 or 0 in are labels on transitions between states.

If the whole string has been consumed and the current state is accepting (state *yes* in this example) then the parity has checked correctly. In this example the state name (*zero* or *one*) tells us what the current parity bit of the string should be. If the state is *zero* and the string contains a 0 next the transition is back to the *zero* state, if it is a 1 the transition takes us to the *one* state. Similarly a 1 takes us from state *one* to state *zero*, whilst a 0 keeps us in the same state. The additional transitions to *yes* are used to match the final parity bit to the computed parity of the string.

This is a non-deterministic FSA: when in the *zero* state, a 0 can both take us back to the *zero* state or on to the accepting *yes* state. Since the *yes* state has no outgoing transitions we

must have processed all of the symbols in order to terminate here and accept the string. If not then this path is ignored and we continue as if the only valid transition for 0 is back to *zero*. Thus this transition to *yes* is only considered if the last symbol is being read. There is a similar situation for 1 transitions from the *one* state.

We will occasionally make use of FSAs later when analysing code (Section 4.2.1) and describing partitions (Section 5.2.1), but most information we use will be held in the two variable version. We will sometimes refer to the one variable version as a 1FSA to differentiate it from the two variable type.

A *two-variable* FSA differs in that it attempts to accept a *pair* of strings and inspects one symbol from each of them, at each transition. Thus it can be used to define a relation between pathnames, two pathnames x and y are related if the pair are accepted by the 2FSA. We use this relation to encode the linkage of x to y via the pointer field associated with the 2FSA. We term x the *source* pathname and y the *target* pathname.

This can be thought of as a 1FSA, but with the set of symbols extended to $A \times A$. Thus each transition depends on a pair of symbols, one from each of the strings. There is a subtlety in that we may wish to accept strings of unequal lengths, in which case the shorter one is extended with the additional padding symbol: '$-$' . This results in the actual set of symbols being: $((A \cup \{-\}) \times (A \cup \{-\}))\backslash(-,-)$. It is worth noting that the double padding symbol is excluded as it is never needed. Some restrictions on the transitions are required to ensure that the padding symbol can only ever occur at the end of strings. Only transitions labelled with $(a,-)$ (for any non padding symbol a) can occur after a transition labelled $(a,-)$. The equivalent rule applies for transitions of the form $(-,a)$. As an example, the full set of possible transition symbols our binary tree example described earlier and based on the two pointer fields l and r is: $\{(l,l),(l,r),(l,-),(r,l),(r,r),(r,-),(-,l),(-,r)\}$.

We use the following notation for transitions in 2FSAs: if a 2FSA has a transition from state s to state t with symbol (a,b), we denote it as $s \xrightarrow[(a,b)]{} t$. We call a transition back to the same state (of the form $s \xrightarrow[(a,b)]{} s$) a *self transition*.

3.2.1 Mealy machine view

We will sometimes informally treat these automata as if they were *Mealy machines*. At each transition a Mealy machine takes an input symbol from the input string and outputs a symbol, which is appended to the output strings. Thus in our context it takes the first pathname as input and outputs a second pathname as the target of that pointer. So the transition $s \xrightarrow[(a,b)]{} t$ would consume the a symbol from the first pathname and output b as the next symbol in the target pathname. Note that viewing them in this way makes some 2FSAs that would be deterministic into non-deterministic 2FSAs. For example a 2FSA with transitions $s \xrightarrow[(a,b)]{} t$ and $s \xrightarrow[(a,c)]{} u$ from state s must non-deterministically output either b or c when a appears in the first pathname.

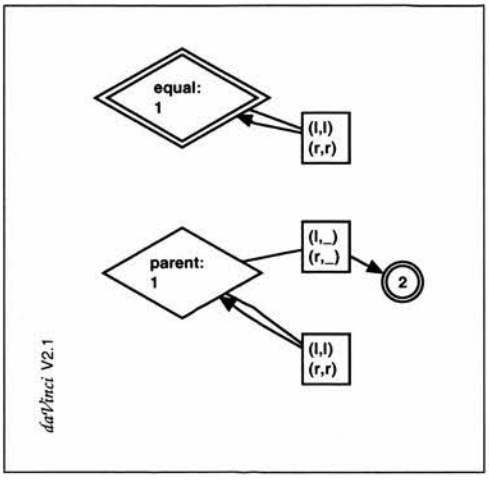


Figure 3.4: Two simple examples of 2FSAs, relating to the binary tree.

This occurs when one source pathname can be linked to a number of target pathnames. Treating them in this manner does not alter the relation encoded by the 2FSA, and may aid understanding certain operations performed on 2FSAs, for example when discussing the issue of composition of 2FSAs in Section 3.2.2.1.

A couple of illustrative examples of simple 2FSAs are given in Figure 3.4. These are both given in terms of the binary tree structure and thus use its alphabet of l and r directions, although they could be extended to any tree structure. The first 2FSA (`'equal'`) is the equality relation, only identical pairs of pathnames are accepted by it. It has one accept state (which is the same as the start state), with the only transitions being ones back to this state labelled with pairs of identical symbols (l,l) and (r,r). As this 2FSA moves through the two pathnames, any different pairs of symbols occurring in the two pathnames cause that pair of pathnames to be immediately rejected by the 2FSA. As a result the only pairs of pathnames accepted will be identical ones. Consider the pair of pathnames $l.r.r$ and $l.r.l$. The three pairs of symbols fed through the 2FSA will be (l,l), (r,r) and (r,l). The first two pairs of symbols will leave the 2FSA in the first state, the last (r,l) has no matching transition from the state and thus causes it to fail and not accept the pair of pathnames.

The second 2FSA in Figure 3.4, (`'parent'`) is the relation that maps any node to its immediate parent node. This has two states, with only the second one classified as accepting. As before, only identical transitions map back to the first state. Two pathnames in the process of being accepted will move repeatedly back to this state provided they share an identical prefix. In order to be accepted they must move to the second state. The only pairs of symbols that permit this are $(l,-)$ and $(r,-)$, in other words the last symbol of the first pathname is irrelevant (either an r or l), provided there is one, but the second pathname must have been fully

consumed at this point and thus only the padding symbol will match. Thus, for example, the pairs of pathnames $l.l.l.r$ and $l.l.l$ (think of it as $l.l.l.-$ to give the last symbol pair of $(r,-)$) will be accepted. However, $l.l.l.r$ and $l.l.l.r$ will be rejected, since both pathnames will have been consumed with the state machine left in state 1, which is not accepting. So in general the first pathname must be exactly one symbol longer (either an l or an r) than the second. This encodes the relation of one pathname to the pathname that represents its parent in the tree.

We will occasionally be required to employ *non-deterministic* versions of these automata during our analysis. As described previously deterministic 2FSAs allow only one transition from a particular state with a particular transition symbol; non-deterministic ones relax this condition, allowing many such transitions. The set of relations described by deterministic and non-deterministic 2FSAs is identical and we can always convert from any non-deterministic 2FSA to its equivalent deterministic one. Most of the manipulations we introduce in the next section take deterministic 2FSAs as input, but sometimes it will be simpler to define a non-deterministic 2FSA with a particular property we require and apply the determinising algorithm to it afterwards.

These 2FSAs are exactly the *left-synchronous rational transducers* used in [104].[3] The more general *rational transducers* allow padding symbol transitions (or more correctly empty or epsilon: ε transitions). A rational transducer that can be converted to a left-synchronous transducer is said to be *left-synchronizable*. Later we will allow epsilon transitions to be introduced into some of the transducers we build. The assumption is that they can be subsequently left-synchronised in the manner described in Section 4.3.1.

3.2.2 2FSA Manipulations

We will also use 2FSAs to hold and manipulate the dependency information that we gather. 2FSAs can be manipulated in a number of different ways. We will gather here and summarise the most important manipulations that we use later in the thesis.

- *Logical Operations.* We can perform basic logical operations such as AND (denoted $\wedge$ and sometimes termed *conjunction*), OR ($\vee$), NOT (!) on these 2FSAs. We define subtraction in the same way it is used in set theory, to subtract one 2FSA (B) from another (A) is to remove all of the pairs of paths that occur in both B and A from A. Thus $\mathsf{A} - \mathsf{B} = \mathsf{A} \wedge !\mathsf{B}$.

- The *Exists and Forall* automata. The 1FSA, $\exists(\mathsf{F})$, accepts x if there exists a y such that $(x,y) \in \mathsf{F}$. Related to this is the 2FSA, $\forall(\mathsf{R})$, built from the 1FSA R, that accepts (x,y) for all y, if R accepts x.

[3] This work uses a similar formalism, but does not consider data structures more complex than nested arrays and trees.

- *Composition.* Given a word of directions ($l.n$, say), we will want to calculate its 2FSA relation, in this case using the 2FSAs for l and n. This *composition* 2FSA can be computed: given two 2FSAs, R and S, their composition is denoted by R.S. This is defined as : $(x,y) \in$ R.S, if there exists a z, such that $(x,z) \in$ R and $(z,y) \in$ S. This is described in detail in the next section.

- *Inverse.* The inverse 2FSA, denoted F^{-1}, accepts (y,x) if, and only if, F accepts (x,y). Its construction is simple: it is built by swapping the order of each pair of symbols in each transition of F. Thus every transition $s \overset{\longmapsto}{(a,b)} t$ is replaced by $s \overset{\longmapsto}{(b,a)} t$.

The AND operation on two 2FSAs, A and B, is performed in the following manner. A new 2FSA X is constructed as follows:

- States in X are created from each pair of states $[a,b]$, with the first state from A and the second from B.

- Transitions in X are defined as follows. X has transition $[a,b] \overset{\longmapsto}{(x,y)} [c,d]$ if A has transition $a \overset{\longmapsto}{(x,y)} c$ and B has transition $b \overset{\longmapsto}{(x,y)} d$.

- The initial state is $[a_0,b_0]$ where a_0 is the initial state of A and b_0 is the initial state of B.

- If p is an accept state of A, AND q is an accept state of B, then $[p,q]$ is an accept state of X.

The construction of the OR 2FSA is almost identical, the only change being the last definition of accepting states:

- If p is an accept state of A, OR q is an accept state of B, then $[p,q]$ is an accept state of X.

3.2.2.1 Composition of 2FSAs

The composition operation is important so we will briefly describe its construction here. In our implementation we use a library of 2FSA manipulations 'fsalib' [97] which include in particular the composition and determinising algorithms. This operation is most easily understood if we adopt the Mealy machine view mentioned in Section 3.2.1, and think of the 2FSAs in terms in inputing and outputting symbols.

We have R and S and wish to form the combined 2FSA denoted R.S. The basic idea is simple: we run both 2FSAs together and feed the output symbols of R into the input of S. The states of R.S are pairs of states, the first from R, the second from S. The input symbols are taken from the input symbols of R, the output of R.S is the output of S. To map the transition from state (s_a,s_b) under symbol (x,y), we:

- Find all transitions from s_a in R with input symbol x to state u. Each one gives us a possible (non-deterministic) transition in R.S.

- For each one with output symbol t, say, find out whether S has a transition from s_b to state v with input symbol t. If it does and has output y, then add the transition $(s_a, s_b) \xrightarrow[(x,y)]{} (u, v)$

This leads to a non-deterministic 2FSA. Once determinised, this yields the correct 2FSA R.S. As is often the case with 2FSA manipulations, the padding symbols complicate matters, so we will not try to handle all the possible sub-cases here, but instead refer the reader to [95] for the full details.

As a worked example consider forming the composite direction $n.p$ from the n and p directions in Figure 3.8. Informally we expect these to cancel out and leave a 2FSA much like the equals one found in Figure 3.4.

Each state we consider will be a compound state $[a,b]$ with a from the n 2FSA and b from the p 2FSA. In this example the possible states will be $[1,1]$, $[1,2]$, $[2,1]$ and $[2,2]$. For a state to be accepting it must consist of two accepting states, the only accept state in this example is therefore $[2,2]$. The state state is $[1,1]$.

So we begin in state $[1,1]$. From state $[1,1]$, we can either take an l symbol as input or an r, since these are the two options for n in state 1. We take the l direction first.

The n 2FSA can either:

- Output an l and move to state 1. 2FSA p then takes the l as input, outputs an l and stays in state 1.

- Output r and move to state 2. Then p can take the r as input and either:

  - Output r and stay in state 1
  - Output l and move to state 2

If we condense these down, this results in the following transitions from state $[1,1]$:

- To state $[1,1]$ with pair (l,l).

- To state $[2,1]$ with pair (l,r).

- To state $[2,2]$ with pair (l,l).

Note that there is non-determinism here with two possible transitions with symbol (l,l).

Now return to state $[1,1]$, but with an input symbol of r. Here the n 2FSA can only do one thing, output an r and remain in state 1. However the p 2FSA can either take the r input and:

- Output r staying in state 1

- Output l move to state 2.

This gives us the additional transitions from state $[1,1]$:

- To state $[1,1]$ with pair (r,r).
- To state $[1,2]$ with pair (r,l).

We now need to repeat this with the remaining states. From state $[1,2]$, we can accept as inputs either l or r. Taking r first, there is no transition in p from state 2 with this as the first symbol. Thus we can disregard this as a possible transition. Turning to the l input, the only possibility is for n to output l, and p to output r and move to state 2. Thus we have the sole transition from $[1,2]$ to be the one to state $[1,2]$ with pair (l,r). We can simplify drastically here, since $[1,2]$ is not accepting and has only self transitions, no successful journey through the state machine can visit it, and we can remove it as a fail state. Similarly, we can remove any transitions to it from other states.

The same thing occurs with state $[2,1]$, it has only one self transition and can therefore be removed as a fail state.

For state $[2,2]$, n can take as input r, outputs l, which is converted back to a r by p. Thus the self transition with symbol (r,r) is given. Since $[2,2]$ is accepting, we do not cull it, as with the others.

Collecting this together, and removing the fail states, we get the full state transition information:

- From state $[1,1]$:
  - To state $[1,1]$ with pair (l,l).
  - To state $[2,2]$ with pair (l,l).
  - To state $[1,1]$ with pair (r,r).
- From state $[2,2]$ to state $[2,2]$with pair (r,r).

These are shown in diagram form in Figure 3.5. This is a non-deterministic 2FSA, and a little hard to interpret directly. We can make it deterministic in the following way, this acts as a good example of how this process works in general.

First, we relabel state $[1,1]$ as 1 and $[2,2]$ as 2. To make it deterministic we form a fresh 2FSA, each state of which is a set of states from the original. This corresponds to the concept of the 2FSA being in a number of states at the same time. Thus this new 2FSA has two states written $\{1\}$ and $\{1,2\}$. The $\{1,2\}$ state is accepting. Now to form the transitions.

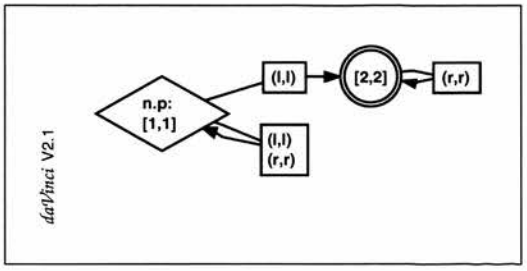


Figure 3.5: The non-deterministic 2FSA for the compound direction $n.p$

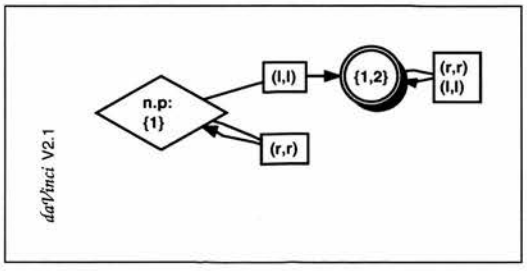


Figure 3.6: The final deterministic 2FSA for the compound direction $n.p$

From state 1 (l,l) can either take us to 1 or 2. Thus, we add the transition from $\{1\}$ to $\{1,2\}$ under symbols (l,l). Again from 1 (r,r) takes us back to 1, so we have the transition $\{1\}$ to $\{1\}$ under (r,r).

From state 2 (r,r) takes us back to 2, so from compound state $\{1,2\}$, (r,r) takes us to $\{1,2\}$. We also have a self transition from $\{1,2\}$, under symbol (l,l).

If we collect these together, we have the set:

- $\{1\} \overset{\longmapsto}{_{(r,r)}} \{1\}$
- $\{1\} \overset{\longmapsto}{_{(l,l)}} \{1,2\}$
- $\{1,2\} \overset{\longmapsto}{_{(r,r)}} \{1,2\}$
- $\{1,2\} \overset{\longmapsto}{_{(l,l)}} \{1,2\}$

These are shown in Figure 3.6. This 2FSA is simple enough to interpret directly. It only ever transforms l into l and r into r, so it is close in effect to the equality 2FSA we were expecting. However its domain is restricted, only paths of the form $r^*.l.(l|r)^*$ (using regular expression notation) are accepted. Put more simply, the pathnames consisting of only r symbols are excluded, these are the nodes down the right-hand edge of the binary tree. This makes sense, since these pathname is also excluded from the domain of the n direction, as there is no node to the right of these nodes.

We next include some general definitions for referring to special one and two variable FSAs.

Definition 1 *The relation* $[=]$ *is the equality relation,* $(x,y) \in [=] \iff x = y$. *We may refer to the equality 2FSA:* $=$.

Definition 2 *The* empty *2FSA is one that accepts no pairs of pathnames. Viewed as a graph of states and transitions, there are many 2FSAs that are equivalent to the empty 2FSA; for example any 2FSA with no accept state. However, we can apply simplification algorithms that map any such 2FSA to the canonical empty 2FSA, the one with no states.*

*When we wish to determine if two 2FSAs (*A *and* B*) are equal it is simpler and more reliable to form the combination '*A - B*' and check it for emptiness rather than to try to simplify and compare both.*

Definition 3 *Given a 2FSA* F, *we define the following 1FSAs:*

- *First(*F*). Accepts x if* F *accepts* (x,y) *for any y.*
- *Second(*F*). Accepts y if* F *accepts* (x,y) *for any x.*

Definition 4 *Given a 1FSA* F, *we define the following 2FSAs*

- *Double*(F). *If* F *accepts the word x then Double*(F) *accepts* (x,x).
- *FirstAll*(F). *If* F *accepts x then it accepts* (x,y), *for all y.*
- *SecondAll*(F). *If* F *accepts x then it accepts* (y,x), *for all y.*

The composition of a 1FSA F and a 2FSA T is a 1FSA denoting the mapping by T of all the possible strings accepted by F. In other words (x,y) is accepted by F.T if and only if x is accepted by F and (x,y) is accepted by T. We actually compute this by extending F using the *Double* operation and then applying the 2FSA composition directly. This gives:

$$\mathsf{F.T} = \mathrm{Double}(\mathsf{F}).\mathsf{T}$$

We will often need to be able to define an order on the words that 2FSAs accept. We use this when we deal with certain multi-valued 2FSAs in Section 3.6.1, and again in Chapter 4, whenever we need the correct sequential execution order of control words. The ordering we use is the *lexicographic* order obtained by extending the order on the alphabet of symbols in the following way:

Definition 5 *The* lexicographic *order* $<_L$ *on words is defined as follows:*

- *The alphabet A has an ordering* $<_A$.

- *Extend this to words with* $w <_L w'$, *iff*
  - $\exists\, x$ *and* x' *in* A $(x <_A x')$, u, v, v' *in* A^*, *such that* $w = u.x.v$ *and* $w' = u.x'.v'$
  - *Or* $\exists\, v$ *in* A^* *such that* $w' = w.v$ [4]

A 2FSA which we denote '$\geq$' can be constructed to store the relation defined by $(x,y) \in \geq$ if and only if $x <_L y$.

3.2.3 Regular Expression Syntax

We may refer to 1FSAs (and in some cases 2FSAs) by using their standard *regular expression* syntax. This will give expressions such as $A.B^*$ and $A^+.B$. These are fully described in [8], but we give a brief description of them to permit explanation of the syntax we use.

If r and s are regular expressions, denoting languages R and S then so are:

- $r|s$: The language containing elements from either R or S. (It can also be given as $R \cup S$ in standard set notation).
- $r.s$ The language made up of any element from S appended to any from R.
- $(r)^*$. The language consisting of zero or more repetitions of elements of R. This is known as a *Kleene star* operation. We will also use the notation r^+ to denote one or more repetitions, it is equivalent to $r.r^*$.

Whilst 1FSA are often most easily understood as regular expressions, 2FSA are usually simplest to follow diagrammatically. Throughout this thesis we will use the state diagram both for definition and illustration.

3.3 Structure Description Language

The linkages in the structure are described in addition to the program by providing a separate 2FSA specification for each link direction. We use a simple representation: a textual version of the 2FSA. This could be provided by the programmer in addition to the code to be analysed. This burden could be eased by use of predefined libraries of common structures. The full grammar of the structure description language is given in Figure 3.9.

The language description for the example binary tree structure is given in Figure 3.7, which we explain next. These 2FSAs are shown in pictorial form in Figure 3.8.

[4] This covers the case when w is a prefi x of w'. Thus, just as in an English dictionary, the word *stand* occurs before *standard*

```
class Binary_Tree {
Tree * l;
Tree * r;
Tree ~ n;
Tree ~ p;
};
{l: }
{r: }
{n:
States 2;    Start 1;    Accept 2;
Transitions[    [(l,l)->1;(r,r)->1;(l,r)->2;]      //From state 1
                [(r,l)->2;]                        //From state 2
]}
{p: (n)^}
```

Figure 3.7: The textual version of the description for links in the binary tree.

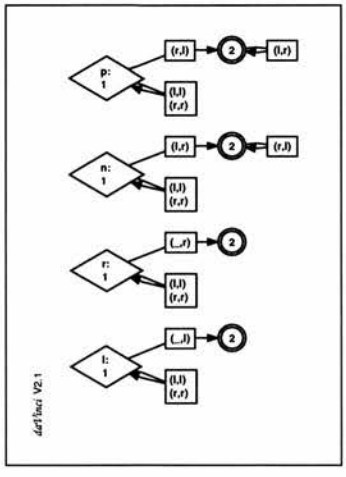


Figure 3.8: The 2FSAs for the links directions, shown in diagram form. The rhombus nodes are start nodes and are labelled with the direction name e.g. (l). Other states are circles and are labelled with state number e.g. $S:2$. The boxes hold the pair of symbols for that transition. Double borders indicate an accept state.

The basic type description of the structure is enclosed with a `class` block, that names the structure (`Binary_Tree`) and lists the member fields of it, just as in a class description in C++. The pointer types are differentiated in the program, with'`*`' denoting a generator pointer and '$\sim$' denoting a link pointer. The description of the link directions 2FSA then follows. The descriptions for the generator are not required since they can be inferred, but are mentioned in this description for completeness.

The description of each link direction is given explicitly in terms of a 2FSA with all of the transitions supplied. The following syntax is used to give a textual description of the automaton. It begins with the number of states (here 2), the start state number (1) and a list of accepting states (just state 2). The transitions for this 2FSA are then given with the transitions from each state given in order, enclosed by square brackets.

A link description can also be defined in terms of an expression referring to other link descriptions or generator directions. Here, the syntax `p: (n)` $\wedge$ is used to designate that the link direction `p` is the inverse of the link direction `n`. The expressions used can combine existing link description in a number of ways which relate to the basic operations given above. In addition we allow the *transitive closure* operation, which we describe in more detail in Section 4.4.

Note that the expressions may include some operations (in particular reverse and closure) that cannot be resolved correctly for all possible 2FSAs. In the case of the closure operation (which we fully describe later in Section 4.4), the heuristics used to compute it may fail. In the case of the reversal operation described in Section 3.4.3, the reversed 2FSA may not exist. If this is the case, the expression can generate an error to alert the programmer when it is parsed.

3.3.1 Building pointer descriptions

The examples given earlier in Section 3.2 were fairly simple, and it should be straightforward to see how the 2FSA descriptions were derived. We now turn to slightly more complex examples, and try to give some idea as to how they are constructed.

Referring back to the binary tree example, consider the task of designing the link description 2FSA for the 'next' (n) link direction.

Returning to the example structure in Figure 3.1, we can first look at some example pairs of pathnames that are linked by the n pointer and that we must link under the n 2FSA direction description. In the first level of nodes of the tree, just below the root, are two nodes with pathnames of a single symbol: l and r. Clearly the n direction must link l to r. Also r has no n node, so r must not link to anything.

In the next level there are four nodes, each with a two symbol pathname. The leftmost is $l.l$, which links to it sibling $l.r$ using the n pointer. Reading rightwards, this $l.r$ node must in turn link to $r.l$, which must be linked to $r.r$. As before the rightmost node, in this case $r.r$ has no n neighbour.

structuredesc	::= ***declarestruct definestruct***	
declarestruct	::= 'class IDENT {' ***decfieldlist*** '};'	
decfieldlist	::= ***decfield***	
	| ***decfieldlist decfield***	
decfield	::= 'IDENT *' GENIDENT ';'	
	| 'IDENT ' LINKIDENT ';'	
	| ***scalartype*** GENIDENT ';'	
scalartype	::= 'int | float | double | char'	
definestruct	::= ***linkdefinelist***	
linkdefine	::= '{' LINKIDENT ':' ***linkdesc*** '}'	
linkdesc	::= ***fsadefin***	
	| ***fsaexpr***	
fsadefin	::= 'States' INT ';'	
	'Start' INT ';'	
	'Accept' INTLIST ';'	
	'Transistions' '[' ***translist*** ']'	
translist	::= ***trans***	
	| ***translist trans***	
trans	::= '[' ***movelist*** ']'	
move	::= '(' GENIDENT ',' GENIDENT ') ->' INT ';'	
fsaexpr	::= '(' ***fsaexpr*** ')'	
	| ***fsaexpr*** '.' ***fsaexpr***	(composition)
	| ***fsaexpr*** '||' ***fsaexpr***	(or)
	| ***fsaexpr*** '&&' ***fsaexpr***	(and)
	| ***fsaexpr*** '^'	(inverse)
	| ***fsaexpr*** 'REVERSE'	(reversed)
	| ***fsaexpr*** '@'	(closure)
	| '!' ***fsaexpr***	(not)

Figure 3.9: The 2FSA description language. The symbols GENINDENT and LINKIDENT are used to differentiate between generator and link identifiers, and show where each may appear.

Generalising somewhat, if we consider a node that is the left child of its parent, its pathname ends in a l symbol. If we move eastwards from this node, we reach its right-hand sibling node, whose pathname is the same, apart from the last l which is flipped to a r symbol. This is expressed in the equation $n(x.l) = x.r$

Similarly, a right hand child node's pathname ends with a r symbol, call this pathname $x.r$, say. If we know the next node of its parent pathname x, we just need to append an l to get the next node of the right hand child. We can express this recursively in the equation: $n(x.r) = n(x).l$. By applying this and the equation from the previous paragraph we can convert any pathname into the pathname of the next node.

We can convert this pair of functions into an iterative procedure for finding the next pathname. We assume we are progressing through the pathname in reverse, from its end to its beginning. Thus with the pathname $l.r.l.r.r$, we begin from the r end. Processing each symbol in turn:

- If it is an r: flip it to an l. Now process the previous symbol.
- If it is an l: flip it to an r, and STOP.

This converts $l.r.l.r.r$ to $l.r.r.l.l$. This algorithm processes the pathname in reverse. In order to produce a 2FSA we need to process path names from the beginning as so must reverse this process. To do this we spot that the procedure mentioned above locates the last l in the pathname, flips it to an r and swaps the remaining r's to l's.

We can express this information by stating that the next direction links nodes of the format, $x.l.r^q$, to nodes given by $x.r.l^q$, where q is any integer (or zero), and x is any path name. Converting this in terms of the pairs of symbols that we need in 2FSAs, we want a 2FSA that accepts pathnames with an identical prefix (any number of either (l,l) or (r,r) pairs), followed by one (l,r) pair and then any number of (r,l).

This description can now be readily converted to a 2FSA. The n direction is given in Figure 3.7. The p direction can be found is derived in much the same way as the n direction and is included along with others for the l, r directions.

The generator 2FSAs (l and r) are simpler to understand, since they accept pairs of paths that are identical up the point where the first terminates and the second has an additional symbol (either an l or r). Although they will be needed during analysis, these generator descriptions do not need to be supplied as they are simple enough to be derived automatically from the alphabet of the structure.

3.4 Example structures

We now look at a selection of example pointer structures that have been mentioned in the literature and can be described using the 2FSA formalism.

3.4.1 Skip lists

It may appear that this approach only applies to tree structures, but list structures are covered as well, since we can think of a list as a trivial recursive tree structure with each node having only one sub-tree at every level.

The *skip list* described in [80] is a list (linked by n pointers) with an additional *skip* pointer (denoted s) at each node that points to a node further down the list. This allows traversals through the list to be sped up by optionally traversing via the skip pointer. For example, if the list contains a sorted dictionary of words, a search for a specific word can skip some intervening words if the target word pointed to by the *skip* pointer is alphabetically before the word that is being searched for.

A simple concrete example of a skip list one is shown in Figure 3.10. We can describe them using one n generator pointer that gives the list backbone and a s link pointer that can only point 'downwards' in the list. The 2FSA description is given in Figure 3.12. The pictorial representation of this is given in Figure 3.11. Consider the example of deciding whether the pathname $n.n$ is linked to $n.n.n$ by the skip link. These two pathnames give rise to the list of pairs of symbols (n,n), (n,n) and $(-,n)$, with the padding symbol added to the last pair since the first pathname has run out of symbols. Applying the 2FSA, the two (n,n) pairs leave the automaton in state 1, the $(-,n)$ takes it to the accept state 2. Thus these pathnames are linked by a skip pointer.

Similarly, the skip pointer links the pathname $n.n$ to pathname $n.n.n.n$, but does not link it to n (for that would be backwards: up the list) or $n.n$ (itself). In general, it specifies that only pathnames of the form n^p may be linked to n^q, where $q > p$ This skip list example uses the fact that the 2FSA description is not required to link a node to a unique target node. In fact this skip direction links each node to the entire set of nodes further down in the list. This flexibility of target means that corresponding dependency information will be less accurate.

Despite this imprecision it is still possible to make some predictions directly from these descriptions. In any normal, well-formed list, for any node x, the nodes $x.n$ and $x.n.n$ can be said to be *independent*, these nodes are not aliases, an update of the data in one does not affect the update of data in the other.

For skip lists, despite the additional flexibility, we can also have in-dependencies involving the skip pointer: $x.n$ is always independent of $x.s.n$, and $x.n.s$.

Whether these expressions are independent, or not, can be verified directly from the 2FSA

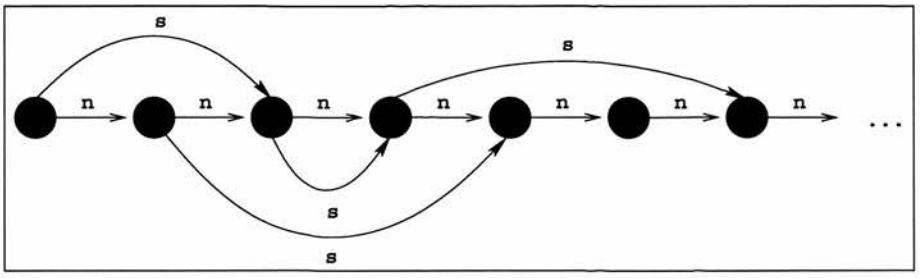


Figure 3.10: An example skip list. The skip pointer jumps forward to a node further on in the linear list.

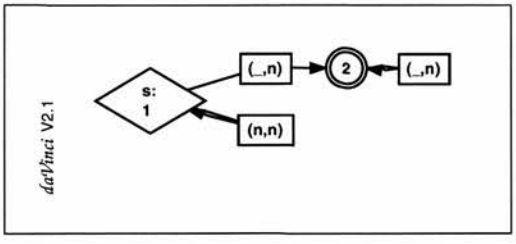


Figure 3.11: The 2FSA description for the skip direction in the skiplist.

descriptions. Suppose we wish to verify that $x.s.n$ is always a separate pointer from $x.n$. Firstly, form the 2FSA for the compound direction, $s.n$, and intersect with the n 2FSA using the AND operation. This resulting 2FSA will accept all the pathnames that are linked by the $s.n$ compound direction and are also linked by the n direction. If this 2FSA is empty (as it is in this example), then the expressions are independent.

This is information that may be used directly to parallelise a program. For example if a program contains two consecutive statements which can never access the same node in the structure, they can safely be run concurrently.

3.4.2 Binary Tree

Figure 3.1 contains our binary tree structure, each node of which has a left and right subtree, and in addition, the nodes in each level are connected as a doubly-linked list, with 'next' and, in the opposite direction, 'previous' pointers.

The description in Figure 3.7 declares each direction as either a generator or a link, followed by the definitions of each. The l and r directions are generators, and as such do not require an explicit 2FSA description.

The n direction is given with a full 2FSA description, the p direction is defined as the inverse of the n direction, denoted by the syntax '`(n)^`'.

```
class SkipList {

Tree * n;
Tree ~ s;
int val;
};
{s:
States 2;
Start 1; Accept 2;
Transitions[
[(n,n)->1;(_,n)->2;]
[(_,n)->2;]
]
}
```

Figure 3.12: The language description for the skiplist.

3.4.3 A Rectangular Mesh Structure

We now take an even more complex structure and explore how it is specified. Consider the *rectangular mesh* structure in Figure 3.13. Such a structure is used in finite-element analysis, to analyse fluid flow within an area. It can be thought of as a two-dimensional version of the binary tree we dealt with in the previous section. Considered as a tree structure, at each level of the tree, rather than the nodes linking up as a one-dimensional list they create a two-dimensional grid.

If, instead, we look at how it is constructed in terms of partitioning a two dimensional area, we get the following insight. The whole area is recursively divided up into rectangles, each rectangle possibly split up into four sub-rectangles. This allows for a greater resolution in some areas of the space. We can imagine each rectangle represented as a node in a tree structure; the variable resolution mesh results in an unbalanced tree, as shown in Figure 3.14. Each node may be split up further into four sub-nodes. Each rectangle in the mesh has four adjacent rectangles that meet along an edge in that level of the tree. For example, the rectangle 4 on the bottom right has rectangle 3 to the left of it and 2 above. This is also true for the smallest rectangle 4, except that it has a rectangle (3) to its right. We call these four directions: l, r, d and u. We therefore have a tree backbone of the structure, with the four generators $d1, d2, d3, d4$, and link pointers denoted by l, r, d, u.

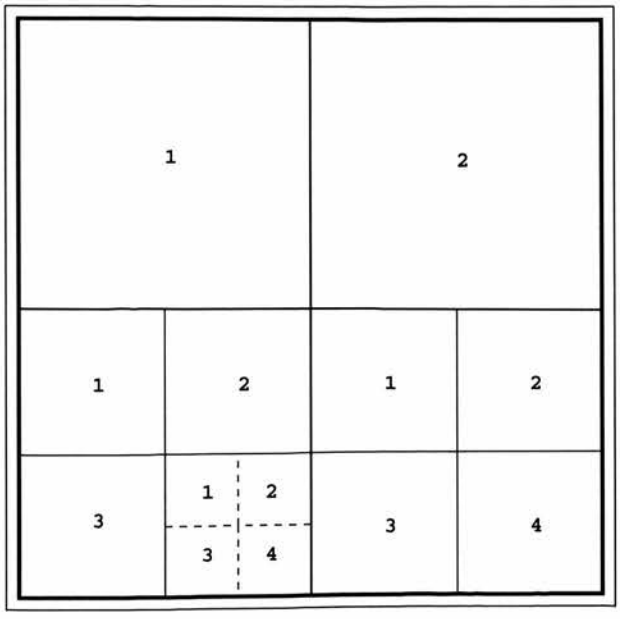


Figure 3.13: The variable resolution rectangular mesh.

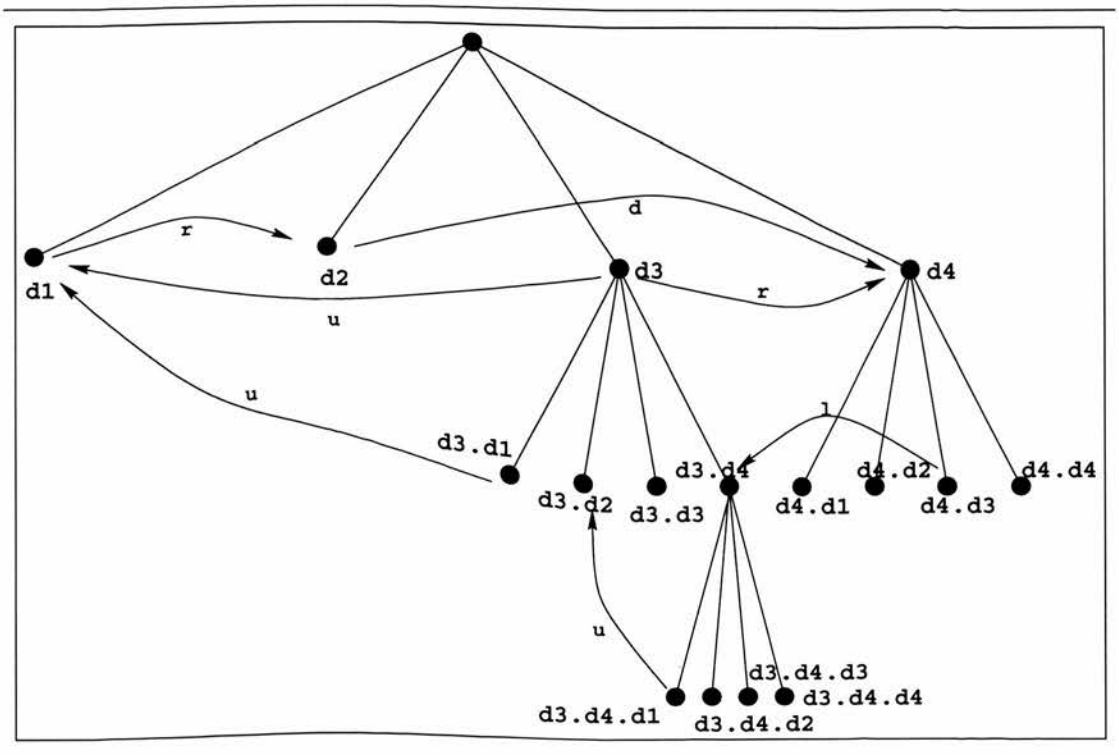


Figure 3.14: The representation of the rectangular mesh as a tree structure with the l, r, d, u links to adjacent rectangles. Not all the links are shown.

```
class Rectangular_Mesh {
  Tree * d1, *d2, *d3, *d4;
  float data;
  Tree ~ l, r, u, d;
};
{l:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
                (d2,d1)->2;(d4,d3)->2;]
              [(d1,d2)->2;(d3,d4)->2;(d1,_)->3;(d3,_)->3;]
       [(d1,_)->3;(d3,_)->3;]
  ]
    }
{r:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
                (d1,d2)->2;(d3,d4)->2;]
              [(d2,d1)->2;(d4,d3)->2;(d2,_)->3;(d4,_)->3;]
       [(d2,_)->3;(d4,_)->3;]
  ]
    }
{u:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
                (d3,d1)->2;(d4,d2)->2;]
              [(d1,d3)->2;(d2,d4)->2;(d1,_)->3;(d2,_)->3;]
       [(d1,_)->3;(d2,_)->3;]
  ]
    }
{d:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
                (d1,d3)->2;(d2,d4)->2;]
              [(d3,d1)->2;(d4,d2)->2;(d3,_)->3;(d4,_)->3;]
       [(d3,_)->3;(d4,_)->3;]
  ]
    }
```

Figure 3.15: The source description for the rectangular mesh.

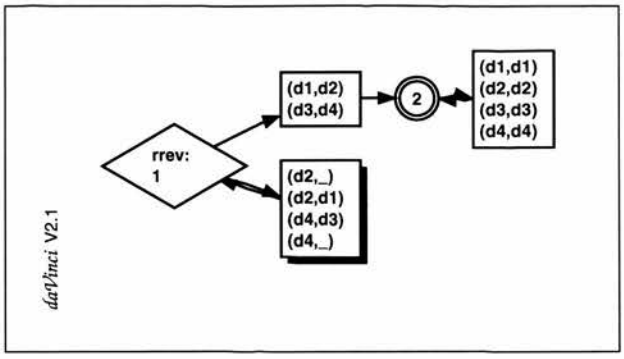


Figure 3.16:

We deal with the possible variation in resolution (or depth of tree) in the following way. We assume that the links pointers join nodes at the same level of the tree, where these are available, or to the parent of that node, if the mesh is of a lower resolution at that point. So moving in direction u from node $d3$ takes us to node $d1$, but going up from node $d3.d1$ takes us to node $d1$, since node $d1.d3$ does not exist.

We next distill this information into a set of equations that hold this linkage information. We will then convert these equations into 2FSA descriptions of the structure. Let us consider the r direction. Going right from a $d1$ node takes us to the sibling node $d2$. Similarly, we reach the $d4$ node from every $d3$. To go right from any $d2$ node, we first go up to the parent, then to the right of that parent, and down to the $d1$ child. If that child node does not exist then we link to the parent. For the $d4$ node, we go to the $d3$ child to the right of the parent. This information can be represented for the r direction by the following set of equations. The symbol x is is used to denote any pathname.

$$r(x.d1) = x.d2 \tag{3.1}$$

$$r(x.d2) = r(x).d1|r(x) \tag{3.2}$$

$$r(x.d3) = x.d4 \tag{3.3}$$

$$r(x.d4) = r(x).d3|r(x) \tag{3.4}$$

These equations are next converted into a 2FSA for the r direction. Note that they convert one pathname into another by working from the end of the pathname back to the beginning. The rule '$r(x.d4) = r(x).d3|r(x)$' does the following: if $d4$ is the last symbol in the string, replace it with a $d3$ or an ε, then apply the r direction to the remainder of the string. This is similarly true for the '$r(x.d2) = r(x).d1|r(x)$' rule. In the rule '$r(x.d1) = x.d2$' the last $d1$ symbol is replaced with a $d2$, and then outputs the same string of symbols as it receives as

$$
\begin{aligned}
d(x.d1) &= x.d3 \\
d(x.d2) &= x.d4 \\
d(x.d3) &= d(x).d1|d(x) \\
d(x.d4) &= d(x).d2|d(x)
\end{aligned}
$$

$$
\begin{aligned}
u(x.d1) &= u(x).d3|u(x) \\
u(x.d2) &= u(x).d4|u(x) \\
u(x.d3) &= x.d1 \\
u(x.d4) &= x.d2
\end{aligned}
$$

$$
\begin{aligned}
l(x.d1) &= l(x).d2|l(x) \\
l(x.d2) &= x.d1 \\
l(x.d3) &= l(x).d4|l(x) \\
l(x.d4) &= x.d3
\end{aligned}
$$

Figure 3.17: Equations for the link directions in the rectangular mesh structure.

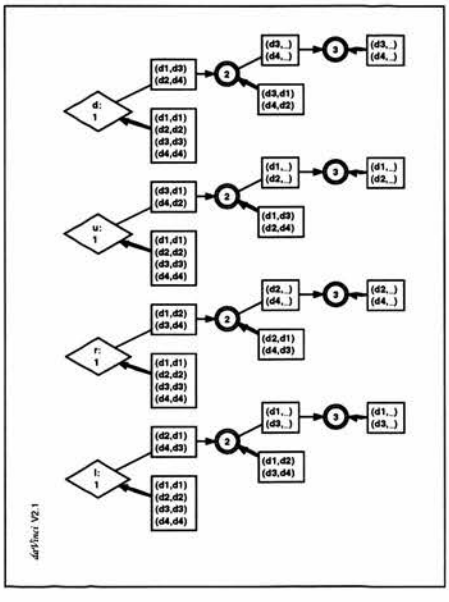


Figure 3.18: 2FSA descriptions for the link directions in the rectangular mesh structure.

input. Viewed in this way we can create a 2FSA that accepts the paths, but in reverse order, see Figure 3.16 for this 2FSA.

We get to the true 2FSA for this direction by applying the *reversal* operation. This is produced by reversing the transitions in the automata and exchanging initial and accept states. Note that the presence of transitions labelled with ε may make this reversal impossible in general. However, for this r link direction this process does work, as it also does for the other link directions in this structure: l, d, and u. See Figure 3.17 for the full set of equations and Figure 3.18 for the 2FSA descriptions. The source code for this description is given in Figure 3.15.

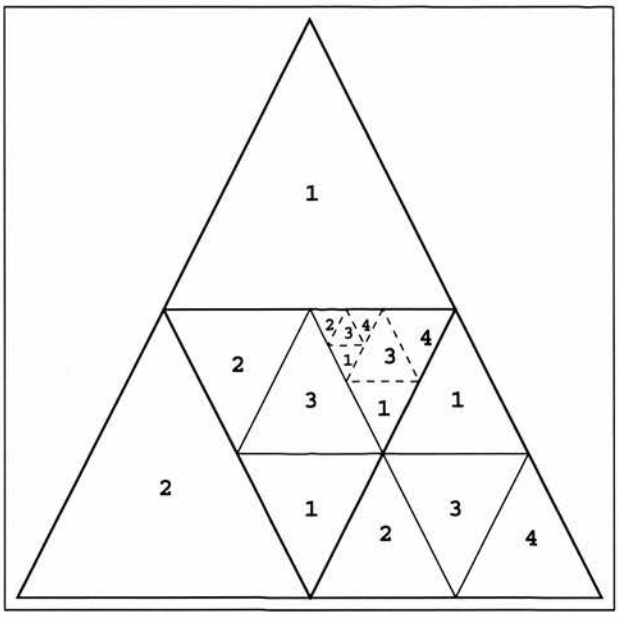


Figure 3.19: The variable resolution triangular mesh.

3.5 A Triangular Mesh Structure

Another recursive structure used in finite element analysis codes is the triangular mesh shown in Figure 3.19. This structure shares many of the properties of the rectangular mesh, but the linking pointer descriptions are significantly different.

Here the area is recursively divided up into triangles rather than rectangular regions. Again, each triangle may be split up into four sub-triangles. In the unbalanced tree structure each node has four sub-nodes, and, as before, the generators are labelled $d1, d2, d3, d4$. This tree form is shown in Figure 3.20. The link pointers l and r denote the left and right directions as for the rectangular mesh, but now each triangle has one vertical neighbour. Due to the way the triangles are split this neighbour may be positioned in either the down or up direction; we denote this direction d.

We can now begin to construct equations that represent the linking properties. Let us first consider the r direction. Going right from a node 2 takes us to the sibling node 3. Similarly we reach the node 4 from a node 3. To go right from a node 1, we go to the parent, then to the right of that parent, and down to the child 2. If that child node does not exist we link to the parent. For the node 4, we go to the child 1 of the right of the parent. We can store all this information for the r direction in the following set of equations.

$$r(x.d1) \quad = \quad r(x).d2|r(x) \tag{3.5}$$

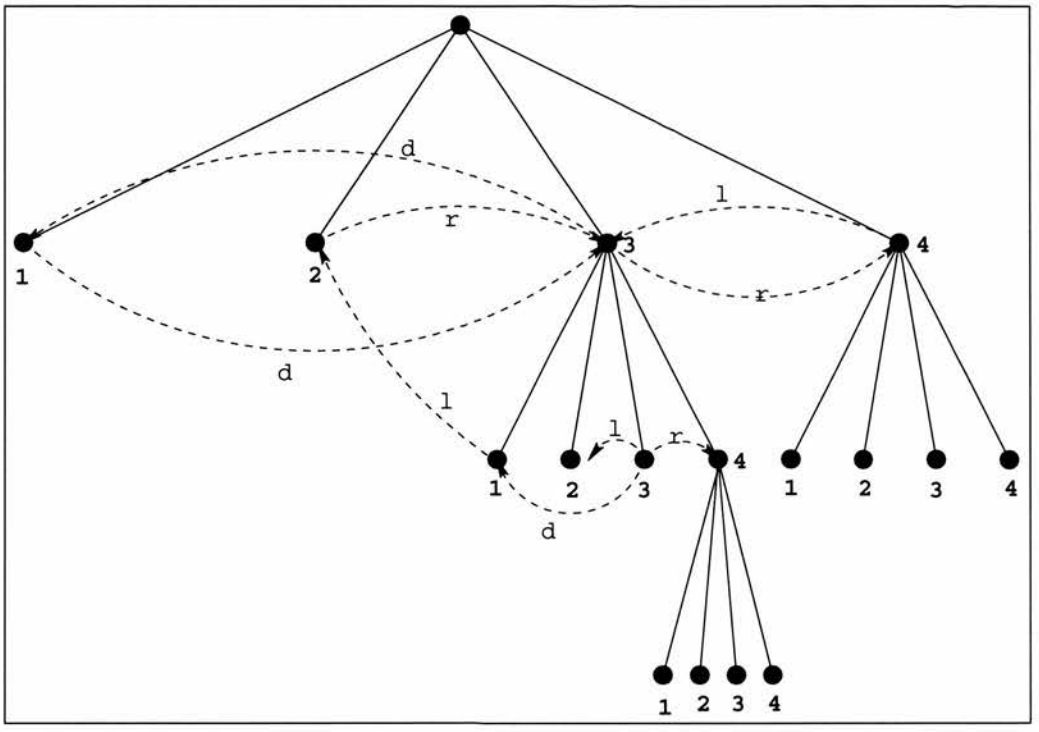


Figure 3.20: How the mesh is represented as a tree structure with the l, r, d links to adjacent triangles. Not all links are shown.

$$r(x.d4) = r(x).d1|r(x) \quad (3.6)$$

$$r(x.d2) = x.d3 \quad (3.7)$$

$$r(x.d3) = x.d4 \quad (3.8)$$

We next convert these equations into a 2FSA for the *right* direction. Note that they convert one pathname into another by working from the end of the pathname back to the beginning. What the rule $r(x.d4) = r(x).d1|r(x)$ does is : if $d4$ is the last symbol in the string, replace it with a $d1$ or a ε, then apply the r direction to the remainder of the string. Similarly for the $r(x.d1) = r(x).d2|r(x)$ rule. The rule $r(x.d2) = x.d3$ replaces the last $d2$ symbol with a $d3$, and then outputs the same string of symbols as are input. For the reverse 2FSA, see Figure 3.22. As before, we can reverse these descriptions to produce the full set, see Figure 3.23 for the equations and Figure 3.24 for the 2FSA descriptions. The appropriate source code is given in Figure 3.21.

These descriptions have been produced with the assumption that the tree may be unbalanced. We may have additional restriction that the tree will always be balanced, so that the link pointers always join nodes at the same depth of the tree. This simplifies the description, see Figure 3.25, with source code in Figure 3.26

```
class Triangular_Mesh {
  Tree * d1, *d2, *d3, *d4;
  float data;
  Tree ~ l, r, d;
};
{l:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
                (d3,d2)->2;(d4,d3)->2;]
        [(d1,d4)->2;(d2,d1)->2;(d2,_)->3;(d1,_)->3;]
        [(d2,_)->3;(d1,_)->3;]

  ]
    }
{r:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
              (d2,d3)->2;(d3,d4)->2;]
              [(d4,d1)->2;(d1,d2)->2; (d4,_)->3;(d1,_)->3;]
              [(d4,_)->3;(d1,_)->3;]
  ]
    }
{d:
  States 3;
  Start 1; Accept 2,3;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
              (d1,d3)->2;(d3,d1)->2;]
              [(d4,d4)->2;(d2,d2)->2; (d2,_)->3;(d4,_)->3;]
              [(d2,_)->3;(d4,_)->3;]
  ]
    }
```

Figure 3.21: The source code description for the triangular mesh.

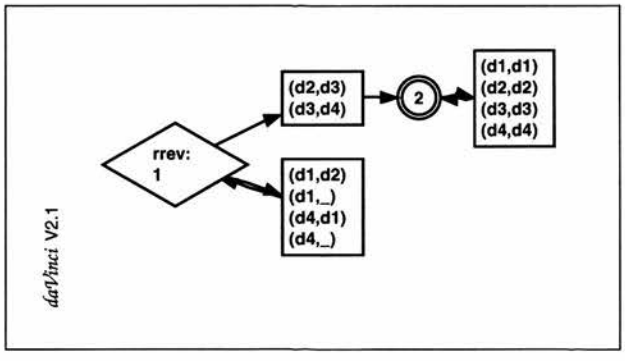


Figure 3.22: The *r* direction of the triangular mesh structure.

$$
\begin{aligned}
d(x.d2) &= d(x).d2|d(x) \\
d(x.d4) &= d(x).d4|d(x) \\
d(x.d1) &= x.d3 \\
d(x.d3) &= x.d1
\end{aligned}
$$

$$
\begin{aligned}
r(x.d1) &= r(x).d2|r(x) \\
r(x.d4) &= r(x).d1|r(x) \\
r(x.d2) &= x.d3 \\
r(x.d3) &= x.d4
\end{aligned}
$$

$$
\begin{aligned}
l(x.d1) &= l(x).d4|l(x) \\
l(x.d2) &= l(x).d1|l(x) \\
l(x.d3) &= x.d2 \\
l(x.d4) &= x.d3
\end{aligned}
$$

Figure 3.23: Equations for the link directions in the triangular mesh structure.

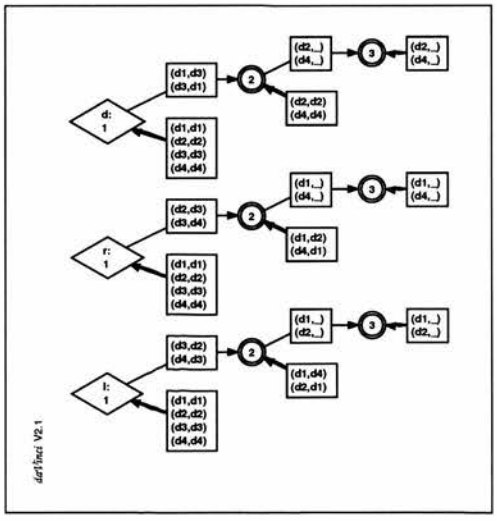


Figure 3.24: 2FSA descriptions for the link directions in the triangular mesh structure.

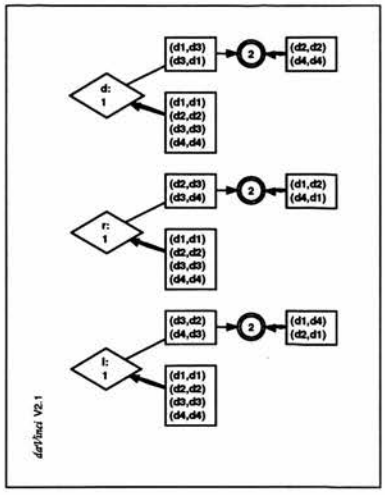


Figure 3.25: Balanced version of the triangular mesh. In this 2FSA description only nodes at the same level are joined.

```
class Balanced_TriangularMesh {

  Tree * d1, *d2, *d3, *d4;
  float data;
  Tree ^ l, r, d;
};
{l:
  States 2;
  Start 1; Accept 2;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
                (d3,d2)->2;(d4,d3)->2;]
        [(d1,d4)->2;(d2,d1)->2;]
  ]
    }
{r:
  States 2;
  Start 1; Accept 2;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
              (d2,d3)->2;(d3,d4)->2;]
              [(d4,d1)->2;(d1,d2)->2;]
  ]
    }
{d:
  States 2;
  Start 1; Accept 2;
  Transitions[
              [(d1,d1)->1;(d2,d2)->1;(d3,d3)->1;(d4,d4)->1;
              (d1,d3)->2;(d3,d1)->2;]
              [(d4,d4)->2;(d2,d2)->2; ]
  ]
    }
```

Figure 3.26: The source code description for the balanced triangular mesh.

3.6 Producing Join Code

The design and maintenance of code that links up the pointers of complex structures can be error prone. If the full description of the link pointers is supplied by the programmer, it would be useful if we could capitalise on this investment and automatically generate code that traverses the structure, linking up the pointers correctly.

We give here a method of producing such *join code*. The code assumes that the basic tree backbone of the data structure has been allocated, with the parent to child pointers correctly set. The *leaf* nodes are those nodes at the base of the structure with no child nodes and all child pointers from a leaf are set to be null. We also assume that the link pointers have been initialised to null.

Given the 2FSA description we want to convert the state machine into a set of recursive functions in C that links up the pointers correctly.

3.6.1 Generating join code

For each link direction we produce code that walks the tree, over every pair of path names keeping track of state within the relevant 2FSA. Nodes are joined when the 2FSA reaches an accept state.

The recursive join code algorithm is given in Figure 3.27. Example join code for the 'n' link direction of a binary tree structure is given in Figure 3.28. Once a binary tree structure has been allocated from a root pointer 'r', we join up the 'n' pointers by calling '`join_n_1(r,r)`'.

Single-valued 2FSAs

As mentioned earlier, the 2FSA of a link direction is a relation that can map a pathname to a number of target pathnames. This is a useful feature of the approach, but causes problems when we try to generate the join code for that direction. In fact in this case, join code is probably inappropriate, the application must decide where to place the links itself. We will later look at a situation where join code is still useful in multi-valued 2FSAs.

When we are producing join code for a link, we may want to insist that the 2FSA describes a single-valued function: each pathname maps to no more than one distinct target node.

We can define a test on a 2FSA F to determine if this condition holds. We construct the following 2FSA SV, and test to see if it is empty.

$$\mathsf{SV} = \mathsf{F}^{-1}.\mathsf{F} \wedge (!=)$$

If SV is empty then the 2FSA is single valued. If it is non-empty, then we must have a pair of pathnames (x,y), $x \neq y$, such that there is a z with $(z,x) \in \mathsf{F}$ and $(z,y) \in \mathsf{F}$. This implies that z is a pathname with two distinct target nodes.

- To build the join code for direction d, with 2FSA D.
- Create a function for each state i in d with prototype:

  `void join_d_i(Tree *a, Tree *b);.`
- The definition of this function is given by:
  - If i is an accepting state then add code that links the pointers:

    `a->d = b;`
  - Now iterate through all transitions from i.
  - If we have a transition $i \overset{\longmapsto}{_{(x,y)}} j$, with neither x or y being padding symbols, we want to call the function for that state in the machine:

    `if((a->x!=NULL)&&(b->y!=NULL)) join_d_j(a->x, b->y);`
  - If we have a transition $i \overset{\longmapsto}{_{(-,y)}} j$, we recurse if the `b->y` pointer is not null:

    `if (b->y!=NULL) join_d_j(a, b->y);`
  - Similarly, if we have transition $i \overset{\longmapsto}{_{(x,-)}} j$, we recurse:

    `if (a->x!=NULL) join_d_j(a->x, b);`

Figure 3.27: Method for generating C code to join nodes linked by direction d.

```
//Join code: generated automatically
// by 2FSA analyser
void join_n_1 (Tree * a, Tree * b);
void join_n_2 (Tree * a, Tree * b);
void
join_n_1 (Tree * a, Tree * b){
  if ((a->l != NULL) && (b->l != NULL))
    join_n_1 (a->l, b->l);
  if ((a->l != NULL) && (b->r != NULL))
    join_n_2 (a->l, b->r);
  if ((a->r != NULL) && (b->r != NULL))
    join_n_1 (a->r, b->r);
}
void
join_n_2 (Tree * a, Tree * b){
  a->n = b;
  if ((a->r != NULL) && (b->l != NULL))
    join_n_2 (a->r, b->l);
}
```

Figure 3.28: Join code for the binary tree structure

Multi-valued 2FSAs

Let us look into a situation where the 2FSA is not single-valued yet the join code is still useful. This occurs in the situation of unbalanced structures where we still require to link to a particular node. If this node does not exist because the structure is not sufficiently deep at that point, then we want to link to the deepest parent (that does exist) of that node.

Suppose we have a 2FSA description where a node a links to a (possibly infinite) number of target nodes b_i. When run over a particular structure, the join code will link a to each b_i in turn. Since the code traverses over the nodes in the structure in lexicographical order, the b_i will be linked in that order. When the code terminates, only the pointer link to one b_i will remain. This b_i will be the last node that exists in that structure.

This situation occurs, for example, in the unbalanced triangular mesh. Take the l direction, linking to the left neighbour node at the same level. If the structure is unbalanced, and there is no such node, then we link to the left node of the parent or the left node of the parent's parent, and so on. The lexicographically last node in the set defined by the 2FSA will be the only node in this set that is a leaf node of the structure. This is the node that we want, so for this example the generated join code does exactly the correct thing.

To summarise, in the multi-valued case, the auto-generated code will join a to the lexicographic maximum of the set b_i. In the situation of link directions in unbalanced trees, this will still be a sensible target for the pointer.

If this is not what is required then either an alternative 2FSA description must be created, or the automatically-generated join code must be rejected in favour of handcrafted code.

3.7 Comparison with other Approaches

As mentioned in Chapter 2 there are a number of existing approaches for describing the shape or linkage patterns in data structures. Not all of them are designed for use in extracting dependency information, but it is worthwhile to compare them with the 2FSA approach outlined in this chapter.

3.7.1 ADDS and ASAP

The ASAP approach [76] uses three different types of axiom to store information about the linkages in the structure.

$$
\begin{array}{lll}
1) & (\forall p) & p.RE1 \neq p.RE2 \\
2) & (\forall p \neq q) & p.RE1 \neq q.RE2 \\
3) & (\forall p) & p.RE1 = p.RE2
\end{array}
$$

Here p, q are any path names and $RE1, RE2$ are regular expressions over A^*. These axioms are then used to prove whether two pointers can be aliased or not. In comparison, our properties are of the form $p.RE1.n = p.RE2$ where $RE1$ and $RE2$ are regular expressions in generator directions only, and n is a link. They are slightly more powerful in that the 2FSA description allows $RE1$ and $RE2$ to be dependent expressions.

We can express each type of ASAP axiom as a 2FSA relation. The regular expression operations $(|, .)$ can be treated using the union and composition operations. The remaining Kleene star operation can be handled exactly if it only contains generator directions. In general, we would need to use the closure approximation method mentioned in Section 4.4.6 to approximate these expressions. This information in the form of 2FSAs could be combined into our dependency analysis.

ASAP descriptions are an improvement on an earlier method, called ADDS [73], which used different dimensions and linkages between them to describe structures. Thus ASAP is more suited to structures with a multi-dimensional array-like backbone, where we assume an underlying tree structure.

If we compare the ADDS/ASAP description for a binary tree with linked leaves (see [9]) with our 2FSA description, we see that our specification system is much more complicated. To justify this, we need to demonstrate that the 2FSA description can resolve dependencies more accurately. Consider the following ADDS description of the binary tree:

```
type Bintree [down][leaves] {
  Bintree *left,*right is uniquely forward along down;
```

```
  Bintree *next        is uniquely forward along leaves;
}
```

This description names two dimensions of the structure `down` and `leaves`. The `left` and `right` directions form a binary tree in the `down` direction, the `next` pointers create a linked list in the `leaves` dimension. The two dimensions are not described as disjoint, since the same node can be reached via different routes along the two dimensions. Now consider the following code fragment, where two statements write to some nodes linked off a pointer `x`.

```
x->l->n->n = ...
x->r = ...
```

Dependency analysis of these statements will want to discover if the two pointers on the left sides can ever point to the same node. This is referred to as the two pointer expressions being *aliased.* Since the sub-directions contain a mixture of directions from each dimension `[down]` and `[leaves]`, ADDS analysis must assume conservatively that there may be a dependency. Using the approach mentioned in Section 3.4.1, and the binary tree 2FSA descriptions in Section 3.4.2, we can produce a 2FSA that accepts all pathnames p for which these two pointers are aliased. This 2FSA, $(l.n.n \wedge r)$, is empty, and thus these writes will always be independent.

3.7.2 Shape Types

Shape Types [80] uses grammars to describe structures, which is more general and powerful than our 2FSA-based approach.

However, many of the questions we wish to ask in our dependence analysis may be undecidable in this framework. Using 2FSAs many of the operations we want to perform have crisp, easily computable solutions. We would lose much of this simplicity using grammars. As we will see later, some of the manipulations we need to employ when using our 2FSA approach to derive dependency information need to be approximated. A more powerful linkage formalism (such as a grammar based approach) is likely to require more complex approximations and heuristics to perform these manipulations.

Also extracting useful parallelisation information from 2FSAs that store dependency information requires much effort, which would be even more difficult if this is in the form of grammar expressions.

The Shape Types work uses the convention that the pointers from a leaf node of a structure point back to itself. This is not really necessary to express the linkages in data structures. If we drop this requirement, we can describe the linkage properties of most of their example structures using our formalism. These include:

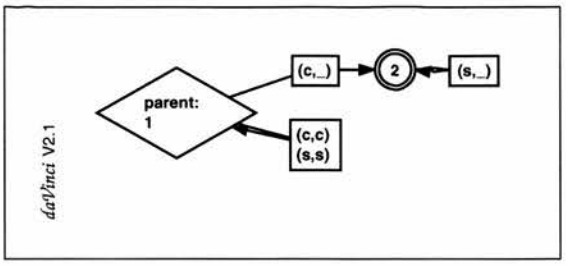


Figure 3.29: The *parent* direction for the left-child, right-sibling tree

- Skip lists of level two. These are the skip lists we mention in Section 3.4.1 but with the restriction that the skip pointers always point two nodes down the list.
- Binary trees with linked leaves. These are identical to the binary tree structure used in this chapter.
- Left-child, right-sibling trees. These permit a variable number of child nodes, by placing them in a list linked from the parent node. These can be considered as a normal binary tree, the left pointer (c) takes us to child nodes, the right pointer (s) to the siblings of that node. The only link direction required is the *parent* direction. This basically removes all of the last s symbols in the path and one of the preceding c directions. This can be expressed as a 2FSA, and is shown in Figure 3.29.

3.7.3 Graph Types

In [79], the 'Graph Types', as described are not directly comparable to ours, being neither a strictly stronger or weaker formalism. They allow descriptions to query the type of nodes and allow certain runtime information such as whether a node is a leaf to be encoded. If we drop these properties from their descriptions, we can then describe many of their data-structures using 2FSAs. This conversion can not be done automatically. Their work does not consider dependency analysis; it is mainly concerned with program correctness and verification. In [60] this is extended to verification of list and tree programs, and hints that this analysis could be extended to graph types.

3.7.4 Symbolic Alias Pairs

In [55] the author aims to extract aliasing information directly from the code. A system of *Symbolic Alias Pairs*, SAPs, is used to store this information. Our 2FSAs have a similar role, but we require that they are provided separately from the program, rather than being harvested directly from it.

For some 2FSAs the aliasing information can be expressed as a SAP. As an example, the information for the n direction in the binary tree can be represented in (a slightly simplified version of) SAP notation as:-

$$(< X(->l)(->r)^{k_1} -> n, X(->r)(->l)^{k_2} >, k_1 = k_2)$$

The angle brackets hold two expressions that are aliased for all values of the parameters k_1 and k_2 that satisfy the condition $k_1 = k_2$. The paper gives an example of a set of paths that are not regular, so SAPs are capable of storing information that a finite state machine cannot. These expressions are not, however, strictly more powerful than 2FSAs, since some 2FSAs can express relations that SAP relations cannot. For example, the alias information held in the 2FSA descriptions for the triangular mesh cannot be as accurately represented in SAP form. This is because 2FSAs can encode substitutions: relations where a sub string has one symbol replaced with another, which SAPs cannot.

3.8 Conclusions

This chapter has described a novel way of specifying the linkages within a particular pointer-based structure. A number of example structures have been specified using this method.

Due to the unfamiliarity with 2FSAs, most programmers will find it easier to design link code directly rather than specifying linkages in 2FSA notation. However extracting linkage information that compilers can use automatically is difficult from unrestricted code. By placing restrictions in the manner in which pointer link code is written it may be possible to derive useful information automatically. Indeed if we restrict linkage code to something close to the join code we generate in this chapter, this process could be reversed to create 2FSA descriptions automatically. This approach has not been pursued any further in this thesis.

If 2FSAs are to be supplied by the programmer, this description places an additional burden, but there is benefit in that some of the tedious code to link up the pointers within the structure can be automatically generated. This reduces the potential error of keeping this code up-to-date with a structure description that evolves as it is being debugged. Alternatively, the descriptions of standard structures could be supplied in libraries from which the programmer can derive their own specific descriptions.

The issue of how to use this information to produce dependency information within the code that uses the structure is treated in Chapter 4.

Chapter 4

Dependency Analysis

In this chapter we look at the dependency analysis that can be performed using the additional information included in the 2FSA data structure descriptions described earlier.

First, the simplified C-like language that we will use to express the recursive programs in this thesis is described. The information we want to extract summarises which sections of the program access which parts of the data structure. Then the method for generating this information and storing it as 2FSAs is explained. We also begin to look for more useful variants of this information, in the form of program dependency and dataflow information.

For some of these programs the dependency information cannot be exactly represented in 2FSA form. In these cases we derive heuristics methods that can be used to derive a useful approximation.

4.1 The Program Model: A subset of C

A fairly restricted programming language for manipulating these structures has been used for this study. This approach seems common for research in this area (see [50] and [28]) although some work has been done with fairly unrestricted versions of languages such as C, for example in [76]. We can justify this restriction if we consider that we are only interested in accesses to and from a single main data structure, and we ignore other language features. The language subset we use must capture at least these elements of the program.

We will use the C language [105], as the basis for the example sequential imperative language throughout this thesis. This is not because C is unique in its use of pointer structures, indeed most languages handle these types. Nor does C handle these structures more cleanly than other languages. In fact C allows pointer arithmetic: pointers can be incremented, compared, even added and subtracted. This allows certain algorithms to work very efficiently, but makes deciding automatically whether two pointers point to the same memory location an undecidable problem (see [106]).

In these respects Java [107], for example, is a cleaner language. Although often touted as a language without the complexity of pointers, in Java all objects are dynamically created, and are accessed by references which are functionally equivalent to pointers. Java does disallow pointer arithmetic. We choose to ignore pointer arithmetic too, since it complicates any analysis without adding any additional expressibility to the language; as mentioned above its main purpose is for the sake of efficiency.

Also C is familiar and pervasive, developing a novel sequential imperative language from scratch to illustrate automatic parallelisation techniques would be excessive and possibly counter productive.

Expressing parallelism has not been a traditional feature of programming languages. Imperative sequential languages such as Fortran and C, were designed with sequential machines solely in mind. Thus certain implementation issues such as the manner in which data structures are laid out in memory are easy to optimise in these languages. However, detailing which parts of a program are independent and can be run in parallel are not.

As mentioned before, we only consider the data accesses to one global data structure. Thus we ignore any dependency analysis that handles standard scalar and array data structures, only considering data dependency *within* the one structure.

We can briefly consider how this 2FSA approach could be applied to a more general situation where there are multiple structures with possible cross linkages. To do this we would need to treat all structures that have some aliasing as part of one global structure, and analyse the completely independent structures separately. Ultimately this approach leads to treating the whole of dynamic (heap) memory as one complex data structure. Forming the appropriate 2FSA descriptions could be difficult if it must be done explicitly by the programmer, but may be feasible if built up automatically from a number of separate programmer supplied descriptions. We consider this again in Section 7.1.2.

The full grammar of the language considered in this thesis is given in Figure 4.2. The terminal symbols IDENT and EMPTY have the following meanings. IDENT is a standard alphanumeric identifier, exactly as used in the C language. EMPTY represents no symbol present, used here since a ***block*** can be empty.

To illustrate the description there is a fragment of typical code in Figure 4.1 which works with one global data structure. This example initialises all the nodes in the tree to the value `1.0`, then traverses the tree using a couple of mutually recursive functions `Traverse` and `Update`.

Data elements are accessed via *pathnames* which correspond to the ***element*** non-terminal in the grammar. These are used in the same manner as conventional pointers. The code consists of a number of possibly mutually recursive functions (***recfunc*** in the grammar). These functions take any number of pathname parameters, and return '`void`'. Only dataflow through

```
main (Tree *v) {
  Init(v->l);
  Traverse(v->l);
}
Init(Tree *i) {
  if (i==NULL) {return;}
  i->l->data = 1.0;
  i->r->data = 1.0;
  Init(i->l);
  Init(i->r);
}
Traverse(Tree *w) {
  if (w == NULL) {return;}
  Update(w->l, w->r);
  w->data = w->p->data;
  Traverse(w->r);
}
Update(Tree *t, Tree *a) {
  if (t == NULL) {return;}
  t->data = a->p->l->n->data;
  if (t->l != NULL) {Update(t->l, a->r);Update(t->r, a->l);}
  return;
}
```

Figure 4.1: A fragment of code from the C subset

recfunclist	::= ***recfunc***
	| ***recfunclist recfunc***
recfunc	::= IDENT '(' ***paramlist*** ')' { ***block*** }
paramlist	::= ***param***
	| ***paramlist*** ',' ***param***
param	::= 'Tree *' IDENT
block	::= EMPTY
	| ***block statement***
statement	::= 'if (' ***cond*** ')' { ***block*** } 'else' { ***block*** }
	| 'if (' ***cond*** ')' { ***block*** }
	| ***instruction***
cond	::= ***element*** '== NULL'
	| ***element*** '!= NULL'
instruction	::= ***element*** '=' ***exp*** ';'
	| IDENT '(' ***callparams*** ')' ';'
	| 'return' ';'
exp	::= ***element***
	| '(' ***exp*** ')'
	| ***exp*** '+' ***exp***
	| ***exp*** '-' ***exp***
	| ***exp*** '*' ***exp***
	| ***exp*** '/' ***exp***
element	::= IDENT
	| IDENT '->' IDENT

Figure 4.2: The grammar of the language of programs we consider for analysis

the structure is permitted. It is assumed that the function that appears first textually (referred to as the *main function*, as in C and actually named 'main' in this example) is the entry point of the program and is given the root path name of the structure as a parameter. Each function may make (possibly recursive) calls to other functions ('Func') using the syntax 'Func(w->d, ...)', where 'd' is any field.

The basic statements of the code are reads and writes to various parts of the structure, denoted ***statement*** in the grammar. A typical read/write statement is 'w->a = w->b', where 'w' is a variable and 'a' and 'b' are words of fields, and denotes the copying of an item of data from 'w->b' to 'w->a', within the structure. Other operations may be performed on the values, such as summation or function calls (free of side effects). Our later analysis ignores the actual operations carried out since it is only concerned with the data accesses performed on the structure.

Note that we do not permit structures to be changed dynamically by pointer assignment. This is a major restriction, and renders out of scope any algorithms that require this. Our approach is only valid for sections of algorithms where the data in an existing structure is being updated, without structural changes taking place. Typically, in a pointer based program a data structure is allocated appropriate memory, initialised, and then a range of structural and non-structural phases are applied. Our analysis only applies to the non-structural phases.

Also, our use of pathnames can be thought of as a more disciplined version of conventional pointers. Allowing traditional unrestricted use of pointers such as assignments to pointers, and pointer arithmetic is therefore undesirable. We neglect loops also, since these can always be converted into recursive function form.

4.1.1 Conditionals

We need to support some form of conditional statements that depend on the state of the data structure. If we do not we may as well perform a symbolic execution of the program and trace its dataflow directly since the executed statements could not vary with a different valued data structure. Even this simplified situation with no conditionals could pose a challenge to static analysis, since with larger programs the information that would be required to be stored and manipulated by such an execution would get prohibitive.

We allow conditional statements that test for null pointers so as to be able to deal with algorithms that handle variable-sized structures. We allow conditionals of the form 'if (pathname == NULL) then ...' or 'if (pathname != NULL) then ...'. We do not perform any sophisticated analysis of the conditional statements, of the type as can be found in [28]. Allowing some statements to be executed depending on a conditional that can only be decided at run-time does restrict the accuracy of the analysis that can be done. For example, any dataflow analysis has the potential to become inaccurate when we cannot decide at compile-time if a particular

```
Traverse(t) {
   if (t == NULL) return;
   Update(t->l);                  //A
   t->data = t->p->data;          //B
   Traverse(t->r);                //C
}
Update(w) {
   if (w ==NULL) return;
   w->data = w->p->l->n->data; //D
   Update(w->l);                  //E
   Update(w->r);                  //F
}
```

Figure 4.3: Sample recursive functions. The A,B,C,D,E,F are statement symbols and not part of the original code.

statement is executed or not. We handle this uncertainty by making the conservative assumption that the bodies of conditionals are executed.

4.2 The Analysis

We will explain the analysis methods in this chapter, using the same piece of code as a running example. The recursive functions in Figure 4.3 operate over the binary tree structure introduced in the last chapter, in Section 3.4.2. We assume the program begins by calling the 'Traverse' function with the root pathname as its argument. The function 'Traverse' recurses down the right-hand side of the tree, copying to that node's 'data' field, the 'data' value from the adjacent (in the p direction) node. As it visits each node it first calls 'Update' on the left-hand sub-tree. 'Update' traverses over the whole subtree, copying all the 'data' values from a nearby node (in the 'p->l->n' direction). Despite being a contrived example, this code demonstrates features typical of codes that manipulate pointer structures; mutually recursive functions, update of fields using values obtained from neighboring or local nodes and recursion in multiple pointer directions.

4.2.1 Control Words and Substitution 2FSAs

We need to distinguish between statements as actual lines of code (the lines lettered with A,B,C,D,E,F in the code fragment above) and *run-time instances* (or just *instances*) of those

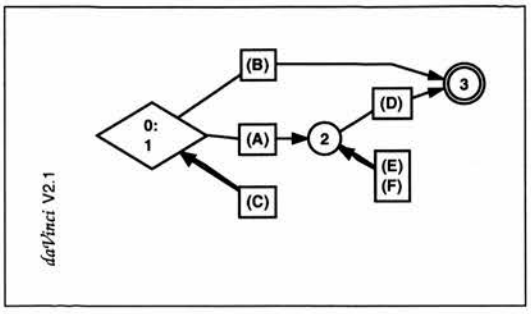


Figure 4.4: FSA that accepts valid control words

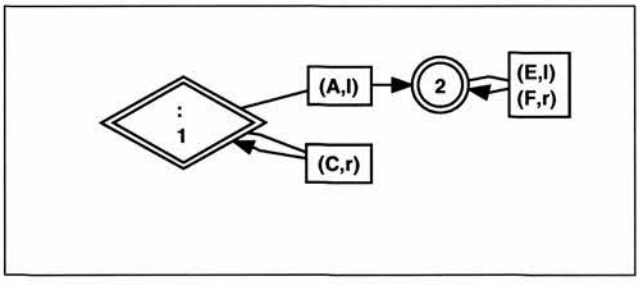


Figure 4.5: 2FSA that maps control words to values of w.

statements. If it occurs within a recursive nest of calls, a statement may be executed a number of times during an actual run of the code. A run-time instance is a particular execution of a statement during program execution. Each distinct run-time instance needs to be given a unique identifier so that we can talk about its dependency properties. These properties will be different for different instances of the same statement.

We also separate statements (and also instances) into *access* statements that read and write into values in the structure, e.g. statement B, '`t->data = t->p->data`', and *call* statements that call another function, e.g. statement C, '`Traverse(t->r)`'. Each instance is named by a *control word*, defined by labeling each source code line with a *statement symbol*, and forming a word by appending the symbol for each function call that has been followed.

For the control word to refer to an access instance it will be terminated by the symbol for an access statement. For our example this means that it will end with either a B or a D. We will use control words that end in a call statement symbol to refer to a whole function call, e.g. the control word C could be used to refer to all of the instances in the whole of the C function call.

Each time the '`Traverse`' function is called recursively from statement C, we append a C symbol to the control word. Source code line B therefore expands to a set of run-time instances, labeled by the set of control words represented by the regular expression $C^*.B$.

A general valid control word for all the control words of the example program is a word from the language given in daVinci notation by the FSA in Figure 4.4. This is quite straightfor-

ward to understand with reference to the original program. State 1 represents the 'Traverse' function, state 2 the 'Update' function, and 3 represents complete valid control words. The C self-transition from state 1 corresponds to the recursive call in statement C, and the A transition to 2 represents the call of 'Update' in line A. The B transition from state 0 to 3 and the D transition from state 2 to state 3 represents the fact that valid control words must terminate with either a B or D to correspond to the structure accessing statements in lines B and D. We could also describe this same set of words by the regular expression $(C^*.(B|A.(E|F)^*.D))$, but here this notation is far less illuminating.

The sequential execution order of the the run-time instances is the lexicographic order (see Section 3.6.1) given from the appropriate order on the statement symbols. This order is simply given by the order the statements occur in the source. Only the ordering of the statements within a function body really counts, since the order that functions are defined in the source is irrelevant as they may be called in any order. This order can be encoded as a 2FSA, which is necessary for the manipulations in Section 4.7, where we need to discover which of a pair of statements occurs first.

We now turn to the data accesses in the code. We term a pathname parameter to a recursive function an *induction parameter*. Each time we call the function recursively from statement A, we append a l direction to the induction parameter w, and for statement C we similarly append a r. The same is true for statements E and F. This information can be captured in a 2FSA for each parameter w, that accepts pairs (c, p), where c is any control word and p is the value of the w, at the point in the program represented by c.

This 2FSA is given in Figure 4.5, which is called the *substitution* 2FSA for that particular induction parameter w. All this 2FSA does is perform a conversion of symbols, mapping C to r, A to l, E to l and F to r.

Note that these substitution 2FSAs convert words in the alphabet of statement symbols to the alphabet of data structure fields. We can either view the alphabet of these 2FSAs as two separate alphabets or one large combined alphabet.

The descriptions for single directions can be used to build composite directions. For instance, the 'read' in statement D requires the composite *read word*: 'p->l->n'. We compose the three 2FSAs, in the manner described in Section 3.2.2 to produce a 2FSA for the whole word.

Given the two pieces of information, the 2FSA for the value of an induction parameter and the 2FSA for the composite word, we can create a composite 2FSA for the complete pathnames of the node accessed by a set of control words relating to a set of access instances.

4.3 Generalised Substitution 2FSAs

Let us generalise the construction of the substitution 2FSA for any induction parameter, once we have a number of parameters for each function the manipulations can get more complex. For each induction parameter, V_i, in each function F, we need to create a 2FSA, labeled F_{V_i}, that maps all control words to the pathname value of that parameter at that point in the execution of the program. The construction of this substitution 2FSA for the variable V_i is as follows.

Definition of F_{V_i}

- Create a non-deterministic 2FSA with a state for each induction variable, plus an additional initial state. State $i+1$ (corresponding to variable i) is the only accepting.
- For each pathname parameter j of the '`main`' function, add an epsilon transition from state 1 to state $j+1$.
- For each induction variable Vk, we seek all call statements of the form:

  ```
  'A: F(...,Vk->g,...)'
  ```

  If '`Vk->g`' is passed as variable m in function F, then we add transition from state $k+1$ to state $m+1$, with symbol (A,g). Here we assume that g is a generator. If g is empty, we use an epsilon symbol in place, then attempt to remove it later as described in the next section. If g is a non-generator we still use the symbol (A,g), but we need to apply the closure approximation described in Section 4.4.
- Determinise the nondeterministic 2FSA.

4.3.1 Left Synchronisation

There is a common case that requires special handling. If a function '`F`' is called with just a bare induction parameter '`Vk`', in the manner of '`F(...,Vk,...)`', then as mentioned in the last section, the substitution 2FSA will contain epsilon transitions of the form (A,ε). The transducer produced is strictly not a 2FSA, but we can attempt to convert it to one. The aim is to remove these transitions by repeatedly composing it with a 2FSA (termed $\mathsf{F}_{!\varepsilon}$) that will remove one ε from the output. This process is called *Left Synchronisation*.

This particular $\mathsf{F}_{!\varepsilon}$ 2FSA is constructed as follows:

Definition of $\mathsf{F}_{!\varepsilon}$

- Create a 2FSA with one state for each generator (and link), plus another three states - an initial state, an accepting final state and an epsilon state.

- Add transitions from initial state. (d,d) symbols where d is a direction (generator or link), make the transition back to the initial state. (ε,d) symbols move to the state for that direction $(d+1)$. $(\varepsilon,-)$ symbols move to the final state.

- Add transitions for each direction state, (direction i corresponds to state $i+1$). Symbol (i,i) leaves it at state $i+1$. (i,d) moves it to state $d+1$. $(i,-)$ moves to the final state.

- Add transitions for the epsilon state. (ε,d) moves to $d+1$. $(\varepsilon,\varepsilon)$ leaves it at the epsilon state. $(\varepsilon,-)$ moves it to the final state.

- There are no transitions from the final state.

If the epsilon transition is not in a loop in the original 2FSA, composing this repeatedly with $F_{!\varepsilon}$ will remove all such transitions. Put in terms of the original code this approach will work if we can bound the number of times that the function can be called in any run of the code. This will occur if the function call does not appear within a recursive nest.

If there are epsilon transitions within loops, this approach will not terminate. In fact it is possible that the required substitution map can not be represented as a 2FSA (*i.e.* the relation is not left-synchronizable). In practice this is uncommon, since recursive calls usually imply progression through the structure to a new node, calling recursively with the same value suggests that an infinite loop is potentially being set up.

For programs with recursion in link directions, these 2FSAs will need to be refined further, possibly needing to be approximated. The next section looks at the issue of approximating in the situation where recursion is over a non-generation direction. This method can also be applied to produce an approximate solution in when epsilon transitions occur within loops.

```
Traverse(Tree *x) {
 ...
 A: Traverse(x->n);
 ...
}
```

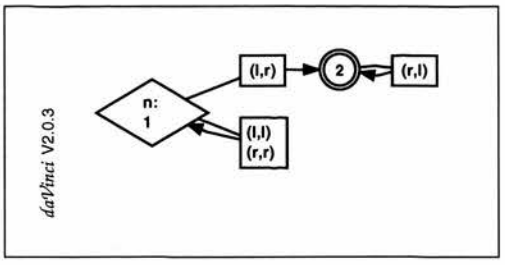


Figure 4.6: A simple recursive function and 2FSA description of the recursion direction n.

4.4 Handling Recursion in Link Directions

In this section we consider the problem of computing access information for functions that recurse in the non-generator directions. In this situation, 2FSAs cannot exactly represent the access information. There are two possible ways of circumventing this problem: use a more sophisticated formal language to store the information, or build an accurate 2FSA approximation. We first look at a method for storing the information exactly.

We will start by illustrating this with perhaps the simplest example of a function that calls itself recursively, appending a non-generator direction to the induction parameter at each call. Consider the example recursive function in Figure 4.6: the data structure `Tree` is the balanced binary tree with `n` and `p` link pointers mentioned in Section 3.4.2. The 2FSA for `n` is also shown in Figure 4.6. The problem we are trying to solve is how to map control words of the form A^* to the value of the induction variable x.

We settle on an approach that approximates the 2FSAs we need directly, skip to Section 4.4.2 for a full description of this method. The next subsection is a digression, looking at possible alternative ways of representing this information. We return to the issue of deriving data dependency in Section 4.6.

4.4.1 Alternative formalisms for storing access information

First we consider the approach of using a more powerful language than 2FSAs to store the information. A range of formal languages have been used in general dependency analysis approaches such as those described in [104]. A good general background to abstract state machines and hierarchies of grammars is given in [108].

There are a number of languages that are a generalisation of the languages described by the 2FSA. Abstract Families of Languages are described in [109]. These families have so called *closure properties*, which means that any language defined as a composite of languages that are members of a particular abstract family, is also a member of that same abstract family. For example, standard regular expressions form an abstract family of languages. This meaning of

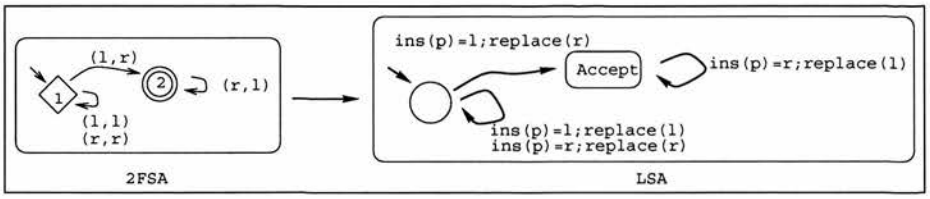


Figure 4.7: Converting a 2FSA description to an LSA.

closure is distinct to the one we use elsewhere in this chapter (see 4.4.2 for the definition) and is similar to the way that 2FSAs are closed under the set of operations mentioned in Section 3.2.2.

Given a particular automaton that accepts a language we can easily enhance it with output transitions to give a transducer that stores a relation, which is what we require for our map from control words to pathnames.

Many of these languages can be classified as state machines that use a variety of different data structures to store the additional state information. Push Down Automata [110], for instance, use a push-down stack as the state storage; *Nondeterministic Real-Time List-Storage Languages* (LSAs) [111] use a list structure for state and Nested Stack Automaton [112] has as a list of stacks. Amongst these formalisms there is a hierarchy, with a formalism A more powerful than another B if A can express a superset of all the languages that B can. There is a related hierarchy of data structures that can be used to store state, here for example a 'list of stacks' is strictly more powerful than a single stack. The more powerful the data structure, the higher up the hierarchy the formalism is and the greater the scope for representation of a particular language.

Amongst these formalisms, LSAs [111] were found to be the simplest that were sophisticated enough to hold the access information for recursive functions exactly. In more detail, an LSA is a finite state automaton with an additional storage provided by a list structure. The list stores a string of symbols, plus a pointer to the *current* symbol. At each transition a number of operations can be performed on the list, the full set of operations is given in Figure 4.9. Operations that to move the pointer `left` or `right`, and `insert` and `replace` symbols are supported. Decisions can be made according to the next symbol in the input stream (`pop(i)`), and the current symbol in the list (`ins(p)`).

Also, rather than just acting as an acceptor of strings, we require a transducer, taking as input the control word and producing the pathname value as output. So we extend the LSA to a transducer by adding the `output(a)` operation, which outputs the symbol a at that transition. We assume that all such output symbols are collected in a list, this list is the pathname.

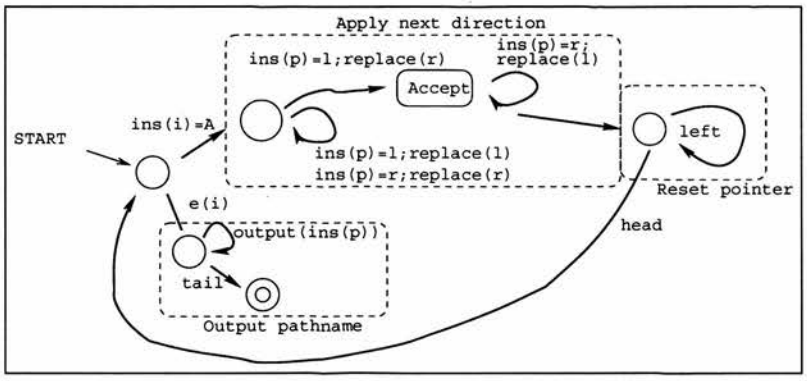


Figure 4.8: LSA for the value of induction variable x.

- `e(i)` - Is input stream empty?
- `pop(i)` - Pop and inspect head of input stream.
- `ins(p)=a; X` - If symbol at pointer of list is a perform action X.
- `replace(a)` - Replace symbol at pointer with symbol and move pointer right.
- `output(a)` - Output symbol a
- `left` - Move list pointer left
- `right` - Move pointer right
- `head` - Is pointer at head of list?
- `tail` - Is pointer at list tail?

Figure 4.9: Operations performed by LSAs

Consider the example code fragment to illustrate how the LSA may be built for a code fragment. We need to create an automaton that will take a control word and produce the set of possible path names for x. In the LSA, we use the list stored as the current value of x. For each link direction, we can convert the 2FSA F into an LSA L_F that applies the direction to the internally stored list. Note that this LSA must have its internal list initialised with a pathname and does not yet output the target pathname.

- Create a state s' in L_F for each state s in F.
- If F has transition $s \xrightarrow[(a,b)]{} t$, add the transition $s' \rightarrow t'$ with operation:

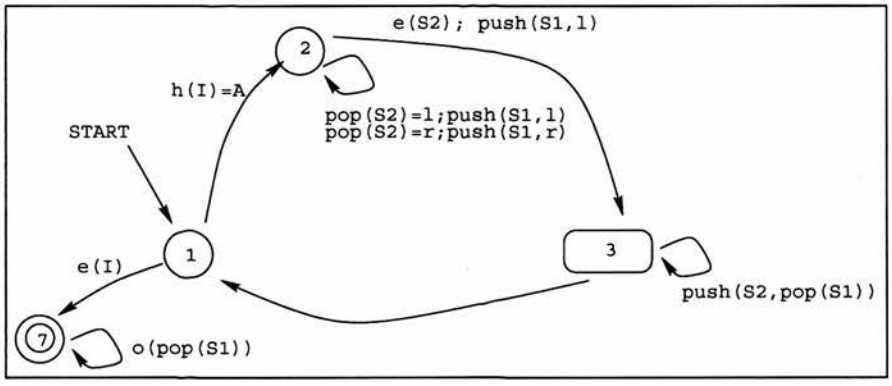
  `ins(p)=a; replace(b).`

Figure 4.7 shows the resulting LSA for the link direction n. It is non-deterministic in that from the start state an l symbol can either be replaced by an r or and l. However only one will lead to an accept state once the whole string has been processed.

For our example code the complete LSA we require needs to append the direction n to the current value of x for each A that is in the control word. Once the end of the control word is reached, it outputs the current list. This full LSA is shown in Figure 4.8, it has been divided into three separate sections that apply the direction, reset the pointer, then finally outputs the pathname. It assumes that the list has been initialised with the appropriate control word. The LSA map for a more general recursive function would probably require appending a different direction to the list for each control symbol.

LSAs do have many of the closure properties that we require for the subsequent analysis phases. We can, for example, compose any 2FSA with these LSAs by appending them to the output stage of the LSA. This would be required when computing the access information for a given statement within a function. Earlier, when producing conflict information between statements we produced two 2FSAs, one that stored the read accesses and another for the writes. Now we have two LSA transducers with these functions. In order to produce conflict information, we need to be able to perform the inversion and composition operations that we can do exactly for the 2FSAs.

LSAs are not closed under composition [113], so we cannot directly compose any two LSAs without some approximation. The best we can do in this case is to use some heuristic methods to approximate the composition operations. A tactic for achieving this would be to approximate the LSA directly with a 2FSA and then proceed with that representation. The advantage is that the composition manipulations will now be over 2FSAs and are readily computed.



- `h(I)` - Pop and inspect head of input stream.
- `e(I)` Is input stream empty?
- `o(x)` Output symbol x.
- `push(stack)`
- `pop(stack)`

Figure 4.10: LSA converted to two stack pushdown automata. There are two stacks, S1 and S2.

4.4.1.1 LSA Approximation using 2FSA

We follow the approach of taking the exact LSA description and approximating it using a 2FSA, by applying a version of the *Pereira and Wright* method described in [114]. This technique produces finite state machine approximations to pushdown automata. A different technique is described in [115], which can be faster in some cases.

A *pushdown automaton* (PDA)is a finite state machine with a pushdown stack storing additional state information. Firstly, we note that every LSA is equivalent to a pushdown automaton with an extra stack: we implement the list with its current pointer with two stacks, one representing the list to the left of the pointer, the other to the right half of the list. See Figure 4.10 for the PDA version of the LSA shown earlier. We can then attempt to apply the approximation method to each stack.

The approximation works as follows. We look at the number of different possible stack configurations for a state in the automaton and if there are only a finite number of configurations we can split the state into that number of states, one for each possible stack. Otherwise we must split the infinite number of stacks into a finite set of equivalent classes of stacks. The method used in [114] neglects portions of the stack that may be arbitrarily repeated. Consider Figure 4.10, the stack S1 at state 2 can contain any string of `l` and `r` symbols, with any portion repeated. In fact no states place any restriction on the stack contents, and any portion may be arbitrarily repeated. Thus the LSA approximates to one where we ignore the stack contents completely. This turns out to be a very poor approximation indeed.

We will not pursue the use of LSAs any further in this thesis. We will instead use a more direct approach, one of producing an approximate 2FSA relation directly from the code description. We turn to this next.

4.4.2 Approximations of 2FSA closures

We return to the problem of approximating a link direction recursion by directly using a 2FSA relation. Recursion in one direction can be approximated by a number of steps in that direction with a 2FSA that approximates any number of recursions. This sort of approximation will however give only coarse-grained access information for a whole nest of recursive calls. We will consider first the example recursive function, then show how this can be used to produce approximate information for any nest of recursive calls.

There is an analogous problem for analysis of dependencies in array-based programs, in which array analysis relations are between integer tuples rather than words. Transitive closures of these relations are required (in the cases of loops and recursive functions) and the problem of computing them is dealt with in [116]. This array work manages to prove exactness for these closures, a problem we will consider later in Section 4.4.3. Note that the closure computation

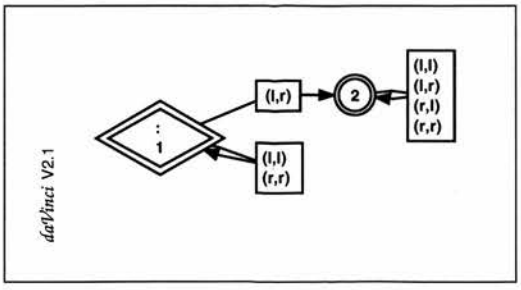


Figure 4.11: The closure relation of the `next` direction. This is designed 'by hand' from the definition.

itself can still be exact despite the fact that it is an approximation to the true access information of the program.

Let us examine the issue of *closures* in more detail, and return to our running example of recursing in the n direction. In the second iteration of the function, the value of x will be `n->n` appended to its initial value and iteration k can be denoted as $\mathtt{n}^k$. This can be computed for any particular value of k, by repeatedly using the composition operator, although the complexity of this 2FSA and the composition will become unmanageable for large k.

Definition 6 *The* transitive closure *(or just* closure*) of the direction p, written p^*, is defined as*

$$p^* = \bigvee_{k=0}^{\infty} p^k$$

Definition 7 *R is any relation on pairs of words. A 2FSA S is a* safe approximation *to R, if $(x,y) \in R$, then $(x,y) \in S$. It is* exact *if additionally $(x,y) \in S \Rightarrow (x,y) \in R$.*

The importance of this safety property is that when we apply it to access 2FSAs, it implies that no dependencies will have been removed, although if it is not exact, there will be some spurious ones added. It is worth noting that safety does not always imply that the approximation is useful; a relation that accepts every pair (x,y) is safe but will probably be a pretty poor approximation.

In the example of the `n` direction, we can work out directly from the definition what the closure relation should be. If we visualise the tree as being balanced, and the n pointers linking nodes across the structure, the closure relation will link x to all nodes on the same level whose path names are lexicographically larger (taking $r > l$) than x. This relation is given in Figure 4.11 and is directly representable as a 2FSA, and the approximation can therefore be exact.

It is important to note that it may be impossible to find an exact 2FSA closure for a particular 2FSA description. We can easily construct an example of a 2FSA whose closure is not

Theorem 1 *If R and p are relations such that $R \supseteq R.p \vee [=]$, then R is a safe approximation to p^*.*

Proof 1 *Firstly, $R \supseteq R.p \vee [=] \Rightarrow R \supseteq R.p^{k+1} \bigvee_{i=1}^{k} p^i$, for any k (Proof: induction on k). If $(x,y) \in p^*$, then $(x,y) \in p^r$ for some integer r. Applying above with $k = r$ implies that $(x,y) \in R$.*

Figure 4.12: The test for a relation R to be a safe approximation to the closure of p. The proof below verifies the properties we require.

exactly representable by any 2FSA in the following way. We define a relation R over the alphabet a,b, such that it accepts $(x.a.b.y,x.y)$ and $(x.b.a.y,x.y)$, for any strings x and y. Thought of as a 2FSA (denoted by R), it simply removes one pair of consecutive a and b (or b and a) from the input string (we will omit the full description of the 2FSA R for the sake of brevity).

We now show that the closure of this 2FSA has no representation as a 2FSA. We proceed by contradiction: assume that this closure (R^*) is also representable by a 2FSA. Let L be the 2FSA that accepts (x,ε), for any x. Then, the relation $P = R^* \cap L$ will accept (x,ε), if, and only if, x contains the same number of as and bs. Thus we can use P to build a 1FSA that only accepts strings that have the same number of as and bs. Such a language is not regular [1] which is a contradiction and therefore R^* is not representable by a 2FSA.

Although the closure will not always be representable as a 2FSA, we will still need to approximate it safely as one. The aim therefore is to develop a test such that given a closure p^* to be computed and a candidate closure 2FSA R, we can test whether R is a safe approximation to that particular closure. This approach permits us to use any heuristic method to produce R, and then check safety separately, see Figure 4.12 for a full description of this test.

Therefore, given a 2FSA, R, that we suspect might be a safe approximation, we can test for this by checking if $R \supseteq R.p \vee [=]$. This is done by forming $R \cap (R.p \vee [=])$, and testing equality with $R.p \vee [=]$.

4.4.3 Exactness

We turn briefly to the issue of whether a particular approximation is exact. It is useful to know this, since the approach already introduces approximation when we gather the dependencies of these function calls together and the programmer can be informed of any additional inaccuracy.

Note first that we cannot, in general, directly use the test $R = R.p \vee [=]$ as a test for the approximation being exact. However an extra property of the link relation, *boundedness*, will let us use this same test as a genuine test for exactness. First we define boundedness:

[1] see any standard text that deals with regular expressions, such as [117]

Definition 8 *A relation p is* right bounded *if, for each y there exists an integer k_y such that for all x, $(x,y) \notin p^{k_y}$.*

Similarly p is left bounded *if, for each x we have a k_x such that for all x, $(x,y) \notin p^{k_y}$.*

A relation is called bounded if it is either left or right bounded.

If we describe right boundedness in terms of the relation describing a link pointer n in a data structure, this means that for any node there is a bounded number of nodes linked to that node via a chain of n pointers. It is important to note that the k can vary for each particular y, and may still be unbounded over all possible y. In fact, if it were fixed or bounded by k' , then we could simply compute the transitive closure by only computing as far as the $k'th$ term. This occurs, for example in the balanced triangular mesh where the d direction is a self inverse, if we travel twice in that direction we get back to the initial node. In this case, we can calculate the closure just by taking $k' = 1$.

Boundedness (of either the left or right variety) is a common property of the link directions in the data structures that we consider. Considering right boundedness, a counter-example would be a 'parent' pointer in a tree, linking each child node to its immediate parent. There is an unbounded chain of parent pointers linking all nodes in the tree to the root node. However, left boundedness implies that there is no infinite set of nodes linked from a node via a chain of link pointers. Since the 'parent' pointer only links to nodes strictly above it in the tree, it is left bounded.

The next and previous pointers in a balanced binary tree only link nodes on the same level, so they are both left and right bounded. For the unbalanced tree, where a next pointer may link to a node in a level closer to the root of the tree, we have left boundedness for both link directions (`next` and `prev`) and similarly for the l and r directions in the triangular and rectangular mesh structures. So all our example structures have the boundedness property.

Constructing 2FSAs that do *not* have the boundedness property is straightforward. An example would be a 2FSA that linked every node in the structure to every other node. Such pathological links are unlikely to be explicitly designed by a programmer. It is unknown whether there are any genuinely 'useful' unbounded link 2FSAs that correspond to likely pointer linkages, the experience gained from constructing 2FSA link descriptions during this work suggests not. However, the nature of such unbounded links would probably make any dependency analysis performed on them too imprecise to be useless.

We now apply the boundedness property to give a test for whether the approximation is exact.

Theorem 2 *If R and p are relations such that $R = R.p \vee [=]$, and p is right bounded, then $R = p^*$, and the approximation is exact.*

Similarly, if p is left bounded and we have $R = p.R \vee [=]$

Proof 2 *If p is right bounded and* $(x,y) \in R$, *then* $(x,y) \in R.p^{k_y} \bigvee_{i=1}^{k_y-1} p^i$. *Since* (x,y) *cannot be in* $R.p^{k_y}$, *it must be in* p^i *for some* $i < k_y$. *So* $(x,y) \in p^*$. *The proof proceeds similarly for left boundedness.*

Thus if a 2FSA is bounded then we have a test for the exactness of the closure approximation. Unfortunately, we cannot automatically use this theorem to ascertain that an approximation is exact, as we know of no general method to verify that a relation described by an arbitrary 2FSA is bounded. However, all the link directions we consider are bounded, so we can apply this test to all our example structures.

In summary:

- We cannot directly compute the transitive closure of 2FSAs.
- We can test whether a given 2FSA is a safe approximation.
- If we know that the original one is bounded we can test whether the approximation is exact.

We now need to consider how such a heuristic approximation can be created.

4.4.4 2FSA approximations

To introduce the heuristic methods we use to compute closures, first recall how `n.n` is constructed by the composition method outlined in Section 3.2.2. In effect, we run two finite state automata at once, feeding the output of the first into the input of the second. The states of the combined automata are pairs of states, with the output being the output of the second automata.

To create an automata for the transitive closure we run abstractly an infinite number of automata, feeding the output of automaton number n into the input of automaton $n+1$. The current state is therefore an infinite string of the states of the individual automata. With this in mind, we can encode the transitions between infinite state strings as a 2FSA, and define a 2FSA map from one of these states to the next under each specific input symbol.

There are a couple of caveats to the notion of treating this strictly as a 2FSA. It differs from one in that the input and output strings are infinite, and there is no notion of the machine accepting, just a (possibly non-deterministic) map from one state to the next. We will use the alternative term *State Transition 2FSA* or *ST2FSA* to emphasize these differences.

4.4.5 State Transition 2FSAs

For each input symbol i and for each potential output symbol o, from the original alphabet of generators, we can define the ST2FSA that converts the current state of the infinite collection of

- The alphabet has one symbol for each state of T, plus one representing the fail state, F.
- Create a nondeterministic 2FSA D with a state for each generator, plus one for the padding symbol and an additional fail state 0. (We do not bother yet with initial or accepting states.)
- Deal with each generator or padding symbol state a. If T has transition $x \xrightarrow[(a,b)]{} y$ then add transition $a \xrightarrow[(x,y)]{} b$ to D. If there is no such transition for all b then add transition $a \xrightarrow[(x,F)]{} b$.
- Deal with the fail state, 0. Add transitions $0 \xrightarrow[(x,F)]{} 0$ for all symbols x.

Figure 4.13: Creation of the dual D of 2FSA T

- Create an alphabet twice the size of the one for the dual, consisting of barred and unbarred versions of each of the symbols.
- Create a nondeterministic 2FSA $\mathsf{ST}_{(i,o)}$ with the same number of states as the dual. Make state i initial and all states accepting.
- Treat each state a. For each transition $a \xrightarrow[(x,y)]{} b$, add a transition $a \xrightarrow[(x,y)]{} b$. Additionally, if $b = o$ then add transition $a \xrightarrow[(\bar{x},y)]{} b$. Otherwise, add transition $a \xrightarrow[(\bar{x},y)]{} b$.

Figure 4.14: Creating the ST2FSA for a symbol pair (i, o)

automata, into the appropriate next state. We also introduce the notion of whether a particular automaton in our infinite collection is *active*. All automata start out being active. If, however, we are producing the ST2FSA for output symbol o and a particular automaton does not output that symbol for a given transition, then it becomes inactive. Encoded along with the string of states is a flag for each automaton that indicates whether that machine is active. We encode this by extending the alphabet of the ST2FSA to be the set of states in the 2FSA with an additional fail symbol representing the fail state, plus 'barred' versions of each state to show that the machine is still active. Thus the alphabet for n is $\{1,2,3,\bar{1},\bar{2}\}$. Symbol 3 represents the fail state which has no barred version since it is irrelevant whether it is active or not.

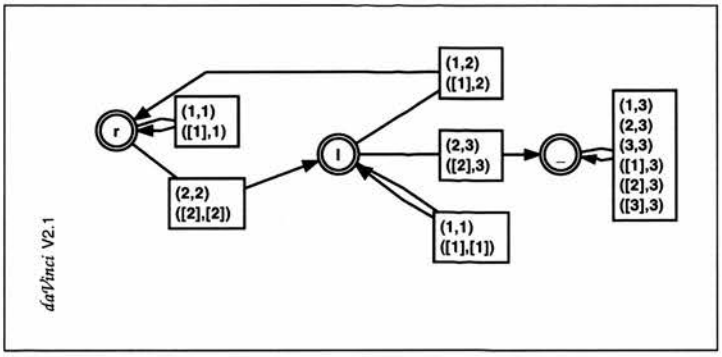
Much of the information in the ST2FSAs is the same for all pairs of generators (i,o), just differing in the barred or unbarred transitions. For the sake of simplicity we create this once in the *dual* of the 2FSA F. Details of its construction are given in Figure 4.13. We then use this dual to build the full set of ST2FSAs (see Figure 4.14 for how each is constructed). In fact we can collect together the ST2FSAs for each output symbol o since they only differ in which state is initial.

These two combined ST2FSAs for n are given in Figure 4.15. Note that the ST2FSAs are drawn up using almost identical notation to the 2FSAs. The only differences are that there are no non-accepting states (all are marked with a double border) and there is no initial state, the initial states for each symbol are instead marked with that symbol. For example, for the input symbol l and output r, we use the the r ST2FSA, beginning with the l state as the initial state. For input r, output r we use the r ST2FSA again, but begin with the r state instead. Also, the barred versions of symbols are drawn in square brackets.

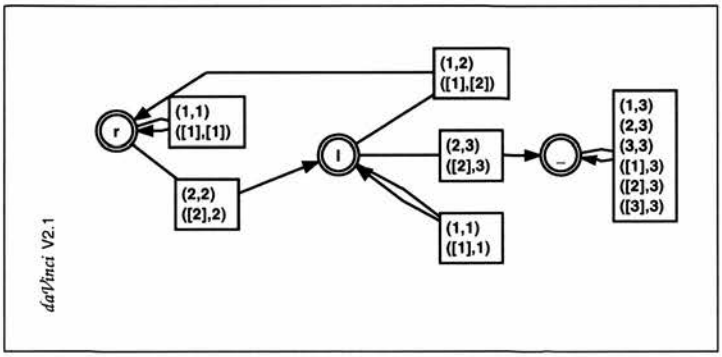
These two ST2FSAs are similar, with the only differences between them being changes between barred and un-barred versions of symbols. For example, in the left hand side l ST2FSA, there is a transition from l state to r state with symbol pair $(\bar{1},2)$. This represents the fact that for the n 2FSA, in state 1 we can take l as an input and move to state 2. This outputs an r, not an l and this is the l output ST2FSA. So the barred symbol becomes and unbarred one, from $\bar{1}$ to 2.

In the right hand r ST2FSA, however, this symbol pair is $(\bar{1},\bar{2})$. The bar is kept since in the n 2FSA, the output is r, which is what is required for the r ST2FSA. Thus this state remains active.

The ST2FSAs completely describe the closure of the direction. They depict the transitions between states represented as strings of states from the original link 2FSA. In our ongoing example, the initial state in next is 1, so the initial state of the ST2FSA is a string of $\bar{1}$ symbols. We represent each of these states as a 1FSA, and apply the different ST2FSAs for each pair of symbols to it just as we would compose a 1FSA and a 2FSA, as described in Section 3.2.2. We



(a) ST2FSA for output of l



(b) ST2FSA for output of r

Figure 4.15: The two ST2FSAs for the direction `next`. The barred symbols referred to in the text as $\bar{k}$ are drawn as [k] in this figure.

can then generate a possibly infinite number of states, and transitions between them. We can think of it as a 2FSA if we additionally consider the following:

- The full description may be unbounded in the number of states it contains. We return to this in the next section.
- Accept states. If state a is accepting in the original machine, any 1FSA state accepting a string containing the symbol $\bar{a}$, will be accepting.
- Failure states. Call any 2FSA state that accepts no strings with any barred symbols a *dead state* since it can never lead to an accept state. For the sake of simplicity we bundle all these dead states into a single failure state.

A state can still be a failure state if is not dead, since we may have a state whose transitions never reach an accept state, yet still has barred transitions within it.

4.4.6 Collapsing to a 2FSA

Note that at this point, as far as computing the transitive closure of a 2FSA is concerned we have not yet made any approximations. Since there may be an infinite number of states we cannot use this representation as an effective way of deciding whether any two words are related via the transitive closure. For words of length less than k say, we could build a 2FSA that answered this question, but longer words would require a different, larger 2FSA.

The method that we have outlined this far creates a potentially infinite automaton. The problem now lies in collapsing a infinite group of states into a finite subset. We use a number of techniques to prune down to a finite number of states which we hope will serve as an approximation for the infinite one. We have experimented with two novel methods. Both of them rely on the fact that we can detect a transition to a fail state.

In the following section we will approximate the closure of a 2FSA D. We build the 'infinite' state machine, I, as described previously. We then approximate this by a finite state machine, M.

4.4.7 Complete States Method

This method takes a subset S_n of n states of the infinite automaton. It then converts each transition that moves to a state not in S_n into a transition to a state within S_n. The following steps outline the method:

1. Choose a maximum number of states n. Start from the initial state and generate all target states for all transitions. Combine all fail states into one unique fail state F. Repeat until we know the transitions from a set S_n of n states.

2. Define the set of *complete states*, E, as the set of all states whose transitions are wholly within S_n:

$$E = \{s \in S_n : \forall \quad \text{transitions} \quad s \longmapsto y, y \in S_n\}$$

   We call states that are not complete *incomplete*.

3. We require that all $x \notin E$ to be accepting. If incomplete states are non-accepting, then we can do one of two things. We can repeat the whole procedure using an $n' > n$, to give us a new set of incomplete states which may now be accepting. Alternatively, we set all incomplete states to be accepting.

4. We next patch up all transitions that lead to states not in S_n. If we have a transition $s_1 \overset{\longmapsto}{(x,y)} s_2$ and $s_1 \notin E$ and $s_2 \notin S_n$, then we remove this transition and replace it with $s_1 \overset{\longmapsto}{(x,y)} s_1$.

5. We now alter all transitions from incomplete states to non-failing states. If there is a transition, $s_1 \overset{\longmapsto}{(x,y)} s_2$, and $s_1 \notin E, s_2 \in S_n, s_2 \neq F \neq s_1$, then we replace it with a transition $s_1 \overset{\longmapsto}{(x,y)} s_1$.

6. We can then minimise M, using standard finite state machine minimisation algorithms. For example we may be able to further merge states of M without affecting the described relation.

The 'Complete States' method was a relatively simple first-pass solution, with the property that it will always terminate for any 2FSA input. It was tested on our sample link directions for the binary tree, rectangular mesh and triangular mesh. It produced exact closure approximations for the next direction for the binary tree and all link directions for the rectangular mesh structures. However it failed for the triangular mesh, producing 2FSAs that were not safe closures. The following method was an attempt to produce a heuristic that could cope with these 2FSAs.

4.4.8 Failure Transitions Method

The following steps outline the method.

- The *failure transitions* FT_s for a state s is the set of symbols that leads to the failure state from that state.

$$(x,y) \in FT_s \iff s \overset{\longmapsto}{(x,y)} F$$

  Two states s and t are *failure transition equivalent* if $FT_s = FT_t$.

- Begin generating a set of states as before.
- If a new state t is failure transition equivalent to an existing one s, we fold them together into one state. We handle other transitions is the following way. For every state p, we map any transition $p \xrightarrow[(a,b)]{} t$ to transition $p \xrightarrow[(a,b)]{} s$. We then assume that t has identical transitions to s and delete state t.
- The process terminates when there are no new states that are not failure transition equivalent to existing ones.

It should be noted that this process will always terminate, since there is a finite number of sets of failure transitions. However this number of states is large, approximately $2^{(p+1)^2-1}$, where p is the number of generator symbols. Even for the binary tree example, with $p = 2$, this is 256 possible states, which is far too many to generate in a practical amount of time. Therefore, in order to ensure termination in a reasonable length of time it is necessary to bound the number of states in the derived 2FSA. If we do not reach termination when this number of states has been created, we can terminate the process by applying the Complete States Method directly at this point. Thus a complete practical closure algorithm will rely on both methods outlined in this section.

In practice, however, if was found that this 'Failure Transition' method terminated on all link direction examples on which it was tested, without requiring application of the Complete State method. Figure 4.16 shows the closure of the 'next' direction, and Figure 4.17 the l directions from both the triangular mesh and the rectangular mesh structures. The 2FSA of the closure of the 'next' direction is possible to understand directly, it links each pathname to all the other pathnames at the same level of the tree (*i.e.* of the same length) that are to the right (*i.e.* the first different symbol in the pathname is an l replaced by an r). The other closures are harder to verify directly, we must trust on the following automated checks.

These computed closures are all found to be safe approximations, including the triangular mesh links for which the Complete States method failed. Since they are all bounded, we can additionally apply the test for exactness mentioned in Section 4.4.3. All these examples are found to be exact.

4.4.9 Generalising to any function call graph

So far we have just looked at a simple recursive function that calls itself in a non-generator direction. We now extend this simple case to any general nest of recursive calls.

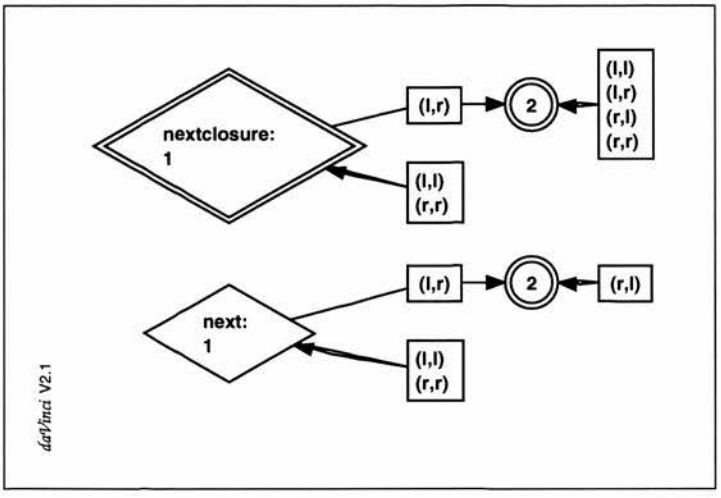


Figure 4.16: The `next` direction from the binary tree structure, with its closure.

We return to the 2FSA expression for the value of an induction variable, x, exactly as derived in Section 4.3. We now assume this contains link recursions somewhere within it, so the 2FSA has a transition with a link symbol.

We now use this to build up an expression for the possible values of x. This approach uses *Arden's rules* (see [117]) which are designed for the purpose of converting an automaton to an expression.

The expression for the value of x at each state of the automaton is built up as follows. If state t has transitions $u_i \rightarrow t$ with symbol (A_i, d_i) for $i = 1 \ldots n$ then we add the expression:

$$E_t = (E_{u_1}.d_1)|\ldots|(E_{u_n}.d_n)$$

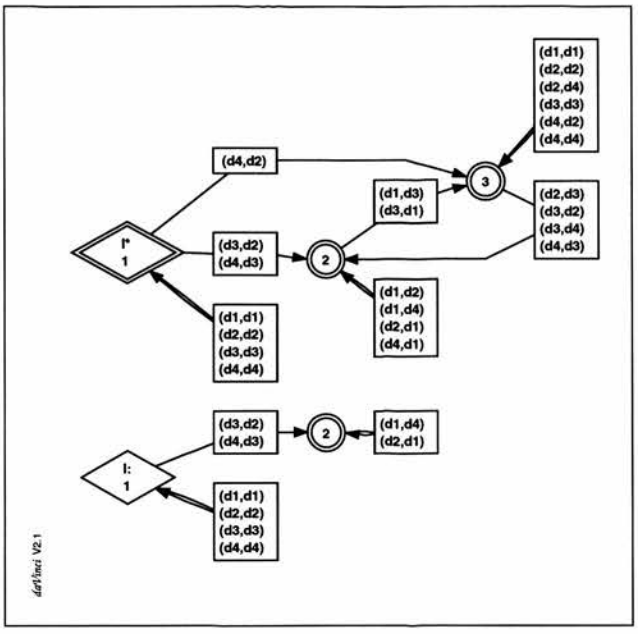
Arden's rules allow us to solve a system of i equations for regular expressions E_i. In particular, we can find the solution of a recursive equation $E_1 = (E_1.E_2)|E_3$ as

$$E_1 = E_3.(E_2)^*$$

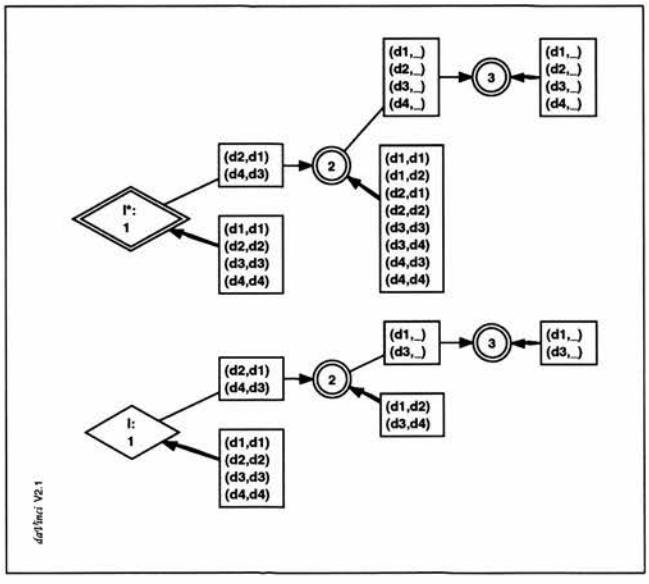
The system of equations is solved by repeatedly substituting one expression in another. When a recursive expression of the above form is produced, it is eliminated by using the above formula.

The output of is a regular expression for the values of x. We can then compute an approximation for this operation using the OR, composition and (possibly approximate) closure manipulations.

The application of this method in practice is next described with an example. The code in Figure 4.18 operates over the binary tree with linked levels described earlier. The 2FSA for the value of the induction parameter x is given in Figure 4.19.



(a) The l direction from the balanced triangular mesh structure.



(b) The l direction from the unbalanced rectangular mesh structure.

Figure 4.17: Some link directions and their computed closures.

```
main (Tree *node) {
  if (node!=NULL) {
    update(node); // A:
    main (node->l); // B:
    main (node->r); // C:
  }
}
update(Tree *w) {

  if (w!=NULL) {
    sweepl(w); // D:
    propagate(w); // E:
  }
}
```

```
propagate(Tree *p) {
  if (p!=NULL) {
    p->data = p->l->data + p->r->data;
// F:
    propagate(p->l); // G:
    propagate(p->r); // H:
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
    x->n->data = x->data; //
I:
    sweepl(x->n); // J:
  }
}
```

Figure 4.18: The program for a set of recursive functions, used to demonstrate the solution of induction variable in a general nest of recursive calls

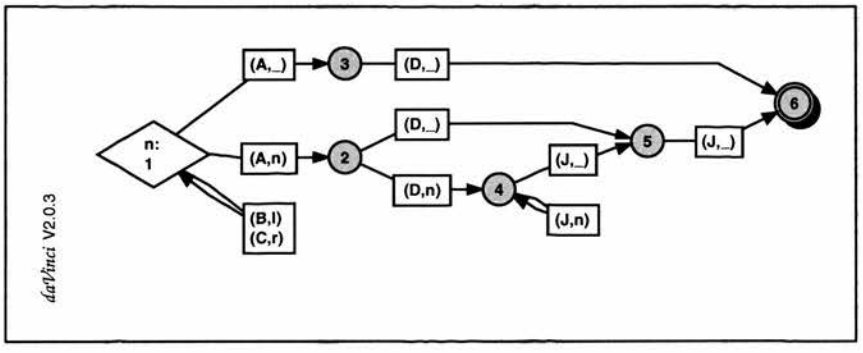


Figure 4.19: The 2FSA map for the value of x at each program point for the code in Figure 4.18. The tainted states are shaded. Note that it contains transitions with the link direction n.

Expression	Tainted	Solved
$E_1 = E_1.l \mid E_1.r$		•
$E_2 = E_1.n$	•	•
$E_3 = E_1$	•	•
$E_4 = E_2.n \mid E_4.n$	•	
$E_5 = E_2 \mid E_4$	•	
$E_6 = E_3 \mid E_5$	•	

Figure 4.20: The expressions to be solved for the value of x for the program in Figure 4.18.

Expression	Tainted	Solved
$E_1 = E_1.l \mid E_1.r$		•
$E_2 = E_1.n$	•	•
$E_3 = E_1$	•	•
$E_4 = E_1.n.n.(n)^*$	•	•
$E_5 = E_1.n \mid E_1.n.n.(n)^*$	•	•
$E_6 = E_1 \mid E_1.n \mid E_1.n.n.(n)^*$	•	•

Figure 4.21: The set of equations solved

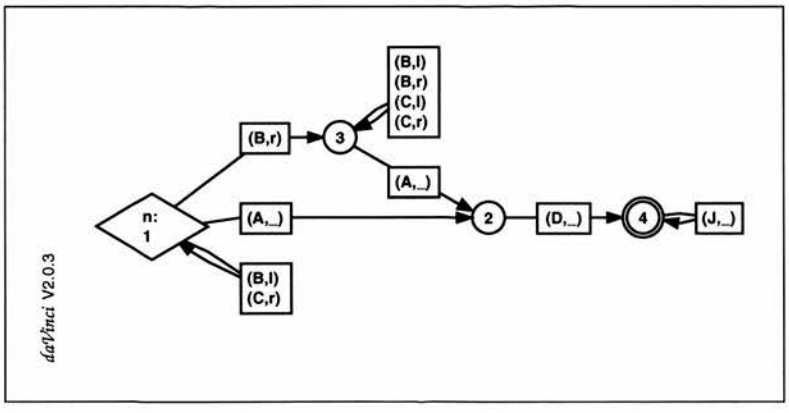


Figure 4.22: The approximate mapping from control words to the value of x

The initial set of equations derived from this are given in Figure 4.20. If one of the d_i in expression E is a non-generator direction then we mark E as *tainted* to indicate that it must be part of the approximation. In addition we also mark all the states reachable from that state as tainted. If an expression is only in terms of untainted expressions then we mark it as solved. We substitute and eliminate the expressions until all are solved. After substituting E_2 in E_4, and eliminating the recursion we obtain $E_4 = E_1.n.n.(n)^*$. The $(n)^*$ closure is of course computed using the previous method. The final set of expressions is given in Figure 4.21.

We can simplify the last one to be $E_6 = E_1.(n)^*$. We next need to turn this solution into a 2FSA that maps control words to values of x. We truncate the original 2FSA so that only the untainted portion is accepting. The original state 6 is the only accepting one, so all we need to do is to append the 2FSA for $(n)^*$ to the truncated 2FSA. Then we tidy up the result by making all control words for statements in the function map to this approximate set of values for x. The final result is given in Figure 4.22.

4.5 Effects of Approximation

Approximation occurs in two separate places in the analysis involved with recursion presented in this chapter. Firstly, a recursive function in a link direction is replaced by a single transitive closure of that direction. Secondly, that transitive closure is approximated by a 2FSA relation. It is worthwhile looking at the effects of these approximations.

The first approximation is the most significant. In essence, it replaces a whole nest of recursive function calls with one function that models the accesses of the whole set. For programs without these link recursion constructs, the access information is exact, with precise information about which control word accesses which parts of the data structure. In the presence of these link recursions, we have much lower resolution information. In the worst case, with a link recursion occurring in the top level function of a program, the access information of every statement in the program will be merged together.

As discovered in Section 4.4.1, to exactly represent this information requires a more powerful formalism. If we want to retain the 2FSA representation of access information this approximation is the best we can do. An alternative approach to the one taken by this work would pursue the LSA formalism and discover whether this would lead to the useful parallelisation information we manage to extract in this thesis.

In practice, the second approximation has much less effect. This is because, for the examples considered here, these closure approximations were found to be exact. Thus no information is being lost in this phase of approximation. However, for more complex compound recursion directions the heuristic closure approximation may be much less precise. This could lead to poor dependency information. However, this cause of approximation can at least be easily detected, allowing the programmer to know where the imprecision in the analysis lies.

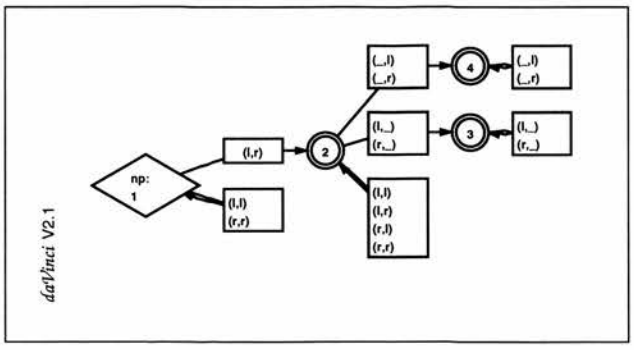


Figure 4.23: The 2FSA $n.n*$.

4.5.1 Other Applications of Closure Computation

Although we have approached the computation of 2FSA closures for the purpose of deriving access information, there are other ways we can use this operation. We consider a couple here, both are still related to producing dependency information.

4.5.1.1 Direct use of dependency information

If we consider the unbalanced binary tree and look at answering some direct questions about the dependencies within. This approach is similar to the kind of manipulations we performed in Section 3.7.1.

We take as an example the n-body example in the ADDS paper [75]. Here it is required to prove that from all points x, travelling in the n direction will never loop around. In other words we need to show that $x.n^k$ are distinct for $n > 0$, for every x. We can do this by forming the 2FSA, $n.n*$, (or n^+ in standard regular expression notation) and checking that it does not map any path to itself, the relevant 2FSA is shown in Figure 4.23. From direct inspection we can see that it will not accept any pairs of paths of the form (x,x). [2]

Therefore all nodes we can visit in the n direction will be distinct, and the update of each node can be run in parallel. This is an improvement on the ADDS approach, since we are not required to explicitly declare the n direction to be 'uniquely forward' as they do. This property can be automatically derived from the general link descriptions required by our 2FSA approach. [3]

[2] We automate this test by performing an AND operation with the 2FSA that accepts equal paths, and checking for an empty 2FSA.

[3] It should be noted that our descriptions are more complex than the ADDS ones in the first place, but once the work has been done to produce them, these kinds of manipulations can be done directly.

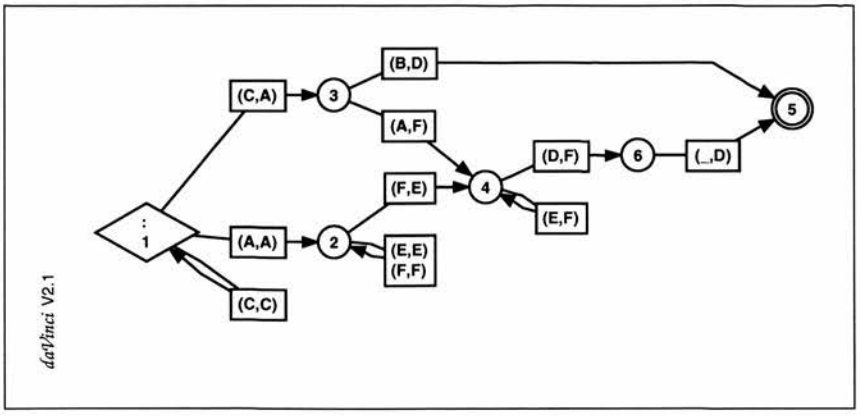


Figure 4.24: The conflict 2FSA for the program in Figure 4.3.

4.5.1.2 Dataflow computation

The 2FSA D (described later in Section 4.7) maps each statement to earlier statements on which it is dependent. *Dataflow* information allows each statement to be mapped to every earlier one that conflicts, and thus the full flow of data throughout the program can be produced. If we could form the transitive closure of D, then we would have full information about the dataflow in the program.

This idea was applied to a couple of test programs. First, the simple example in Figure 4.3 with 12 lines of code and 6 control symbols. The number of control symbols is the best measure of the size of a piece of code under the system we use here. Second, a more complex one in Figure 4.18 with 29 lines of code and 10 control symbols. All timings are taken from a 866MHz Pentium machine with 128MB memory.

Taking the program in Figure 4.3 first, the conflict 2FSA for this program is given in Figure 4.24. It has only six states, as is about as simple as we can expect a conflict 2FSA to be. Once we apply the closure algorithm to it we get the 2FSA in Figure 4.25. This is still quite manageable in size with 9 states, and only takes 7.5 seconds to compute. However the safety check shows this to be an unsafe approximation to the correct transitive closure and hence incorrect as dataflow information.

For the Figure 4.18 program, which still has only 29 lines of code, the conflict 2FSA is much more complex with 39 states. It is given in Figure 4.26. Running this through the closure algorithm took approximately 629 seconds. The resulting 2FSA has 16 states, and is pictured in Figure 4.27. Again this was found to be an unsafe approximation.

From these experiments we can draw the following conclusions:

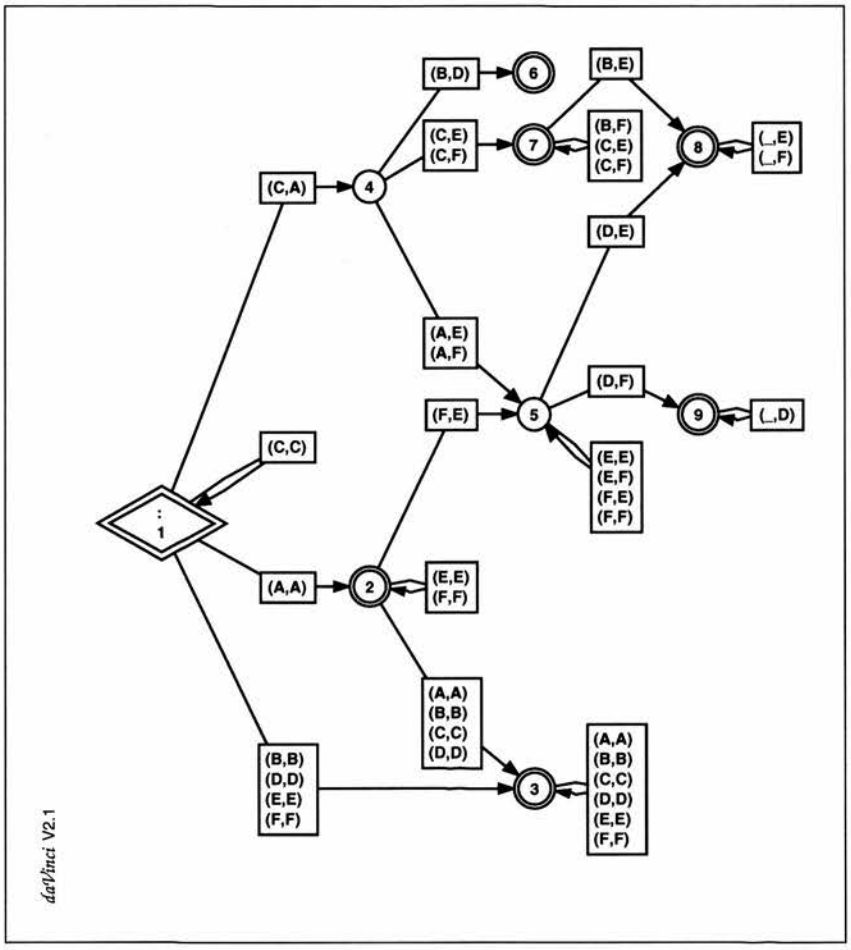


Figure 4.25: The (incorrect) computed transitive closure of the conflict 2FSA in Figure 4.24

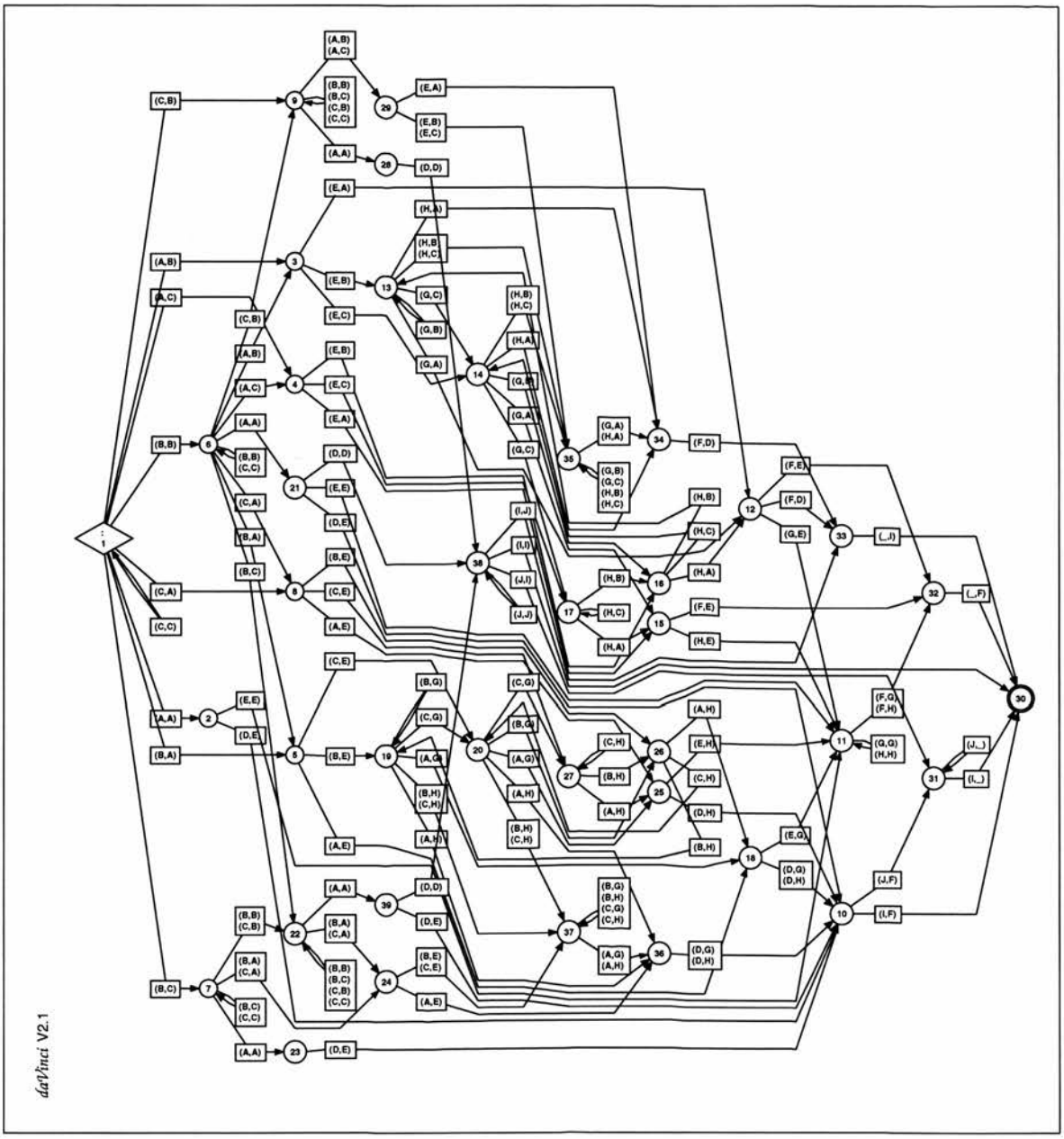


Figure 4.26: The conflict 2FSA for the program in Figure 4.18.

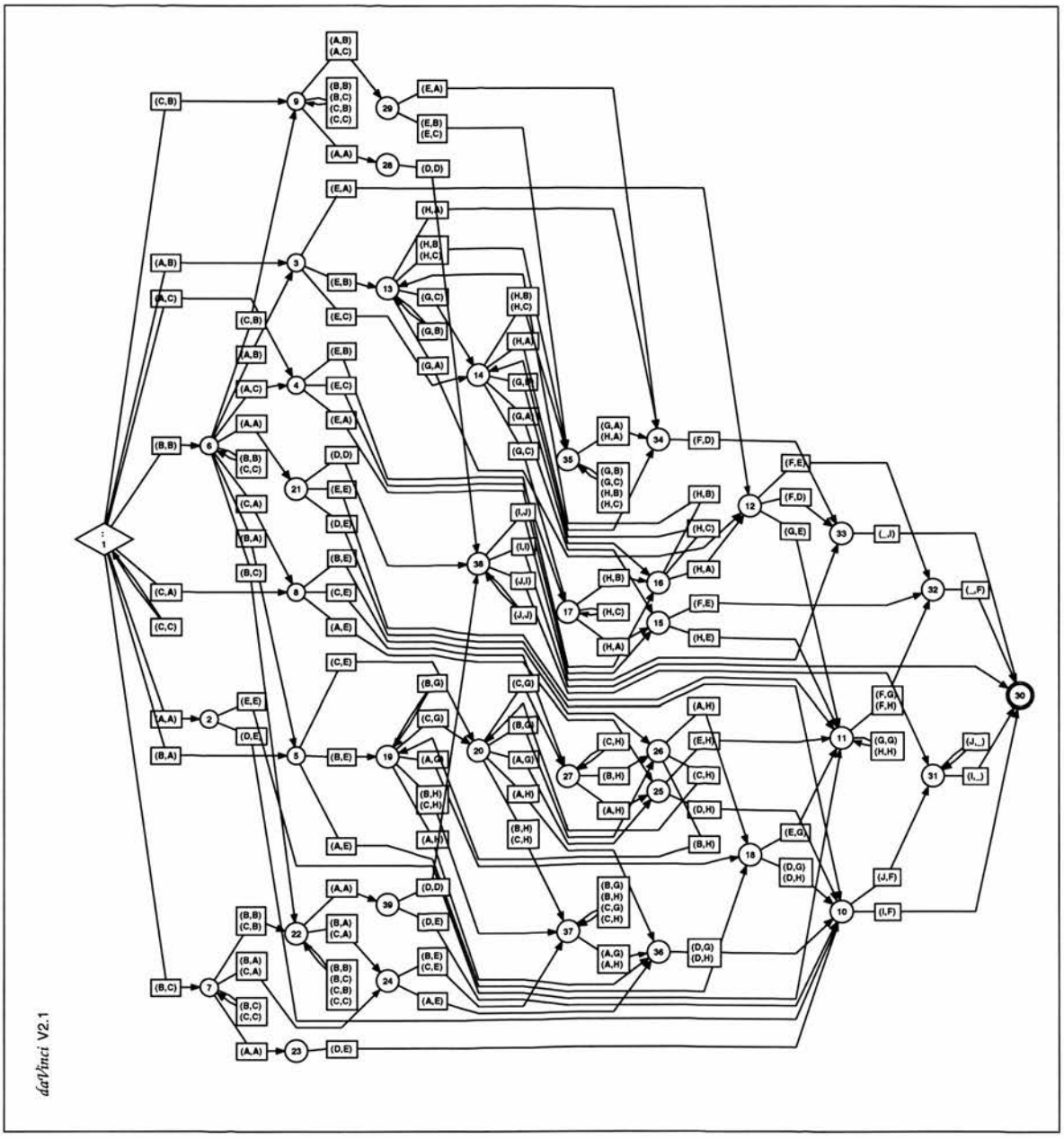


Figure 4.27: The unsafely approximated closure of the conflict 2FSA in Figure 4.26

- The transitive closure heuristic does not seem to be applicable to 2FSAs much larger than the ones used to specify pointer linkages. It is possible that these could be enhanced to give correct safe approximation for these larger 2FSAs.

- The larger example exposes some of the scaling issues with this dataflow approach. The second program is approximately twice the size in lines of code, has a conflict 2FSA of around four times the complexity, and its closure takes around 80 times the computation to compute. These methods do not scale well to larger programs.

These two points are suggestive that this dataflow approach is a unprofitable direction, and we pursue it no further in this thesis.

4.5.2 Link Recursion Conclusions

We have looked at various methods for storing the relation between control words and induction variables for programs that contain recursion in link directions. The LSA formalism is exact, but requires heuristic methods to perform the additional manipulations our analysis requires. We chose to keep the exact manipulations that the 2FSA description allow and produce approximate 2FSA relations for the induction variables. We have extended the approach to any nest of recursive calls, allowing approximations to the access patterns of programs that recurse in any direction of the structure to be built.

4.6 Data Dependency Information

We now consider what information about the data dependency patterns in the code can be extracted from the information we have built so far. Firstly, we define the set of 2FSAs that will summarise the information we are interested in.

Definition 9 *For any control word X we define the sets of pathnames $R(X)$, $W(X)$ and $A(X)$ as:*

- *$R(X)$ is the set read from by the statement represented by X.*

- *$W(X)$ is the set written to by X.*

- *$A(X)$ is the set accessed (read or written) by X. $A(X) = R(X) \cup W(X)$.*

These relations are actually stored by the following 2FSAs.

- *The* Read *2FSA,* R, *that accepts the pair (X,y) if $y \in R(X)$.*

- *The* Write *2FSA,* W, *that accepts (X,y) if $y \in W(X)$.*

4.7 Access 2FSAs

The previous sections have shown how to build up 2FSA descriptions of each of the induction parameters in the program. We now wish to use them to build 2FSAs that store the information of the read and write accesses to the structure that each statement makes. We refer to this collectively as the *Access information.*

For each statement X that reads a node from the structure, by accessing '`p->w`', we can append the statement symbol, X, and the 2FSA for the word, w, for that statement to F_p. Thus if F_p accepts (C,y), this new 2FSA will accept $(C.X, y \rightarrow w)$. It is formed in the following manner:

Construction of $\mathsf{F}_{\mathsf{stat}}$

- Given a 2FSA, $\mathsf{F}_{\mathsf{ind}}$, that accepts (C,y), create a new 2FSA, $\mathsf{F}_{\mathsf{stat}}$, that accepts $(C.X, y \rightarrow w)$, where X is a statement label and w is a read or write word, with associated 2FSA L_{w}.
- For control symbol, X, define $\mathsf{AddFirst}_{\mathsf{X}}$ that accepts $(x.X, x)$ for all control words x. Its construction is straightforward.
- $\mathsf{F}_{\mathsf{stat}}$ is then given by

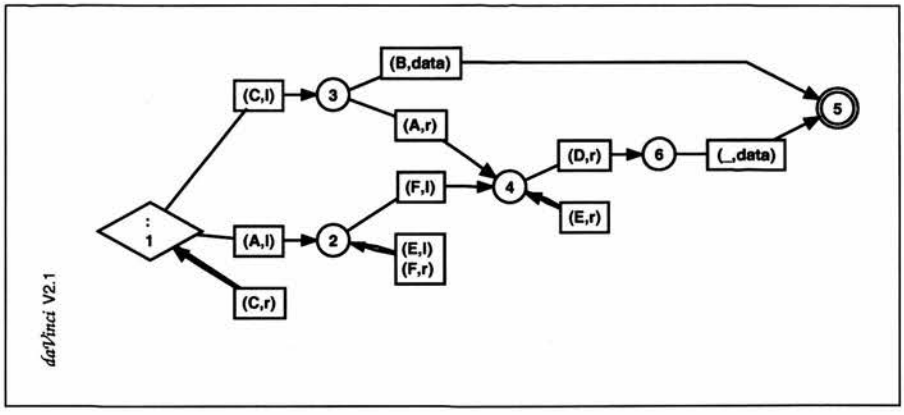
$$\mathsf{F}_{\mathsf{stat}} = \mathsf{AddFirst}_{\mathsf{X}}.\mathsf{F}_{\mathsf{ind}}.\mathsf{L}_{\mathsf{w}}$$

The conjunction (ANDing together) of this set of 2FSAs over all statements X in the program produces the *Read 2FSA*, R, that maps from any control word to all the nodes of the structure that can be read by that statement. Similarly, we can produce a 2FSA that maps from control words to nodes which are written - the *write 2FSA* denoted by W.

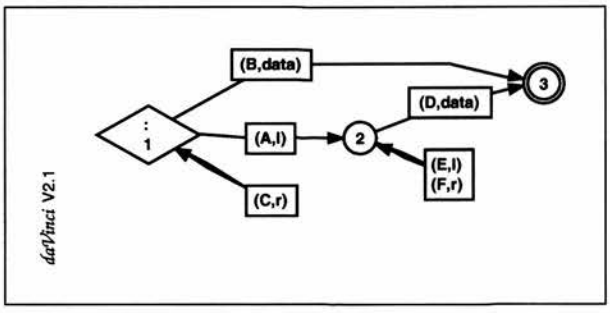
These write and read 2FSAs for our running example are shown in Figure 4.28. The write 2FSA is simple, since in the program we only every write to the '`data`' field of the current function parameter '`t->data`' in statement B and '`w->data`' in statement D. Thus the write 2FSA is very similar to the valid control word 1FSA in Figure 4.4, instead converting each control word into the value of the current function parameter and appending the '`data`' field to each pathname. It is easy to see how such a 2FSA could be constructed by hand.

The read 2FSA is more complex, this arises solely from the fact that the read access in statement B occurs in a link direction (p), and in D the access occurs in a compound direction ($p- > l- > n$). Unlike the write 2FSA, without performing the manipulations mentioned above it would be difficult to directly construct this 2FSA by hand from the original program.

We are now in a position to describe all the nodes which have been read from, and written to, by any statement. Two instances *conflict* if they both access the same memory location with



(a) The Read 2FSA



(b) The Write 2FSA

Figure 4.28: Read and Write 2FSAs for the example code

at least one being a write. The possible types of conflict are therefore read after write (RAW), write after read (WAR) and write after write (WAW).

By combining these read and write 2FSAs we can create 2FSAs that link statements for each of the possible types of dependency (RAW, WAR, WAW). The conjunction of these 2FSAs forms the *conflict* 2FSA, which denotes whether a pair of statements are in conflict with respect to dependencies. The *conflict 2FSA*, C, is defined in the following way:

Definition 10 *The* conflict 2FSA, C, *accepts pairs of control words that represent statements that have a dependency between them: they access the same memory locations and at least one is a write.*

$$(x,y) \in \mathsf{C} \Rightarrow \exists d \in A(x) \cap A(y) \quad \textit{with either} \quad d \in W(x) \quad \textit{or} \quad d \in W(y)$$

The conflict 2FSA is computed from the 2FSAs for the nodes read (R) and written to (W). Firstly each of the RAW, WAR and WAW are computed from the R and W 2FSAs and their inverses. Then C itself is formed from the conjunction of these dependency 2FSAs.

$$\mathsf{RAW} = \mathsf{R}.(\mathsf{W})^{-1} \tag{4.1}$$

$$\mathsf{WAR} = \mathsf{W}.(\mathsf{R})^{-1} \tag{4.2}$$

$$\mathsf{WAW} = \mathsf{W}.(\mathsf{W})^{-1} \tag{4.3}$$

$$\mathsf{C} = \mathsf{RAW} \cup \mathsf{WAR} \cup \mathsf{WAW} \tag{4.4}$$

These 2FSAs so far do not consider whether the instances that conflict actually occur earlier in the sequential execution of the program. So C needs to be stripped of all these bogus dependencies. This is handled by applying a *causal* 2FSA (via the AND operation) that takes into account the causality of the dependency. Note that this causal 2FSA is the inverse of the '$\geq$' 2FSA we defined earlier in Section 3.2.2. If two instances are in conflict, then we use the term *source* to refer to the earlier one.

Definition 11 *We define the* causal *2FSA,* $\mathsf{F}_{\mathsf{causal}}$*, which accepts control words* X,Y*, only if* Y *occurs before* X *in the sequential execution of the code.*

The construction of $\mathsf{F}_{\mathsf{causal}}$ is done as follows. It consists of two states, one initial and one accepting. It has transitions from initial to accepting for all symbol pairs (X,Y) where Y is strictly less than (occurs before in the program) X. Transitions for (X,X) for all symbols X lead back to the initial state. All transitions for every pair of symbols are included from the accepting state back to itself. This 2FSA is not illustrated since it depends on the particular set of control words and for a program of moderate size is too complex to make sense of visually due to the large numbers of transitions. For a program of length n there are $n(n-1)/2$ transitions from

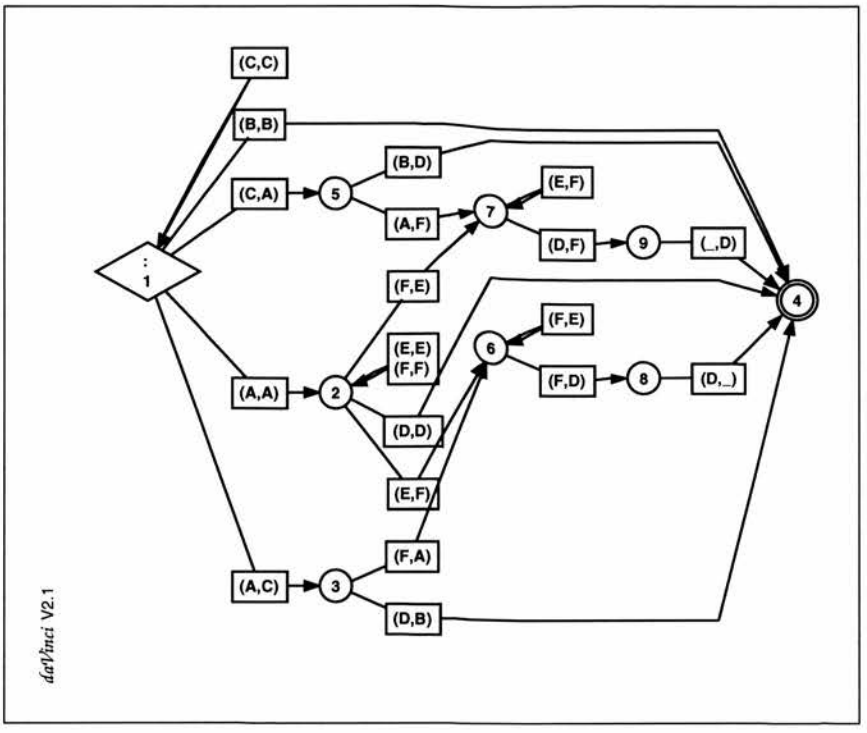


Figure 4.29: Raw Conflict 2FSA for the example code fragment

initial to accepting, n self transitions from the initial state and n^2 self transitions on the accept state.

The 2FSA in Figure 4.29 is the raw conflict 2FSA for the running example. In Figure 4.30 it has been ANDed with the $\mathsf{F}_{\mathsf{causal}}$ 2FSA, which has removed any false sources that occur after the statement. For example the 2FSA in Figure 4.29 accepts the 'self dependency' between statements B and B which is removed in Figure 4.30. Also if we look at the (E,F) transition from state 2 to state 6 in the 2FSA in Figure 4.29, this is removed in Figure 4.30 one since these control words must begin with (A,A) given by the transition from state 1 to state 2. This makes all subsequent control words accepted on this part of the 2FSA redundant since they are bogus dependencies. This happens also with the entire tree rooted with the (A,C) transition from state 1 to state 3.

In summary, 2FSAs (R and W) are built that store the read and write accesses of the program, using the program code and structure description. These may involve some approximation due to recursion in non-generator directions, and flexibility in the original link descriptions. These are *safe* in that accesses are not missed by the approximation process and they are supersets of all possible accesses by a program with a data structure that obeys the 2FSA description.

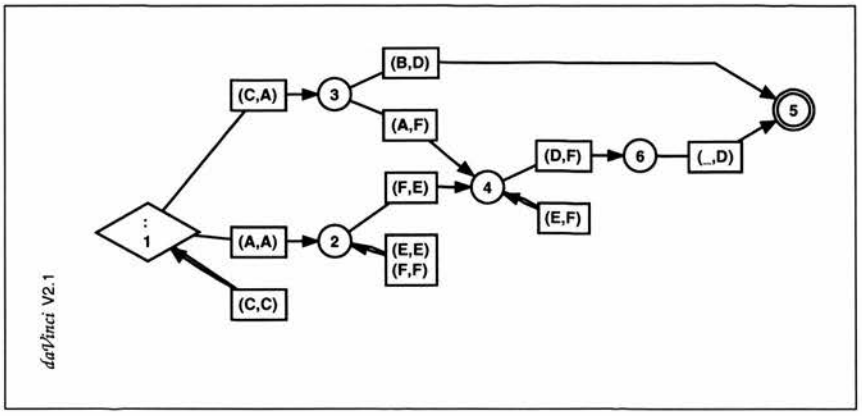


Figure 4.30: Simplified version of the information in Figure 4.29 with the conflict information trimmed dependent on causal information.

We then use these to produce 2FSAs for each type of dependency RAW, WAR and WAW. The final computation of C is given by:

$$C = (RAW \cup WAR \cup WAW) \wedge F_{causal}$$

4.8 Complexity

Any sufficiently sophisticated compile-time analysis is restricted by the fact that in practice these techniques must scale to large programs while still keeping compilation times reasonably short. Although we only ever apply our techniques to short programs it is worthwhile looking at the complexity of our analysis, and how we could scale our implementation to deal with larger programs. Since finite state machines are the fundamental object being manipulated using this method, the complexity of our analysis relate to the complexity of the underlying 2FSA operations.

Our implementation of the alphabet for the finite state machines keeps the generating direction alphabet and control word alphabet separate. If n_g is the number of generating directions, n_l the number of lines in the code, then the size of the alphabet of each 2FSA is $(\mathrm{Max}(n_g, n_l) + 1)^2 - 1$. For a reasonably-sized program, the number of lines in the code will dominate, making this size $O(n_l^2)$. For the size of the programs we deal with this is manageable.

For much larger programs we would use the following technique to keep the alphabet size to a practical level. In the 1FSA that accepts a valid control word there are only transitions from each state for the symbols in one of the functions. This makes this 1FSA *sparse*; only a small fraction of transitions are non-zero. This will also apply to any 2FSA containing access information that we derive from it. We can take advantage of this to reduce the alphabet size by mapping all control words to a smaller alphabet of size $\mathrm{Max}(n_{F_i})$, where n_{F_i} is the number

of lines in function i and the maximum is taken over all functions. Unless the number of program lines is very unequally shared amongst the functions, this will be a much smaller alphabet. Also, for well structured code, the maximum length of functions should not increase as program sizes get larger.

The memory requirements for such an FSA with n_s states will be $O(n_s.n_l^2)$ and storing the information sparsely will decrease this. However, the number of states can potentially increase dramatically: given two FSAs with n and m states respectively, the FSA built by the operations AND and OR can have up to $n \times m$ states. When combining two FSAs to form a composite, the combined FSA can have up to $2^{n \times m}$ states. This level of growth can produce very large FSAs for quite modest sizes of m and n. In turn this has the effect of requiring large amounts of memory and computation time if these FSAs are further manipulated.

In practice, though, the number of states often remains manageable, due to the sparseness of the 2FSAs that describe the link directions. Thus the actual size of 2FSA descriptions is nearly always much smaller than the worst case estimates we give above. As an example using these estimates, the combined word, $p.l$, required for analysis of the binary tree could be expected to require 16 states, in fact it requires only 3 states. The bulk of the computation time of our implementation on our example set was nearly always spent in performing the closure operations. This time was the order of around 5 seconds per program. These computations tend to be only required once per program and minor debugging changes to the code would not in general require a different closure computation to be carried out. These could therefore be performed once and cached for later analyses.

4.9 Conclusions

This chapter has shown the analysis that is required to extract useful dependency information from programs that use the 2FSA data structures. The information is presented and manipulated also in the 2FSA form. Often the information will be approximate, since heuristics must be used in the situation where 2FSAs are insufficiently powerful to hold the full accurate representation. Some additional formal language constructs that allow complete accuracy were studied. These were rejected as an approach since further manipulations with them were more complex and would also require additional approximation.

In this chapter we have begun to look at how the access information can be manipulated into a more useful form. The conflict information C that was computed may be a very large 2FSA with many hundreds of states, even if the original code is only tens of lines. We want to be able to extract useful summary information from it. In the next chapter we will see how one can use this extracted information to annotate programs to permit parallelisation.

Chapter 5

Extracting Parallelism

The information gleaned from the dependency analysis is next used to assist in the parallelisation of the programs. In this chapter we first consider the forms of parallelism that we might find in such programs. Then we outline an approach where the programmer can split a program into a number of sub-threads.

Our analysis can then use this partitioning and dependency information to provide the synchronisation points that are required to safely run these threads in parallel. We illustrate much of this by assuming that there are only two threads of computation. We finish off by showing how our approach can be extended to any number of concurrent threads.

5.1 Types of parallelism

We need to consider the different kinds of parallelism that the dependency analysis information may make available, and what assistance is required to extract it.

Function Call Parallelism Two or more function calls are distributed to separate processors. The calling of such a concurrent thread is termed a *spawn* of the function by the parent thread. Many functions can be spawned at once. Once each function completes execution it signals back to the parent thread. A *synch* function is commonly used by the parent directly after the spawned function calls, this halts it until it has received signals back from all its spawned functions. This parallelism approach requires the analyser to spot when the data read and written by each call does not clash. We can produce 2FSA descriptions of the data accessed by each function call. Note that this information may be inaccurate, since there may be conditional data accesses that depend on run-time information or approximations that are part of the analysis process. However, it will have to be *conservative*, in the sense that the actual set of data will be a subset of that computed.

Fine Grained Parallelism Individual statements are scattered across processors. This requires knowledge about the dataflow dependencies between individual statements. We can produce this information for the programs we consider, but any approximations we apply will detract from the accuracy. It we cannot assume static control, then this will also introduce approximation and any improvement will require techniques like those covered in *fuzzy* approach given in [28]. The transformations of the program for this approach would also be quite complex, since individual statements need to be scheduled by the program.

Function Call Parallelism is the type of parallelism that is being extracted in [58], using the EARTH-C language, and in Cilk, which we discuss later in Section 5.1.1.

Combinator Graph reduction [118] is an example of fine-grained parallelism. It expresses a functional program as a graph, shared sub-expressions in the program are represented as shared subtrees. Combinators operate on their subtrees and rewrite the tree as it is evaluated. For example, an addition combinator may hold pointers to two constant sub-expression nodes, when evaluated it is replaced by a constant node representing the sum of these two. Each sub graph may be evaluated in parallel, exposing large amounts of parallelism. Some of the issues for efficient implementation of such an approach is explored for shared memory architectures in [119].

5.1.1 The Cilk language approach

The 'function call' approach to parallelism has been explored using the language called 'Cilk' [120]. Cilk comes with a tool called the 'Nondeterminator' which, given a Cilk program and the data on which it operates, tests for possible data conflicts in spawned threads. The answers received are approximate, for the given data set.

One use of our dependency analysis could be to decide where, if anywhere, we can insert `spawn` and `sync` commands, given a program in our reduced language. Using the conflict 2FSA (defined in Section 4.7) we can determine when any spawned function calls contain accesses to the structure that would conflict. Ideally, we would want to find completely independent function calls with no points of conflict that could be spawned off on separate processors. Unfortunately, for code that traverses the sort of structures we look at here, there is frequently dependency between function calls within the same function body. This occurs because the link directions cut across what would otherwise be independent sections of the tree. There are two possible ways of tackling the implementation of this parallelism:

- Unroll the recursive functions to a particular depth and look for independent calls from more than one function. This can be expected to find some independent sections, but has the potential of creating a large number of calls that have some dependency.

- Use a system of synchronisations to run two conflicting functions concurrently and ensure the operations occur in correct sequential order.

We will attempt the second approach here, since it is the more general and is simpler to implement. In fact, if using the first approach would uncover parallelism in a particular program, the second approach will always be able to find it as well.

5.2 Parallelisation of 2FSA codes using threads

Our approach is to use a threads library to annotate the original code to produce a version that utilises threads to provide the parallelism. We use the Cilk threads library to illustrate this, it handles parallelism using the following constructs:

- `spawn` *func* to spawn a thread of function *func*. The spawned thread is known as a *child* thread, of which there can be more than one.
- `sync` to wait for all currently spawned child threads to terminate. The calling thread blocks until all the children in the block have completed.
- A set of thread locks and functions to lock and unlock them: `thread_lock()` and `thread_unlock()`.

We should explain the use of these locks and synchronisations more fully. [121] deals with these topics (pages 590-600) with more information on hardware implementation details. A threaded program can set up a number of locks, each of which can be *owned* by at most one thread. A thread can attempt to own a lock `l` by calling `lock(l)`. The lock is tested, if lock it is already owned, then this thread will block, otherwise it will claim the lock and continue. A thread can unlock any lock it owns. These are termed *spin locks*, the thread is assumed to be "spinning", constantly checking to see if the lock has been freed. We must think of the "test and claim" operation as atomic, in other words no other process can interrupt it between when it finds the lock to be free and claiming the lock. If another process could call `lock(l)` between these points, it too would find the lock free and attempt to claim it as well. Atomic locks require some hardware support, typically an atomic exchange operation that swaps the values of a register and a memory location in one step. The `sync` operation (often called a *barrier synchronisation*) can be implemented using a pair of spin locks, one to protect a counter that keeps track of how many threads have reached that point and another that all threads spin on until the last one has arrived.

These primitives allow threads to exclude others from entering certain critical sections of code until they have completed particular operations. We will use them to ensure that certain updates of the data structures will occur in the correct sequential order.

The parallelisation approach described in this chapter will generalise to any threads library that supports the same features. Threads libraries that support this include POSIX threads [122], Cilk threads and the MAPS instruction set we describe later in Section 6.2.1. The MAPS instruction set does not supply these primitives directly, but their functionality can be implemented using a combination of MAPS instructions, or directly via shared memory as we choose to do later.

5.2.1 Partitioning the computation

We want to be able to split the full set of statements that may be executed into separate disjoint threads. We can then analyse the data dependencies between pairs of threads. The aim for the point of view of the programmer is to split the computation in such a way as to minimise the conflict. A range of different partitioning approaches can be experimented with by the programmer and one that has least scope for conflict can be chosen.

Automating this process is quite feasible, but for this thesis we assume a worthwhile partition has been chosen, and our goal is simply to ensure that the chosen scheme executes correctly. This is achieved by deriving the appropriate locking scheme.

A partition $\mathcal{P}$ is defined as a regular subset of the set of valid control words $V \subset C^*$. This is denoted in a partition scheme that accompanies the original program. For flexibility the partition is described using a 1FSA *slice* F that accepts a subset in the alphabet of the control symbols.

To simplify this process the partition is combined with the proper set of valid control words. The actual partition used is the $\mathcal{P}$ containing all control words in $F \wedge V$, making it must be a valid control word.

The set of control words that fall within a particular function call is the set of all valid control words that begin with a particular sequence of symbols relating to the nest of function calls. These are easy for the programmer to specify, since they are only required to supply a 1FSA that accepts all control words beginning with a particular string of call symbols and the valid partition can be computed correctly.

The partition description supplied by the programmer consists of a list of 1FSA descriptions, each one representing a partition of the control words, as shown in Figure 5.1. The reserved word '`contslice`' indicates that it is a control word slice. Here we have two partitions specified, the first that contains all control words beginning with B and the second all beginning with A. Thus the two partitions correspond to the two function calls to the '`propagate`' function occurring in '`main`'. Note that for simplicity these are expressed as 2FSAs that accept all strings beginning with A and B, in truth some of these will not be valid control words. The full

```
main (Tree *node) {
  if (node->d1!=NULL) {
    propagate(node->d1);        //      A:
    propagate(node->d4);        //      B:
  }
}
propagate(Tree *p) {
  if (p!=NULL) {
    p->data = p->l->data + p->r->data
      + p->d->data;     //      C:
    propagate(p->d4);   //      D:
    propagate(p->d2);   //      E:
    propagate(p->d1);   //      F:
    propagate(p->d3);   //      G:
    sweepl(p->l);       //      H:
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
    x->l->data = x->l->data + x->data;  //      I:
    sweepl(x->l);       //      J:
  }
}
{contslice
   States 2;
Start 1; Accept 2;
Transitions[
            [B->2;]
            [A->2;B->2;C->2;D->2;E->2;F->2;G->2;H->2;I->2;J->2;
]
]
  }
{contslice
   States 2;
Start 1; Accept 2;
Transitions[
            [A->2;]
            [A->2;B->2;C->2;D->2;E->2;F->2;G->2;H->2;I->2;J->2;

]
]
  }
```

Figure 5.1: The text of the program with the description of the partitions

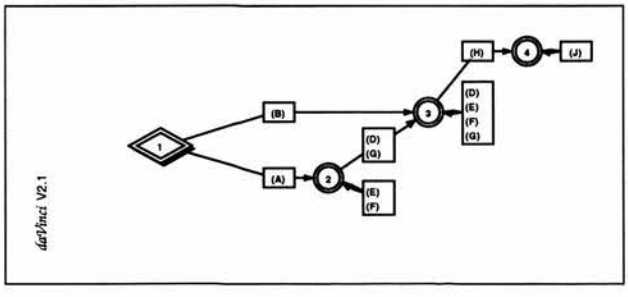


Figure 5.2: The 1FSA that accepts the full set of valid control words for the program in Figure 5.1

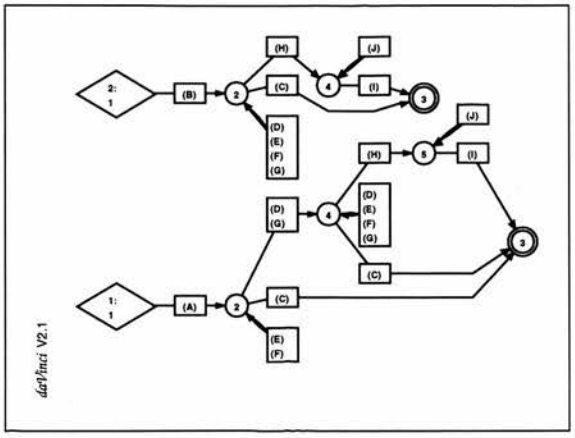


Figure 5.3: The two partitions defined in Figure 5.1

set of valid control words is given in the 2FSA in lower half of Figure 5.2. Note for example that only control words that end in either a C or I are valid. When each partition description is ANDed with this valid 2FSA, we obtain the true partitions given in Figure 5.3. It can be seen that these two partitions cut the whole set of valid control words into two disjoint sets.

Determining a set of thread partitions that are going to parallelise the code efficiently is a difficult task. The dependency information we extract later shows the full dependencies between each thread. This can be used to determine how much synchronisation can be expected, the fewer dependencies there are, the less synchronisation overhead we can expect and the better the partitioning is. We return to the issue of deriving good partitions when we conclude in Section 7.1.3.4.

Deriving complex partitions in the form of 2FSAs could also be a burden on programmers. The system mentioned above permits partitions that consist of whole function calls to be described without explicit knowledge of the complete valid control word 2FSA. We do not consider more complex partitioning than this in this thesis, so all of our example partitions will be this straightforward to specify.

We follow the idea that each partition is running in a parallel thread. If a partition is defined as all the statements in a particular function call then we can run that partition by spawning the function in parallel. If a function makes two consecutive function calls, labelled A and B then we can define these as partitions $\mathcal{P}_A$ and $\mathcal{P}_B$, by taking the subset of control words that begin either with an A or a B.

We assume that all the statements in the first partition will execute earlier than those in the second partition when run in the correct sequential manner. We use the notation $\mathcal{P}_A < \mathcal{P}_B$ to denote this. The order '$<$' on control words referred to here is the order that statements would occur in if the execution occurred sequentially. This is the same as the lexicographical order given when statements are ordered as they appear textually in program code.

This assumption ignores "interleaved" partitions, where $\mathcal{P}_A$ contains some statements that would occur naturally after the ones in $\mathcal{P}_B$. We use this restriction here since it greatly simplifies the locking mechanism we consider later as it removes some potential for deadlock. However, this restriction greatly reduces the scope for parallelism of these programs. It completely removes all potential parallelism between statements in each function and forces them to execute sequentially. In the extreme case if there is a conflict between the last statement of partition A and the first in partition B then the two functions will be forced to execute in sequence, exactly as in the sequential version. The further forward the first conflicting statement in B is and the further back in A the statement with which it conflicts is positioned, the more parallelism will be exploited.

We must bear this in mind later in Section 6.3.4, where we use raw traces to predict maximum amount of parallelism. Often these estimate of large parallelism will require complex partitions that violate this assumption.

Once a set of partitions is chosen we must answer the question: under what circumstances can we spawn these partitions as separate threads? If the conflict 2FSA does not accept any pair of control words, with one in $\mathcal{P}_B$, and another in $\mathcal{P}_A$, then there can be no dependency between the partitions, and we can safely spawn them as independent threads, with no extra communication required between them.

If there are dependencies we need to add some blocking between the threads that ensure the updates occur in the correct order. How to derive this information from the raw conflict descriptions we derived in the last chapter is what we turn to next.

5.3 Building 'Waits for' information

We begin with the raw conflict 2FSA C, which as derived earlier in Section 4.7 is possibly approximate, but safe. We will assume for now that we are only handling one pair of partitions, we will attempt to generalise this approach later in this chapter, in Section 5.6.

We can prune C so that it only contains dependencies between our partitions of interest: $\mathcal{P}_A$ and $\mathcal{P}_B$. We call this the *reduced* conflict 2FSA: C_{BA}. We then produce what we call the *waits-for* information. This maps each control word in $\mathcal{P}_B$ to the last one in $\mathcal{P}_A$ that conflicts with it. Thus each statement is mapped to the one in the previous thread that it must wait for in order for correct execution to proceed. We can then get the threads to synchronise by forcing each statement in $\mathcal{P}_B$ to wait for the appropriate statement, given by the waits-for 2FSA, that executes in thread $\mathcal{P}_A$.

The construction of a waits-for 2FSA from a conflict 2FSA C aims to remove for each statement x all but the latest dependent statement y that occurs before x. Thus we can say that x can be safely executed once we have waited for y to execute.

We now describe how to produce a waits-for 2FSA for a given conflict 2FSA C. In the notation of 2FSA operations, we initially take the following manipulation as our description:

Definition 12 *The Waits-for 2FSA* $\mathsf{W_C}$ *of* C *is given by*

$$\mathsf{W_C} = \mathsf{C} - (\mathsf{C}.\geq)$$

Any $(x,y) \in \mathsf{W_C}$ will have the required waits-for property, since if $(x,y) \in \mathsf{C}$ and $(x,y) \notin \mathsf{C}.\geq$, it implies that there is no z with $(x,z) \in \mathsf{C}$ and $z > y$.

However, the waits-for 2FSA given above may miss dependencies in the following way: there may be an x such that there is a $(x,y) \in \mathsf{C}$, but there is no z such that $(x,z) \in \mathsf{W_C}$. This means that some statements in the second thread will have dependencies in the first, but will have been removed from the waits-for information. The next section aims to deal with exactly this issue.

5.3.1 Missed dependencies

We now take a closer look at the issue of missed dependencies in the waits-for information. These can occur in the following situation.

Suppose there is an infinite chain of control words $y_1, y_2, \ldots$, with $y_i \leq y_{i+1}$ for all i. Now suppose there is a statement x that occurs later than all of them and is dependent on them all, $(x,y_n) \in \mathsf{C}$ for all n. Also assume that if there are any other statements x depends on, then these occur earlier than y_1.

Then if $(x,z) \in \mathsf{W_C}$, implying $(x,z) \in \mathsf{C}$ and $(x,z) \notin \mathsf{C}.\geq$. So there exists no u such that $(x,u) \in \mathsf{C}$ and $u \geq z$.

Either:

- $z = v$ such that $(x,v) \in \mathsf{C}$ and $v < y_1$. But we have a contradiction since $(x,y_1) \in \mathsf{C}$ with $y_1 > v$.
- $z = y_i$. But $(x,y_{i+1}) \in \mathsf{C}$ with $y_{i+1} > y_i$, again a contradiction.

The contradiction proves that there is no such z in $\mathsf{W_C}$, and all the dependencies of x have been pruned away.

5.3.2 Pruning infinite dependencies

We solve this problem by applying a transformation called *Prune-Infinite* to C before we produce the waits-for information. This operation must have the following property. If x has an infinite chain of dependencies y_i, as described previously, then a *false dependency* (x,z) to C must be added, where $z > y_i$ for all i. We need to find such a z. The false dependency will often

- Locate a state s with transition of the form $s \xrightarrow[(-,A)]{} t$, for any A that occurs within a loop of states L.
- Make all states in L accepting.
- Remove all transitions of the form $l \xrightarrow[(-,B)]{} l'$ between states l and l' in L.

Figure 5.4: Removing infinite sets of dependencies

be to a function call point in the program rather than a normal write statement. In terms of the original program, rather than a statement having dependencies to a infinite chain of statements inside an earlier function call, the false dependency will be targeted just to the function call statement.

We call this augmented conflict 2FSA C'. We can ensure that no dependencies have been missed by checking that $\text{First}(\mathsf{W}_{C'}) = \text{First}(\mathsf{C})$. More simply, where we are considering dependencies between a pair of threads we only need to check that the dependencies from the second thread are conserved. If the threads are described by partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ then we need only check:

$$\text{First}(\mathsf{W}_{C'}) - \mathcal{P}_1 = \text{First}(\mathsf{C}) - \mathcal{P}_1$$

Now we must look at how to construct C′ from C. Firstly consider the circumstances under which an infinite chain of dependencies can occur. If we have a 2FSA, C, mapping control words to control words then a word x can map to an infinite chain of y_i if, and only if, there is a transition of the form $s \xrightarrow[(-,A)]{} t$ where A is a particular control symbol, that occurs within a loop of linked states of the 2FSA. In only this situation can an xx of fixed length be mapped to an infinite set of y_i of ever increasing lengths.

The "Prune-Infinite" algorithm locates these loops where an infinite control word can be identified, makes all states in the loop accepting, and removes any $s \xrightarrow[(-,A)]{} t$ transitions within it. This will have the effect of removing the infinite chain and adding the extra dependencies to any remaining states within the loop. The full algorithm is given in Figure 5.4, which removes all of the infinite set of dependencies, and adds additional ones that corresponds to the function call(s) that spawn the infinite set. Not that since we only ever remove $(-,A)$ transitions, this transformation does not affect the value of First(C).

In practice, these loops will often be very simple, just transitions from an accepting state s to itself of the form: $s \xrightarrow[(-,A)]{} s$. In this situation the algorithm just removes that transition.

As an example if $A.C.D$ is dependent on the set $A.B.B^*.E$, all these dependencies will be replaced by one from $A.C.D$ to $A.B$. There is a subtlety here in that $A.B$ is actually less than all of the control words in this particular infinite chain set. This is a more general concern, since

with this approach the added dependency will still appear to occur earlier. This anomaly can be resolved in the following way. The standard lexicographical order only really makes sense when comparing the order of the assignment statements, and not function call statements. The complication arises in the case of call statements because the call begins executing before all of its child statements, and finishes only after all of them have completed.

Luckily this does not impact what we are doing here. The aim is to find statements that finish *after* the ones with the dependency. Since the completion of the function call statement implies the completion of all its child statements, it is still the appropriate place to insert any lock-freeing code.

So the full set of manipulations to derive the waits for information from the conflict information C is as follows:

- Produce the reduced conflict information between the two threads:

$$\mathsf{C}_{\mathsf{BA}} = \mathsf{C} \wedge \mathrm{FirstAll}(\mathcal{P}_B) \wedge \mathrm{SecondAll}(\mathcal{P}_A)$$

- Prune to remove infinite dependencies:

$$\mathsf{C}'_{\mathsf{BA}} = \mathrm{PruneInfinite}(\mathsf{C}_{\mathsf{BA}})$$

- Form the waits-for information:

$$\mathsf{W}_{\mathsf{C}} = \mathsf{C}'_{\mathsf{BA}} - (\mathsf{C}'_{\mathsf{BA}} \cdot \geq)$$

Most of these operations are quick to compute on even large 2FSAs, since they are linear in the number of transitions. The main bottleneck is in the waits-for creation, since it involves a composition with the 2FSA that maps all control words onto all lexicographically later words. This 2FSA is large, it has a number of transitions proportional to the square of the number of program statements. For the short ($15-20$ line) programs we consider in this thesis this is no problem (less than half second in analysis times), and this should allow programs of greater length (hundreds of lines) to be analysed in practical timescales.

5.3.2.1 Example Manipulations

As an illustration of these manipulations, we show them being applied to an example. Figure 5.5 is a 2FSA containing raw conflict information. Here the loop of transitions $(\_, G)$, (J) from the state labelled 14 back to itself causes infinite chains of dependencies. For example there is an infinite set of dependencies from the one statement $B.C.J$ to the set of statements

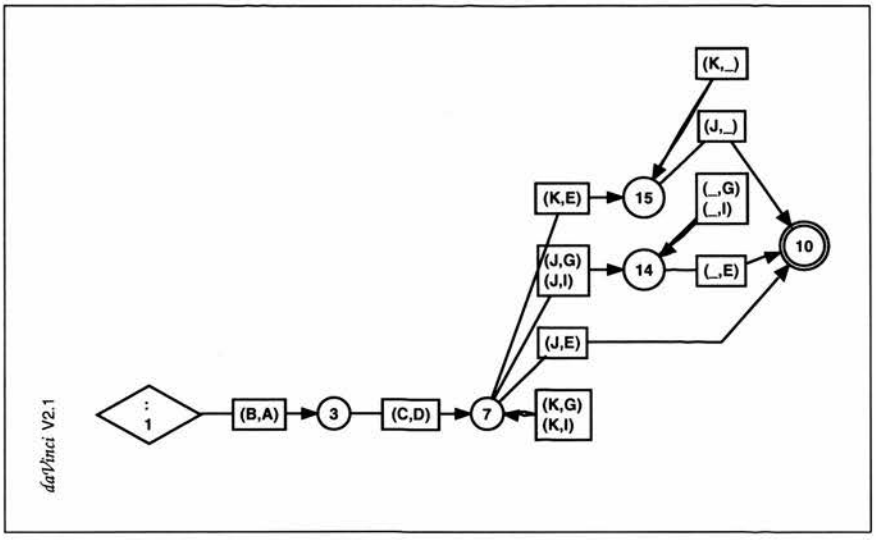


Figure 5.5: Raw conflict information

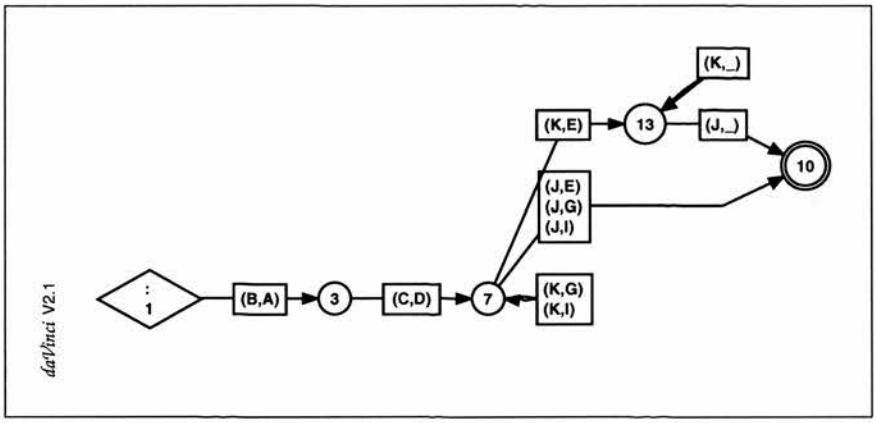


Figure 5.6: Confict information with the prune-infinite phase applied

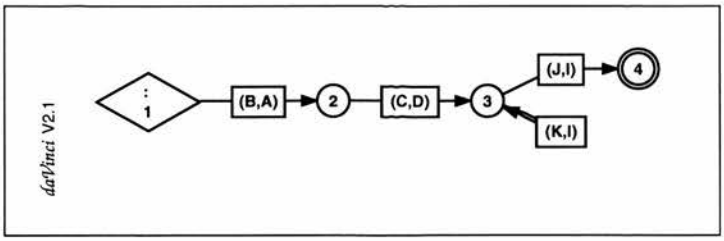


Figure 5.7: The waits-for phase applied to the pruned confict 2FSA.

$A.D.G^+.E$. The 2FSA shown in Figure 5.6 has had these infinite chains of dependencies pruned by the procedure outlined above. This has meant removal of state 14[1], and directly linking all its inputs to the accepting state. Thus, amongst others, the set of dependencies between $B.C.J$ and $A.D.G^+.E$ have been replaced by a single dependency between $B.C.J$ and $A.D.G$.

Next the waits-for manipulation is applied, giving the 2FSA in Figure 5.7. Looking back to Figure 5.6, the set of transitions (J,E) and (J,G) from state 7 to state 10 are removed since they occur before (J,I), and the (K,E) transition between states 7 and 13 is removed since it occurs before (K,I), as does its neighboring (K,G) transition. For example, the following set of dependencies:

- $B.C.J \rightarrow A.D.E$
- $B.C.J \rightarrow A.D.G$
- $B.C.J \rightarrow A.D.I$

is pruned down to just the last one: $B.C.J \rightarrow A.D.I$.

After cleaning up and a certain amount of state relabelling, the 2FSA in Figure 5.7 holds the final dependency information.

5.4 Locking the Threads

Once we have computed the waits-for information, we need to use it to force the necessary parts of the computation to occur in the correct sequential order. Each statement s in $\mathcal{P}_B$ that conflicts can be mapped to the one in $\mathcal{P}_A$ that it must wait for before it can execute. If we have a thread lock T_s for each control word s in $\mathcal{P}_B$ that needs to wait for statement r in $\mathcal{P}_A$, we can synchronise the threads in the following manner:

1. Insert an attempt to gain lock T_s before statement s.
2. Insert an unlock of T_s after r has executed.
3. Lock all thread locks initially. Spawn $\mathcal{P}_A$ and $\mathcal{P}_B$.

There is a potential deadlock problem if a statement is waiting for another that for some reason does not get executed. This will occur if one statement is in a conditional section of code. We will tackle this later in Section 5.5 on deadlock . A more serious problem is that we require a different lock for each conflicting statement. In these programs that recurse over a variable-sized structure, we cannot tell statically how deep the recursions will go and how

[1]The procedure has also had the side effect of relabelling state 15 to 13, but otherwise none of its transitions have been changed.

many statements will be executed. Thus we do not know in advance how many locks will be needed, and the number of locks for a particular run of the code may be huge. We will consider a situation where we can synchronise the threads using only a pair of locks.

Definition 13 *Two threads T_1 and T_2 are* "dependence ordered" *if for every pair of statements $a < b$ in T_2 with a' and b' such that a waits for a' and b waits for b', if there is a c that waits for c' with $a < c < b$ then we must have $a' < c' < b'$.*

In other words, using the above locking scheme, the locks are grabbed and freed in order. Using an additional lock to stop thread T_1 from running on and freeing more statements, we can synchronise both threads using a pair of locks, l_1 and l_2. The scheme is as follows:

1. For every statement s in $\mathcal{P}_B$ that needs to wait, insert an attempt to gain lock l_2 before s. Free lock l_1 after s.

2. If s needs to wait for statement r in $\mathcal{P}_A$, then insert an unlock of l_2 and a lock of l_1 after r has executed.

3. Lock l_1 and l_2. Spawn partitions $\mathcal{P}_A$ and $\mathcal{P}_B$.

We need to be able to check whether the waits-for information we have generated ($\mathsf{W}_{C_{BA}}$) allows this to be done.

Definition 14 *The dependence $\mathsf{W}_{C_{BA}}$ is* monotonic, *if for every $(x,x'),(y,y') \in \mathsf{W}_{C_{BA}}$, if $y > x$ then $y' > x'$.*

Theorem 3 *We can check a 2FSA R for monotonicity by forming the 2FSA*

$$\mathsf{M} = \mathsf{R}.>.\mathsf{R}^{-1}.>$$

If there is no x such that $(x,x) \in \mathsf{M}$ then R is monotonic.

Proof. If we have an x such that $(x,x) \in \mathsf{M}$, then there exist x', y, y' such that $x\mathsf{R}x' > y'\mathsf{R}^{-1}y > x$. Thus we have a counterexample for monotonicity.

Theorem 4 *If the conflict 2FSA $\mathsf{W}_{C_{BA}}$ is monotonic, then the threads are dependence ordered and can be synchronised using a pair of locks.*

Proof. If $\mathsf{W}_{C_{BA}}$ is monotonic then $a < c$ and $c < b$, which implies $a' < c'$ and $c' < b'$.

This can be extended to any fixed number of locks. We may be able to partition the waits-for 2FSA $\mathsf{W}_{C_{BA}}$ into a number (n) of smaller ones $\mathsf{W}_1, \mathsf{W}_2, \mathsf{W}_3, \ldots$, such that

$$\mathsf{W}_{C_{BA}} = \mathsf{W}_1 \vee \mathsf{W}_2 \vee \ldots \vee \mathsf{W}_n$$

Now each W_i can be checked for monotonicity, and it can be locked with its own pair of locks. We use this technique later in the example in Section 5.4.1.

Each thread needs to compute where to lock and unlock, and which lock to use. We can split $W_{C_{BA}}$ into its first and second components (these are 1FSAs) to find the statements in $\mathcal{P}_B$ that need to wait and the ones in $\mathcal{P}_A$ that are to be waited for.

$$A = \text{Second}(W_{C_{BA}})$$

$$B = \text{First}(W_{C_{BA}})$$

Each thread then needs to compute the current control word in the appropriate one of these 1FSAs and check it for acceptance. Perhaps the simplest way of implementing this efficiently is by cloning the functions for each state of this 1FSA. This means that the lock or unlock statements can be placed only inside the function bodies that correspond to the accept state, and transitions are handled by calling the appropriate cloned function. An example of this is given in Section 6.4.2.

5.4.1 Monotonicity Example

We now illustrate the use of these ideas on the example in Figure 5.8. The waits-for information is shown in Figure 5.9. If we compute the relevant M for this we get the 2FSA shown in Figure 5.10. By inspection we can see that this 2FSA accepts some equal pairs control words. For example by tracing through transitions (B,B) to state 3, (C,C) to state 5, (G,G) back to state 5 and (F,F) to state 7, we can see that any control word of the form $B.C.G*.F$ will be accepted with itself. Although it is relatively simple to check this property by visual inspection, this process is readily automated since we can simply form the 2FSA, $M\wedge =$, and check it for emptiness.

As a result the dependency $B.D \rightarrow A.D$ requires a lock of its own, and the whole set can never be dependence ordered since these cut across the remaining dependencies. We follow the approach described earlier of splitting the waits-for graph, ignoring the $B.D \rightarrow A.D$ dependencies for now, and considering the rest separately. These remaining dependencies are shown in Figure 5.11, compare this with Figure 5.9, here only the (D,D) transition from state 2 has been removed, removing the dependency $B.D \rightarrow A.D$. We can now discover whether this subset can be locked with a single pair of locks. If we compute the relevant M for this we get the 2FSA in Figure 5.12. As can be seen from careful inspection (there are no same symbol transitions (X,X) from state 7), or by computing the 2FSA $M\wedge =$, which turns out to be empty), no identical pairs of control words are accepted by this 2FSA.

```
void main (Tree *);
void Traverse(Tree *);
void Update(Tree *);
main (Tree *m) {
  Traverse(m->l);   // A
  Traverse(m->r);   // B
}
Traverse(Tree *w) {
  if (w == NULL) {return;}
  Update(w->l);     // C
  w->data = w->p->data;  //D
  Traverse(w->r);  //E
}
Update(Tree *t) {
  if (t == NULL) {return;}
  t->data = t->data + t->p->data;  // F
  Update(t->l);     // G
  Update(t->r);     // H
}
{Pa
   States 2;
Start 1; Accept 2;
Transitions[
            [A->2;]
            [A->2;B->2;C->2;D->2;E->2;F->2;G->2;H->2;]
]
  }

{Pb
   States 2;
Start 1; Accept 2;
Transitions[
            [B->2;]
            [A->2;B->2;C->2;D->2;E->2;F->2;G->2;H->2;]
]
  }
```

Figure 5.8: A traversal algorithm operating over the balanced binary tree, with the two partition descriptions corresponding to the two function calls in statements A and B

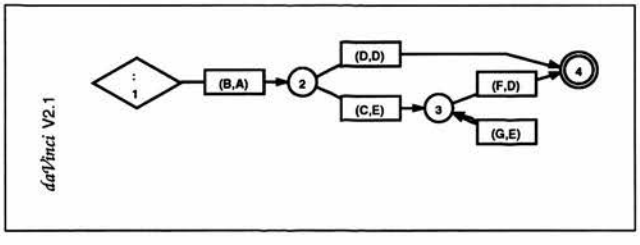


Figure 5.9: Waits for information for binary tree example in Figure 5.8

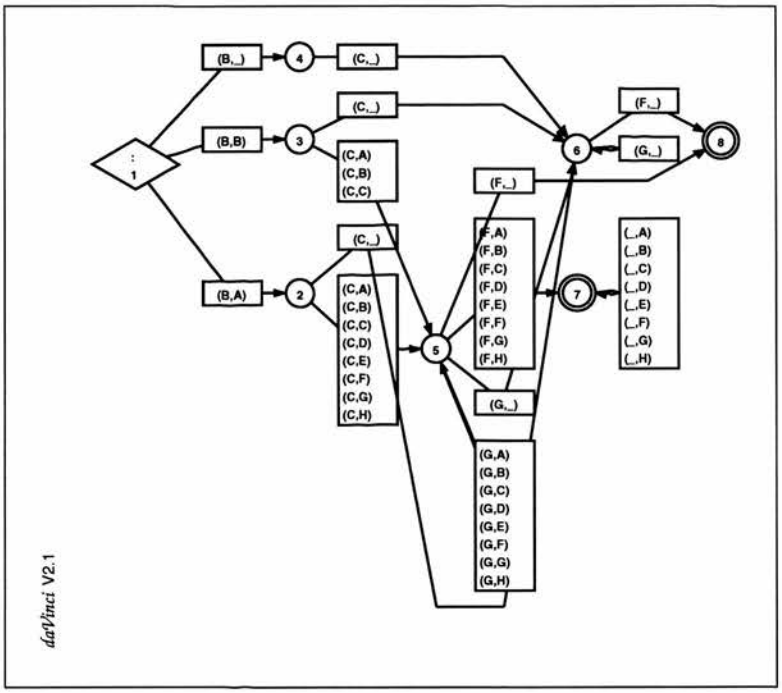


Figure 5.10: The M 2FSA, used to check for monotonicity

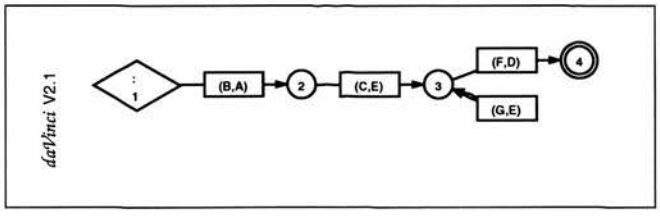


Figure 5.11: Waits for information for a subset of the dependencies.

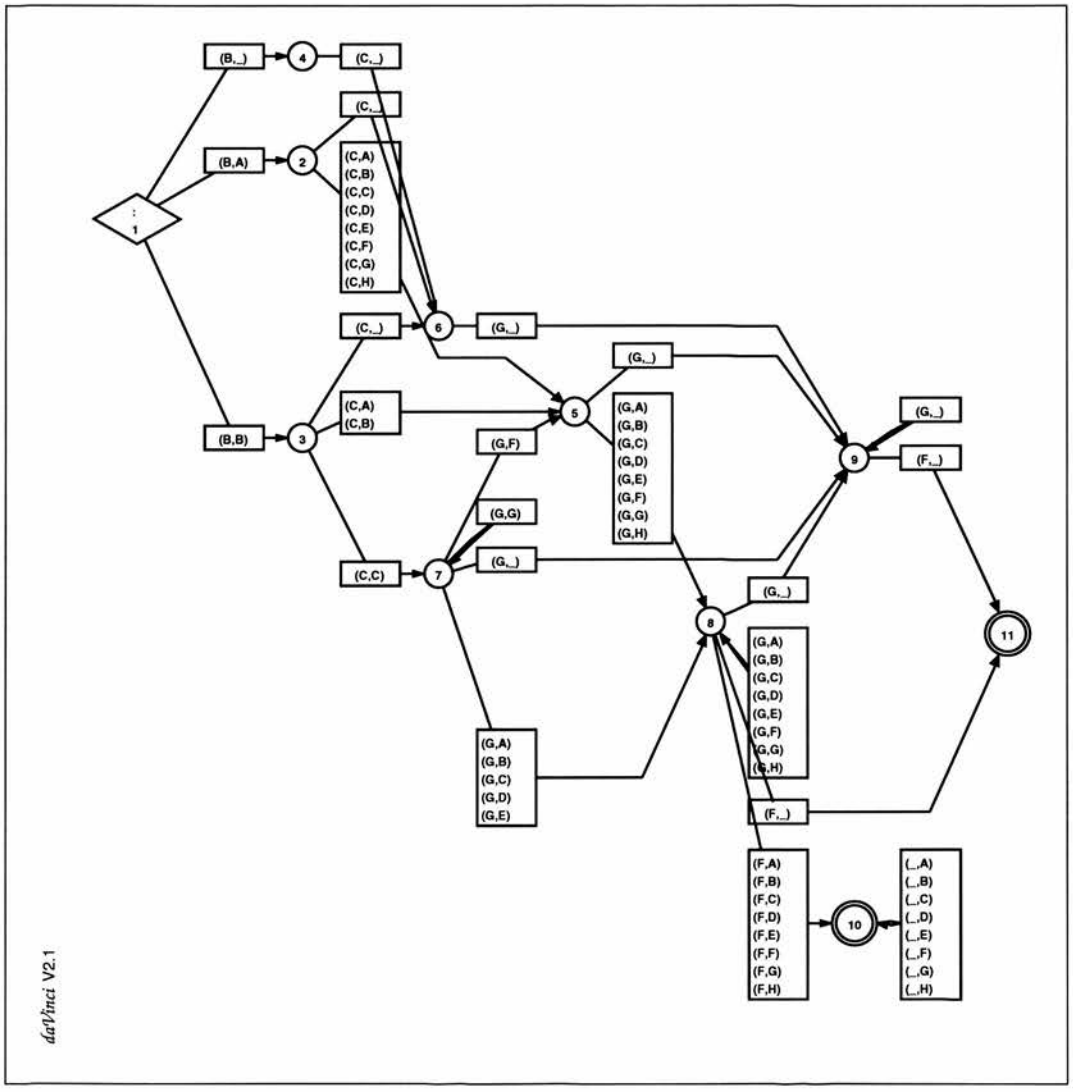


Figure 5.12: The M 2FSA for the subset of dependencies in Figure 5.11

This implies that the cut down 2FSA has the monotonicity property, and therefore these dependencies can be protected by a pair of locks. Since the dependencies $B.D \rightarrow A.D$ can also be protected by a single lock, for this partitioning this code only requires 3 locks to ensure correct execution.

5.4.1.1 Unbalanced Trees

In the above example it was assumed that the binary tree was balanced, in other words the tree was the same depth at all points. It is interesting to follow the analysis of the above code, but instead assume that it is executed over a binary tree that is not necessarily balanced. The different link descriptions are given in Figure 5.13. Since there is more flexibility in the description (a n pointer can point to any parent node of its right-most neighbour), we expect that any dependency information will be less precise and less likely to offer parallelisation opportunities. When we follow through the analysis we see that this is in fact the case. See Figure 5.14 for the waits-for information for this example.

We also have the dependency $B.D \rightarrow A.E$, so we must complete the whole of thread A before $B.D$ can run. Previously we had $B.D \rightarrow A.D$, which left some room for parallelism. Also we now have $B.C.G^k.F$ which must wait for $A.E.E^k.E$ rather than $A.E.E^k.D$, which removes the `Traverse` function call in statement E from running concurrently. All in all, there will not be much room for useful parallelisation in this example.

5.5 Deadlock

We mentioned briefly the issue of deadlock earlier, and we must return to address it here. Strictly speaking, deadlock will occur between two threads, $\mathcal{P}_A$ and $\mathcal{P}_B$, in the notation of this chapter, in the following situation:

> $\mathcal{P}_B$ is waiting for a statement in $\mathcal{P}_A$ and $\mathcal{P}_A$ is waiting for a statement in $\mathcal{P}_B$.

However, our earlier assumption that $\mathcal{P}_A < \mathcal{P}_B$, means that no statement in $\mathcal{P}_A$ will need to wait for one in $\mathcal{P}_B$. Thus this strict deadlock cannot occur.

However, suppose statement P needs to wait for Q. Suppose further that Q executes depending on a condition not known at compile time. This includes the majority of the statements, since the kind of codes we are analysing consist of recursive functions traversing through a pointer structure and terminating on a null pointer, and we do not know the size of the structures at compile time. If Q is never executed, then P will never be unlocked and both threads will be deadlocked.

There are two possible approaches to solve this: a fine-grained approach that uses the actual values of the control words at run time, and an execution dependent approach that enables us to check in certain situations whether deadlock can occur.

```
class Unbalanced_Binary_Tree {

  Tree *l;
  Tree *r;
  int data;
  Tree ~n;
};

//Give the operations
{n:
  States 2;
  Start 1; Accept 2;
  Transitions[
              [(l,l)->1;(r,r)->1;(l,r)->2;]
              [(r,_)->2;(r,l)->2;(_,l)->2;]
  ]
    }
```

Figure 5.13: Source description for the unbalanced binary tree.

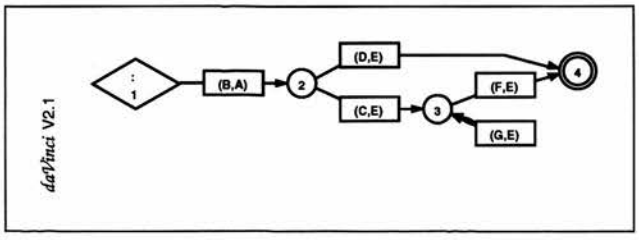


Figure 5.14: Waits for information for the unbalanced binary tree.

5.5.1 Fine-grained approach

Each time that Q unlocks it checks to see if in the time since it was last executed any unlockings have been missed. If, for example, a couple have, it unlocks twice before proceeding. After the last Q has been executed we check again to see if any have been missed. This can be accomplished, if given the FSA that accepts the locking control words and a control word, we can compute the next accepted word.

Definition 15 *For a given FSA* (A) *we define the next accepted control word Next(*A*, x) = y s.t* $y > x$ *and* A *accepts y and if* A *accepts* $z > x$*, then* $y < z$*.*

Define the After *2FSA of* A *that maps each word accepted by* A *to all later words accepted by* A*.*

$$\mathsf{After}(\mathsf{A}) = Double(\mathsf{A}). < .Double(\mathsf{A})$$

Then the Next 2FSA is given by:

$$\mathsf{Next}(\mathsf{A},x) = \mathsf{After}(A) - (\mathsf{After}(A). <)$$

This approach requires us to explicitly compute the control word near each Q, so we cannot use the function cloning approach above. This introduces a certain amount of overhead into the computation. For efficiency reasons we would need to ensure that this approach is only used at the higher levels of the function call tree so that this overhead is not incurred many times in an inner loop of recursive calls.

5.5.2 Execution dependent approach

Definition 16 *Two statements* s_1 *and* s_2 *are* execution dependent *if* t_1 *executes if, and only if,* t_2 *does.*

This approach ensures that all pairs of block/unblock statements are execution dependent. There are a number of situations where statements will be execution dependent. We can exploit these by placing related block and unblock statements in them.

1. Both statements occur in the same basic block, *i.e.* within the same set of conditional nest statements.

2. In a tree structure, child nodes imply existence of the parent node. For a function call that traverses the structure, we can assume that the function calls that operate on parent nodes must have been run, if the corresponding one has been executed for the child nodes.

3. If we assume that the run-time data structure is larger than a certain size, then there will be statements in the program outside conditional blocks that are always executed. For example, for the code in Figure 5.8, if the binary tree structure is assumed to have at least one level, the `l` are `r` pointers in the `main` function will be non null, so when the `Traverse` functions are called we can always assume that statements $A.D$ and $B.D$ occur. If we can always propagate locks outside conditional blocks in this manner, we can assume they will be execution dependent.

In [28, 29] the issue of when statements are execution dependent is considered in more detail. We will use the conditions described previously to ensure that locks are placed in positions where deadlock will not occur.

5.6 Extending to more than 2 threads

So far we have only looked explicitly at parallelising two threads. Much of the preceding approach will generalise directly to the case where we have more than two threads. This is done by treating a many-threaded implementation as a set of pairs of threads. We runs the same analysis over each pair of threads, discover the set of waits-for information for each pair, and generate the locking schemes as before. Now, however, a statement may have to wait for statements from more than one other thread.

This can introduce a large number of locks. We may be able to reduce the number of locks, in the following situation:

- Statement s in thread C, must wait for t_1 in thread A and t_2 in thread B. ($\mathcal{P}_A < \mathcal{P}_B < \mathcal{P}_C$
- t_1 must execute before t_2, due to some other locking between the threads.

In this situation we can remove the s waits for t_1 and just have it wait for t_2. This is similar to the "waits-for" pruning, but we would compute it after we have calculated the "waits-for" information for each pair of threads.

5.7 Conclusion

In this chapter we have outlined a method for aiding the parallelisation of these programs. The method uses the dependency information and a programmer supplied description of the set of threads to show where locks and synchronisation points are required to run those threads concurrently.

Chapter 6

Targeting a Multithreaded Micronet Architecture

This chapter aims to apply the methods described in previous chapters, to code generation for a research multithreaded platform.

Firstly, we describe the Microthreaded Micronet architecture design that was used, and its simulation platform. We then describe our implementation of the analysis methods described in this thesis.

We then take a set of three example programs and create parallel versions of them using the techniques described earlier in this thesis. Where appropriate we will create two- and four-threaded versions in addition to the sequential implementation. Each of these versions are compiled to multithreaded machine code, which is then run on the simulation platform.

6.1 Description of the Multithreaded Architecture

As we mentioned in Section 1 there are a wide variety of parallel machine architectures with different properties. One way to classify them is by communication delays and time penalties of spawning processes, or threads which are a more lightweight alternative. The wide variety of architectures will then fit within a spectrum between the two extreme points:

- A loosely coupled network of workstations. These might be connected in a Local Area Network, or distributed across the planet and joined by the Internet. Communication delay is of the order of milliseconds for a LAN and may be seconds for the Internet. Remote process creation could take seconds for a LAN and minutes in the case of Internet connections.

- A Multithreaded Processor. Communication is via shared memory and threads can be spawned onto on chip *Thread Processing Units* (TPUs). Both delays can be of the order

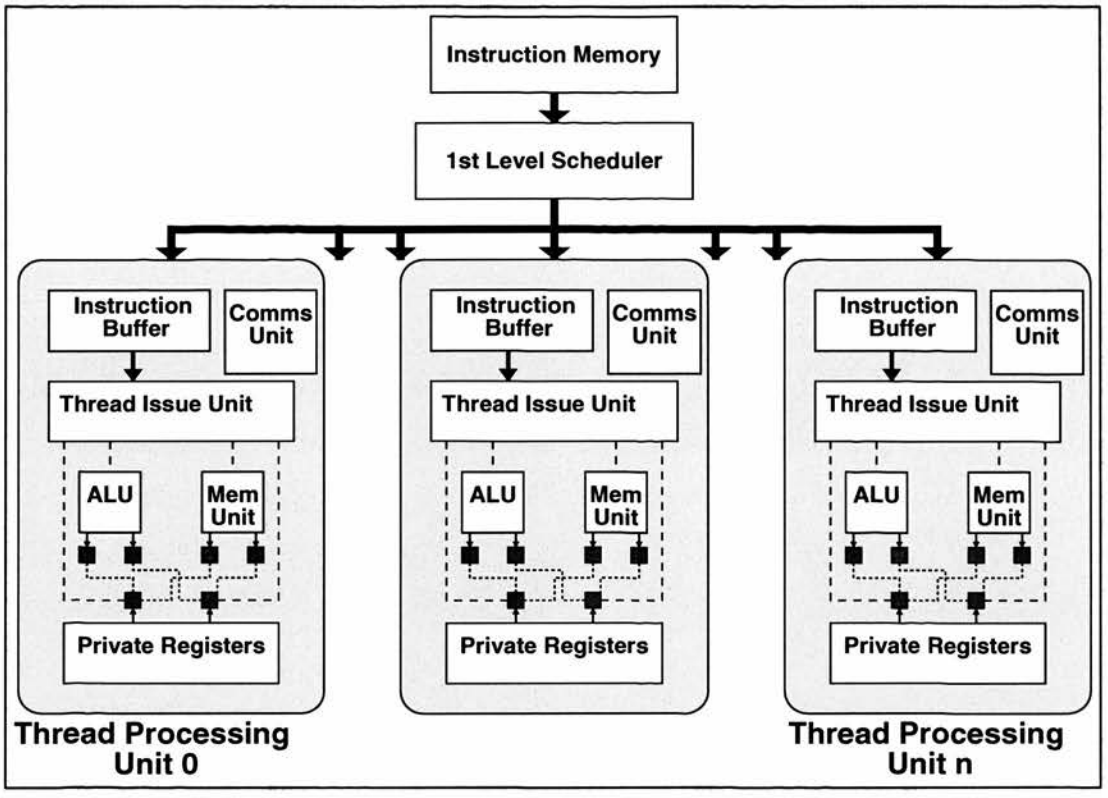


Figure 6.1: The Micronet-based Multithreaded Architecture

of nanoseconds.

Throughout this work we have aimed to extract fine-grained parallelism from the programs, in particular the sets of locks that we describe in Section 5.4.1, where a pair of threads continually block on a pair of locks. In a loosely-coupled system this would require large chunks of computation between each synchronisation point before the parallelisation would be of any use. For large, very long running computations, this might be feasible. In a lower latency, multithreaded processor, however, this fine-grained parallelism may be exploitable for shorter programs.

Therefore, a suitable hardware architecture model for demonstrating the approach of this thesis will be one where thread creation and communication between threads is as cheap as possible. Also, a large, complex data structure can be handled simplest in a situation where there is one memory space shared between the threads. This avoids the issues of distributing the structure over a number of separate memory spaces, such as one would get in a network or cluster of workstations. Thus a multithreaded processor architecture is appropriate as a target for fine grained parallelisation.

Others have found these architectures to be a good fit for certain classes of problems. For example some dynamic unstructured applications such as in [123] multithreaded architectures

are preferred over explicit message passing or shared memory solutions.

The architecture we use (see Figure 6.1) consists of a number of Thread Processing Units (TPUs, three are shown) that access a shared memory. Internally, each TPU is a processor following the Asynchronous Micronet model described in [124].

Briefly, the Micronet approach to processor architectures removes the global clock and decentralises control in the processor, allowing a number of separate *microagents* to collaborate asynchronously. This gives the design much parallelism, even within a single TPU. Each TPU has its own set of microagents, of five basic types:

Private Registers : Allows register locking and register value retrieval.

ALU [1]: Executes the arithmetic and logical operations of the TPU.

MemUnit : Local interface to the shared memory of the processor.

Thread Issue Unit : Takes an instruction from the local TPU Instruction Buffer and initiates it by splitting it into micro operations for the set of microagents that execute them.

CommsUnit : This handles the instructions that communicate between different threads.

These designs can accommodate more than one microagent of each type per TPU. The ALU is a good example of one that can have multiple instances, which would give even more within-TPU parallelism. The architecture we use here is the simplest in that it uses only one of each type.

External to the set of TPUs are the *Instruction Memory* and *1st Level Scheduler* components. The Instruction Memory component acts as an interface to the shared memory that holds the whole instruction list of the current program. The 1st Level Scheduler component passes instructions to whichever is the appropriate TPU for executing them.

Each TPU has the ability to spawn a new thread on another idle TPU. The spawned thread receives copies of the whole register set of the original thread.

6.2 Producing Multithreaded code

In order to handle the multithreaded properties of the architecture a number of special purpose machine instructions have to be used. In this section we describe how to compile C code down into the instructions used in the multithreaded processor.

[1] Arithmetic and Logical Unit.

6.2.1 MAPS Instruction set

The architecture uses an instruction set called *MAPS*, based on the MIPS instruction set. In summary, MIPS instructions can either load or store memory values to a register or operate between registers (for a full description see [125]). MAPS is a MIPS subset in that, for example, the floating point instructions were not yet implemented, and a superset since we add multithreading instructions. These additional instructions handle the extra thread forking properties of the architecture. We have a simple C API that maps a function to each low level threading instruction, allowing direct access to the processor threading features from C programs.

The four additional instructions implemented are as follows:

Fork: frk $thd, $succ, lab This forks a new thread on TPU number $thd. The new thread begins with a jump to the label. Since the TPU may be busy, the spawn may be unsuccessful. A boolean is written into the memory address $succ indicating whether the fork was a success.

Stop: stp $succ If the fork of a thread was successful ($succ is true) it will terminate the thread. If no thread was spawned then nothing is done.

Signal: psg $thd, $add This sends a message to TPU $thd. The value of the message is a memory address $add.

Wait: wat $add This forces a thread to wait for a signal from any other thread. The value of the message is written into the address $add.

The full MAPS instruction set also includes instructions for handling thread interrupt and resume. These would be required, for example, for a full implementation of a POSIX threads library. Our annotations use only the subset described above. Our use is simplified further in the following ways:

- Since the number of spawned threads and the number of TPUs is known as compile time, we know in advance when thread spawning will be successful. Thus we ignore the *$succ* value of the fork and stop instructions.
- We rely only on signalling between threads. The value of the message (*$add*) is unused in both the signal and wait instructions.

There is a lack of any I/O implementation or other operating system support in the simulation platform. To provide some of this functionality there are additional debugging instructions implemented in the simulator. These allow verification that the code is executing correctly, and permit accurate timing of subsections of the code to be produced.

In more detail these instructions are as follows:

```
...
frk(1, &threadsuccess, thread1);
threadzero();
wat(&Waiter);
goto spawnfinish;
thread1:
threadone();
psg(0, 0);
stp (threadsuccess);
spawnfinish:
...
```

```
        ...
        li      $thd, 1
        la      $add, succ
        la      $wait, waiter
        frk     $thd, $add, thread1
        jal     threadzero
        wat     $wait
        j       spawnfinish
thread1:
        jal     threadone
        li      $to, 0
        la      $val, value
        psg     $to, $val
        stp     $succ
spawnfinish:
        ...
```

Figure 6.2: Left: C code spawning two function calls in parallel and Right: Equivalent MAPS assembly

Print register: prg $reg Prints to standard output the value of the register. This is accessed in C by a function of the prototype '`void prg(int Val)`', so that the value of actual C variables can be retrieved directly. This was used to verify the correct execution of the code. To do this a checksum consisting of the sum of all the values stored in the nodes of the structure was constructed after the completion of the algorithm. Thus the operation of the threaded version of the code can be compared with the sequential. These checksum values were used to debug the simulator and threaded implementation and are not reported here.

Print time: ptm Prints the current simulation time. In the C version, sections of code can be bracketed by calls to '`ptm()`' to give fine grained timing information. This was used to benchmark the sections of the parallel algorithm, and to measure any wasted time due to synchronisations.

The fragment in Figure 6.2 shows how to spawn a function `threadone` on another TPU, call a function `threadzero`, then synchronise when both functions terminate. The original C code for this behaviour and the equivalent in MAPS assembly is given.

6.2.2 Implementing locks using MAPS

For simple examples we can implement the locking mechanism we require by simply sending signals between the appropriate threads. For example if thread 2 is required to wait for thread 1 to free a lock we can implement this as thread 1 executing a '`psg(2,0)`', and thread 2 running '`wat(0)`'. We refer to this method of locking as *psig locking*.

This simple approach is not effective if thread 2 may receive signals from a number of different threads, or if a number of '`psg`' are executed before any are received. Since there is

```
int flag[10];
void lock_init() {

  int i;
  for (i=0;i<10;i++) {
    flag[i]=1;
  }
}
void wait(int locknum){

  while(flag[locknum]==1){
  }
  flag[locknum] = 1;
}
void unblock(int locknum) {

  flag[locknum]=0;
}
```

Figure 6.3: The *mem locking* mechanism, implemented without using the MAPS 'psg' and 'wat' instructions.

no queueing of signals within the processor some will be lost. A general solution to this would ideally require the use of the additional MAPS instructions that handle thread interrupt and resume. Since these were not implemented in the available processor simulation, a workaround was used for the more complex examples that required it. We refer to this approach of locking as *memlocking*. The implementation code for this method is given in Figure 6.3. Note that this implementation is still far from perfect, this locking does not work without hardware mutual exclusion, since the '`unlock`' and '`wait`' operations are not fully atomic and it would be possible to get incorrect behaviour if multiple threads accessed them simultaneously. However, since the simulation operates in a purely sequential manner this does not happen in practice.

Note that the thread waiting on the lock does a 'busy-wait' in the `wait()` function; constantly looping to recheck the lock. Since this is on a multithreaded processor this does not have the performance inefficiencies it would have if this were a thread competing for a single processor; the other threads can still proceed when one busy-waits.

It is worth noting that some optimising compilers may remove the test from this loop, since the body of the loop cannot change it. The workaround in C would allow the array to be marked as '`volatile`', and this would stop the compiler from optimising away the test and converting it to an infinite loop. Happily the SUIF compiler did not perform this optimisation with our standard set of passes, so this turned out to be not an issue.

The other issue is one of speed of performing the unlock/wait operations. The memlock signalling will be considerably slower, since we need to write in and out of main memory. The psig locking relies on the fact that each TPU can write directly into one of the registers of the other TPUs. This locking can potentially be implemented as quickly as any other register write.

The speed difference can be readily measured experimentally using the following benchmark:

- Spawn two threads A and B.
- A waits for thread B to unlock it, then unlocks B.
- Meanwhile, B unlocks A, then waits for A to unlock it.

These tests were run on the multithreaded simulator platform. If we subtract this time from the time to spawn two empty threads we get the overhead time for the two synchronisations. The times given are shown in Figure 6.4. Note that these times are the minimum possible time for one synchronisation; in practice, when threads synchronise one may have to wait for a period until the other unlocks. These timings show that there is a ten-fold increase in time for the memlock compared to the psg lock. If many lockings are required in a program, then this overhead could become significant.

We now use the example given in Section 6.4.3 (specifically the code given in Figure 6.22 to see how the overhead of the memlock compares to the psg lock. For this example, running

Synchronisation Type	Time / ns
Memlock	545
Psg	42.6

Figure 6.4: Comparison of the two types of synchronisation we use

Number of threads	Spawning overhead / ns
2	851
4	1720

Figure 6.5: Times for spawning two and four threads

the two-threaded version we get a time of 422.7 microseconds when using psg locking and 424.6 microseconds for memlocking. This is an overhead of only one half of one percent. Thus we will not worry about this affecting the general timings, and use the safer memlocking wherever appropriate. Other codes with more locks would have larger overheads, however.

6.2.3 Thread Spawning Implementation

The correct spawning of a set of threads requires them to synchronise when they complete. This is needed if we want to time the threads, since we must wait for all of them to complete before we can stop the timer. For correctness we use the memlock method to synchronise the threads as they finish. The system we use here uses n locks to synchronise n threads. Thread 0 waits on a lock for each of the other threads that it spawned. Each thread unlocks its lock n when it finishes. Example code for two and four threads is given in Figure 6.6. The total overhead of spawning two and four threads is given in Figure 6.5.

6.3 Implementation Details

The diagram in Figure 6.7 shows how the various tools of the implementation fit together. To summarise the analyser '*tfsa*' produces information to enable the program to be parallelised. The standard SUIF compiler plus custom MAP passes generate the MAPS assembly code. This is run on the simulator of the SPAM multithreaded processor. Each of these tools is described in more detail below.

6.3.1 The 2FSA analyser, '*tfsa*'

This analyses the algorithms description, and produces dependency information to inform the parallelisation of the code. The input files that the analyser works with are:

```
spawn_pair(a,arga,b,argb) {
  frk(1, &thread1success, thread1);

thread0:
a(arga);
wait(0);
goto spawnfinish;

thread1:
 b(argb);
unblock(0);
stp(thread1success,0);
spawnfinish:
```

```
spawn_four(a,arga,b,argb,c,argc,d,argd) {
  frk(1, &thread1success, thread1);
  frk(2, &thread2success, thread2);
  frk(3, &thread3success, thread3);

thread0:
 a(arga);
 wait(0);
 wait(1);
 wait(2);
 goto spawnfinish;

thread1:
 b(argb);
 unblock(0);
 stp(thread1success,0);

thread2:
 c(argc);
 unblock(1);
 stp(thread2success, 0);

thread3:
 d(argd);
 unblock(2);
 stp(thread3success, 0);
spawnfinish:
```

Figure 6.6: Example code of the spawning functions

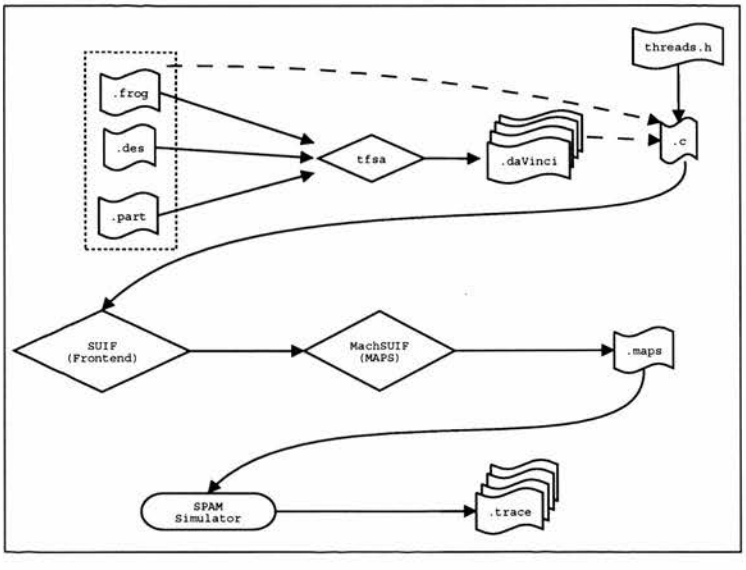


Figure 6.7: How the various pieces of the tool flow fit together

- A description of the algorithm, given as a set of recursive functions reading and writing the structure (.frog)
- A 2FSA description of the data structure (.des)
- A set of partitions that suggest a parallel set of threads This can be given as a separate (.part) file or included with the original program.

All these are supplied by the programmer. A future implementation would perhaps extract the simple algorithmic description from the full original C code implementation.

The tool '*tfsa*' is written in C++, using the GNU compiler generation tools '*flex*' and '*bison*'. It uses the '*kbmag*' library [97], for its basic manipulation of 2FSA descriptions.

The analyser outputs a set of dependency information files. These are in the format for viewing with the 'daVinci' graph visualisation tool [1]. In particular, for each pair of threads, a *waits-for* dependency diagram is produced showing where the statements of the later thread must wait for statements of the earlier to complete before they execute. This is described in Section 5.3. This information is used by the programmer to insert thread-locking statements into the full C implementation of the algorithm. A more sophisticated implementation could automate much of this step.

The system uses a threading library that abstracts a number of different thread libraries:

- Cilk threads
- POSIX threads
- MAPS threads

The choice of library can be made at compile time by the insertion of an appropriate `#define` directive. The Cilk and POSIX threads library versions allow the correctness of the parallel implementation to be verified quickly without full simulation on the processor. The MAPS library version can be compiled to run on the MAPS processor.

6.3.2 Assembly generation using SUIF2

The MAPS version of the C code can be compiled down to generate code for the multithreaded processor.

SUIF2 [126] is a research compiler project, developed at Stanford University and part of the National Compiler Infrastructure (NCI) project. 'Machsuif' is the back-end code generation tool used with SUIF2. They are designed to allow optimising transformations and different back-end code generators to be slotted in as extra passes to a generic compiler framework. We used it here to compile C to MAPS. A front-end SUIF2 pass was created to preserve the

dummy fork and other thread functions, so they could be passed through the other front end compiler passes. A back-end Machsuif pass was written by Shi Zong and Janek Mann. This was based on the existing Alpha platform backend supplied with the Machsuif implementation. This converts the SUIF intermediate form to MIPS instructions, and generates the additional MAPS threading instructions.

6.3.3 The Multithreaded Simulator

The MAPS processor can be simulated by the 'SPAM' tool. This outputs a number of traces, plus any timing results that have been inserted into the code.

It was implemented as a Fourth Year undergraduate project [127], and includes a full model of a memory hierarchy subsystem, written by Japheth Hossel [128]. An overview of whole framework is given in [129].

The model simulates the multithreaded architecture design to the level of individual micro-operations (register requests, ALU operations, memory accesses, register writeback requests). All timings given in this chapter rely on the timing model within the simulator. Each minor operation is assigned a delay in the simulator, compound operations are simulated in terms of these lower level operations and get corresponding estimated execution times. The model is wholly deterministic, yet because of the asynchronous nature of the design two identical instructions at different points in a program may take different times to execute. Thus the timing of an instruction is dependent on which instructions execute before it.

The model also does not attempt to model below the levels described above. As an example, in certain asynchronous architectures, the timing of ALU operations will depend on the particular input values, and not just the type of operation as in our simulation.

In a true asynchronous system there would be variation in the time taken for separate runs of the same executable. This is due to the slight variations in timings of operations and the lack of a global clock to keep everything in lock step. A more capable simulator would model this using randomised delays. Since this is not done all runs of the same executable give the same timings, so there was no point in repeating timing measurements.

6.3.3.1 Sensitivity Analysis

When running simulations of this complexity, it is possible that benchmarked timings will be highly sensitive on particular values chosen for the delays that underpin the simulation. More importantly, from the point of view of this work, the measured speedups we obtain should not be significantly dependent on these values. The simulator used in this study has around 30 different timing parameters; a full sensitivity analysis on all of these is well beyond the scope of this work.

Number of threads	1	2
NormalSim Time	36670	21670
SlowSim Time	36830	21750
NormalSim Speedup	1	1.6922
SlowSim speedup	1	1.6933
% Variation Execution Times	0.44	0.37
% Variation Speedup ratios	N/A	0.067

Figure 6.8: This compares the timings of the standard simulator (NormalSim) with the slower one (SlowSim). All times are in nanoseconds.

Two timing parameters are key to the whole simulation: 'SIG_DELAY' the propagation delay of a signal between microagents and 'HS_DELAY' the time taken for a handshake between two communication units. A slower version of the simulator (denoted *SlowSim*) was built by increasing each of these delays by 50%. All other timings were kept the same. The value of 50% was chosen because increasing these values by a greater factor, for example doubling, was found to greatly increase the time taken for the simulation to run.

In Figure 6.8 we see the results when we run the one- and two-threaded version of the trimesh example (depth4) we use later in Section 6.4.1. We find as expected that the full execution times for each benchmark are increased by a small amount: around 0.4%. However the speedup between them is affected far less, only by 0.067%. This suggests that the measurements we make later are robust with respect to these simulation parameters; we are interested in relative speedups not absolute timing values.

6.3.4 Tracing

A method of visualising the amount of parallelism in the example codes was employed. Firstly, the code was instrumented manually to allow the full set of read and writes to the structure to be captured when the code was run. This was done by adding macros to each node update statement, for example the code:

```
p->data = p->l->data + p->r->data + p->d->data;
```

was annotated in the following way:

```
p->data = p->l->data + p->r->data + p->d->data;
Inst();
```

```
Read(p->l);
Read(p->r);
Read(p->d);
Write(p);
```

The 'Inst' macro marks the boundary between statements, since a statement may read from a variable number of locations. The 'Read' and 'Write' macros output the memory location (literally the value of the pointer) plus the type of access (either *Read* or *Write*) to a log file. Thus, when the code is executed a full trace of memory accesses is built. An additional 'SetThread' macro keeps track (in a global thread counter) of which thread each instruction would be in if the threaded version was being run. The sequential version updates the thread counter to the correct value as each thread is initialised.

A simple tool was written to analyse the trace. It builds a dependency graph from the log, creating a node for each statement and linking nodes when there was a conflict between the instruction. The links are just stored in a n by n matrix, where n is the number of statements in the executed program. For analysing much larger codes a more efficient pointer based implementation would need to be used, but this was sufficient for the sizes of programs used in this chapter. Each link is labelled with the type of conflict (WAW, RAW or WAR). Each instruction was labelled with an instruction number, and thread number. The tool outputs a dependency graph for the whole program, which could be viewed using daVinci. The tool could be run in two main modes:

Raw . All types of dependencies (WAW, RAW, WAR) are shown. The thread numbers of statements are ignored.

Threaded . This shows only dependencies between threads. In order to respecting the particular threading partition, additional dependencies are added that link from each statement to the previous one in that same thread. This forces all statements in each thread to occur sequentially. In addition, any other dependencies between statements that are in the same thread are ignored, since the sequential ordering will ensure that these are respected anyway.

The raw mode is useful to get some idea of the maximum potential parallelism in the code. The threaded mode shows the maximum possible parallelism available using the particular partitioning that is being considered. The approach followed in this thesis has not been to find the best parallelism strategy, just to ensure that a particular strategy executes correctly by respecting the dependencies. This may still involve some inefficiency due to false dependencies being introduced by the approximations that occur in the analysis. Thus if the dependency approximation results in false dependencies that reduce the parallelism this will be exposed

```
main (Tree *node) {

  Tree * root;
  int c, i;
  root = newTree();
  allocate(5, root);
  join(root);
  root = root->d3->d3;
  OutputTime();
    propagate(node->d1);        //      A:
    propagate(node->d3);        //      B:
  OutputTime();
}
propagate(Tree *p) {
  if (p!=NULL) {
    p->data = p->l->data + p->r->data
      + p->d->data;     //      C:
    propagate(p->d1);   //      D:
    propagate(p->d3);   //      E:
    sweepl(p->l);       //      F:
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
    x->l->data = x->l->data + x->data;  //      G:
    sweepl(x->l);       //      H:
  }
}
```

Figure 6.9: Program that recurses over the triangular mesh

when we compare the experimentally achieved parallelism with the figure obtained from this tool. Thus the threaded mode output of the tool gives a fairer yardstick with which to compare the benchmarks in this chapter.

Given a directed acyclic graph (DAG) of dependencies, the depth of the graph is defined as the longest path through the graph. If we assume that each statement takes an equal amount of time to execute, this depth is proportional to the minimum possible execution time. Since daVinci has a feature for computing the graph depth, this was used to obtain these depth values.

Similarly, under this simple execution time model, the total number of nodes in the graph is proportional to the sequential execution time. Thus the ratio of the total number of nodes to the graph depth gives a prediction of the potential speedup and hence the amount of parallelism in the scheme. This model is restricted in that it assumes that each statement executes in the same amount of time and ignores other overheads such as function call timings, conditional statements and thread spawning and communication delays.

6.4 Examples

We will take three examples through the parallelisation process to implementation on the simulated processor. To illustrate the applicability over a range of different data structures, we take examples using each of the binary tree, triangular mesh and rectangular mesh.

6.4.1 Example 1: 'trimesh': Triangular Mesh

For the first example we return to the triangular mesh structure, first described in Figure 3.5. The program detailed in Figure 6.9 recurses over this structure.

There is a certain amount of housekeeping code in the 'main' function. The tree is created using the newTree function to allocate a root node, and the allocate function to generate a tree of the appropriate depth (5 in this case). The join function calls the join code generated as described in Section 3.6. The statement 'root = root->d3->d3;' allows the recursion to begin from within the structure, so that the effects of recursing at the edge of the structure are removed. Finally, the two propagate functions are spawned between a pair of timing statements.

The other examples will use a similar main function to the one shown here. We do not attempt to extract parallelism from these sections. Indeed, the initialisation code has a large amount of parallelism in it, each recursive call of the allocation is independent. Also the join calls will all be independent if the 2FSA description is single-valued as described in Section 3.6.1. The available potential parallelism varies depending on the size of the tree, a factor that can only be known at run-time, but is likely to be far larger than the amount we attempt to extract here. This falls into the class of problems that are *embarrassingly parallel* [130].

The propagate function recurses over two of the sub trees at each node, the sweep functions cuts across horizontally. We assume that the initialisation code, mentioned earlier has been run prior to this phase of the algorithm. We will run through the full parallelisation process for this example. We consider two-threaded and four-threaded versions, and compare them with the sequential case.

We apply the tracing method described in Section 6.3.4 to this example. In Figure 6.13, we can see how the potential speedup in the traced version of this code varies with the depth of the tree and the raw, two- and four-threaded versions we produce in the next sections.

The raw version shows that there is a large amount of parallelism, over 10 for the depth 4, roughly doubling as the depth increases. When we add the two-threaded constraint this parallelism decreases rapidly. This is due to the fact that we are adding dependencies that force the statements to be sequential within the thread scheme we choose. This increases the DAG depth of the dependency graph thereby cutting the speedup. For both the two-threaded and four-threaded versions the amount of parallelism increases very slightly with depth. Here the limiting factor is not the dependencies between threads, since these occur close to the start of the threads as we can see from the task dependency information in Figure 6.10 and Figure 6.12. These do not change as the depth of the structure increases. What limits the parallelism is the unequal sizes of the tasks, which does not vary much as the depth increases either.

6.4.1.1 Two-threaded version

We now describe the process of creating a two-threaded version of the code. We do this by partitioning the above program into two threads, T_0 and T_1, by splitting the computation into two separate function calls at statements A and B. T_0 will consist of all control words beginning with an A, and T_1 will have all those that start with B. The two threads then consist of the sub-tasks:

T_0: $A.C \rightarrow A.D \rightarrow A.E \rightarrow A.F$

T_1: $B.C \rightarrow B.D \rightarrow B.E \rightarrow B.F$

For these two threads we can compute the waits-for information that we describe in Section 5.3, and given in Figure 6.10. This tells us that the statement $B.C$ must wait for statement $A.C$ to finish executing before it is safe for it to be run. This can be ensured by inserting a synchronisation point for the two threads, at these points in the program. Thread T_1 waits (blocks) for a signal from T_0 before statement $B.C$ and T_0 sends the signal after executing $A.C$. Figure 6.10 illustrates the available parallelism by showing the dependencies between the sub-tasks in each of the two threads. We denote a thread sending a signal to thread i by S_i, and a thread blocking for a signal by W, giving the schedule:

T_0: $A.C \rightarrow S_1 \rightarrow A.D \rightarrow A.E \rightarrow A.F$

T_1: $W \rightarrow B.C \rightarrow B.D \rightarrow B.E \rightarrow B.F$

Since there is a good overlap of the two threads, depending on the relative sizes of the sub-tasks, there is good potential for speedup over the sequential version. This is borne out by the trace predictions in Figure 6.13, which predicted a speedup of around 1.3 for this threaded version, which vary slightly depending on the depth.

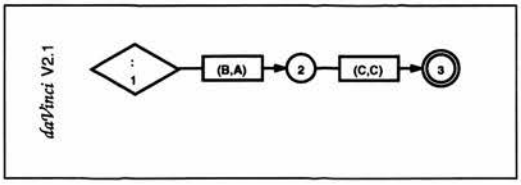
6.4.1.2 Four-threaded version

We now aim to extract more parallelism by producing a four-threaded version. We produce the extra threads by bisecting each of the two threads with statement C and function call D in one thread, and calls E and F in the other. This gives us the following tasks in each thread:

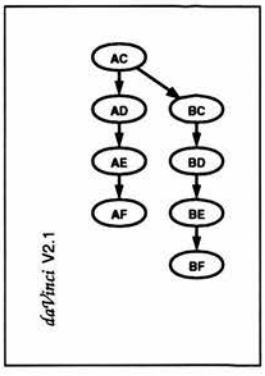
T_0: $A.C \rightarrow A.D.C \rightarrow A.D.D \rightarrow A.D.E \rightarrow A.D.F$

T_1: $A.E.C \rightarrow A.E.D \rightarrow A.E.E \rightarrow A.E.F \rightarrow A.F$

T_2: $B.C \rightarrow B.D.C \rightarrow B.D.D \rightarrow B.D.E \rightarrow B.D.F$

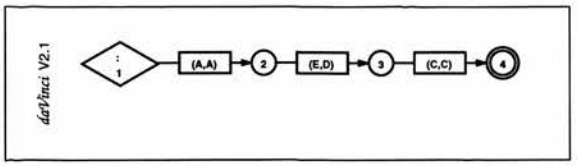


(a) Waits-for information between the two threads

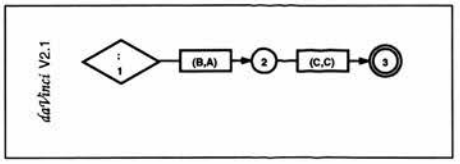


(b) The dependencies between the sub tasks

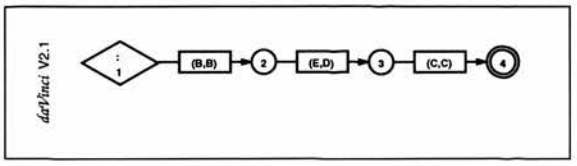
Figure 6.10: The waits-for information and sub task dependencies for the two-threaded version of the triangular mesh example.



(a) Between thread 2 and 1



(b) Between thread 3 and 1



(c) Between thread 4 and 3

Figure 6.11: Waits-for information for the triangular mesh example

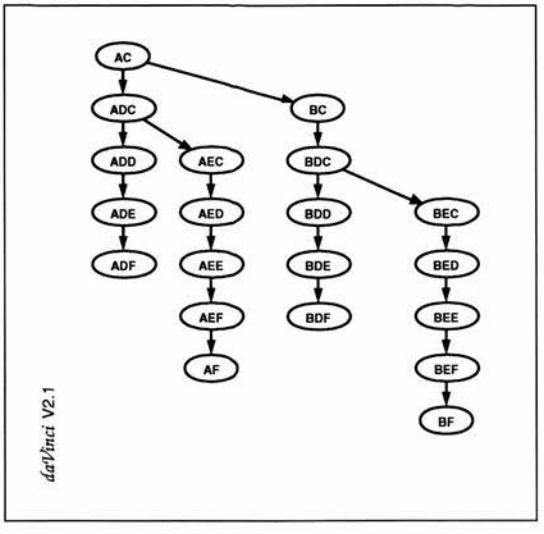


Figure 6.12: A graph showing the dependencies between the sub tasks in each thread in the triangular mesh example.

T_3: $B.E.C \rightarrow B.E.D \rightarrow B.E.E \rightarrow B.E.F \rightarrow B.F$

As before we produce waits-for information between each pair of threads. The analysis finds that there are conflicts between pairs of threads $T_1 \rightarrow T_0$, $T_2 \rightarrow T_0$ and $T_3 \rightarrow T_2$. When the wait-for information is produced it gives us the dependencies between the sub tasks in each thread shown in Figure 6.12. This gives us the waits-for dependencies shown in Figure 6.11. To summarise:

- Between 2 and 1: $A.E.C$ must wait for $A.D.C$
- Between 3 and 1: $B.C$ must wait for $A.C$
- Between 4 and 3: $B.E.C$ must wait for $B.D.C$

To respect these dependencies we require the following schedule:

T_0: $A.C \rightarrow S_2 \rightarrow A.D.C \rightarrow S_1 \rightarrow A.D.D \rightarrow A.D.E \rightarrow A.D.F$

T_1: $W \rightarrow A.E.C \rightarrow A.E.D \rightarrow A.E.E \rightarrow A.E.F \rightarrow A.F$

T_2: $W \rightarrow B.C \rightarrow B.D.C \rightarrow S_3 \rightarrow B.D.D \rightarrow B.D.E \rightarrow B.D.F$

T_3: $W \rightarrow B.E.C \rightarrow B.E.D \rightarrow B.E.E \rightarrow B.E.F \rightarrow B.F$

Again the schedule suggests that the overlap between the threads is small, and we should still expect good speedup improvements over the one- and two-threaded versions. Looking at the trace predictions we obtained earlier in Figure 6.13, we should expect to get speedup of around 2.2, with slightly more for deeper structures.

6.4.1.3 Results

When we run the example code on the microthreaded simulator, we get the timings given in Figure 6.14. Here, and later, we use the term *speedup*, which is calculated as:

$$\text{Speedup on n threads} = \frac{\text{Sequential Execution Time}}{\text{Time for n-threaded version}}$$

Ideally the speedup for n threads would be n. The main issues relevant to the approach we use here that would cause lower speedup are:

- Overhead: the time taken to spawn the threads and the time wasted waiting at synchronisation points. The timings taken earlier suggest that the thread spawning is negligible compared to the time of this benchmark. A chain of dependencies, such as we will see in the later example in Section 6.4.2 could occupy a larger portion of synchronisation time.

Depth of structure	Total statements	DAG depth	Potential Parallelism
4	153	14	10.9
5	649	30	21.6
6	2665	62	43.0

(a) The raw parallelism for various tree depths

Depth of structure	Total statements	DAG depth	Potential Parallelism
4	153	119	1.29
5	649	495	1.31
6	2665	2015	1.32

(b) The potential parallelism for the two thread strategy

Depth of structure	Total statements	DAG depth	Potential Parallelism
4	153	71	2.15
5	649	291	2.23
6	2665	1179	2.26

(c) The potential parallelism for the four thread strategy

Figure 6.13: The trace results for the trimesh example.

Number of Threads	Depth 4	Depth 5	Depth 6
One	126	460	1740
Two	90.0	341	1300
Four	50.6	193	746

(a) Execution times, in microseconds, of the one-, two- and four- threaded programs over data structures of various depths

Number of Threads	Depth 4	Depth 5	Depth 6
Two	1.40	1.35	1.34
Four	2.48	2.38	2.34

(b) Speedups of the two- and four-threaded versions for the various depths

Figure 6.14: Timing results and speedups for the trimesh example.

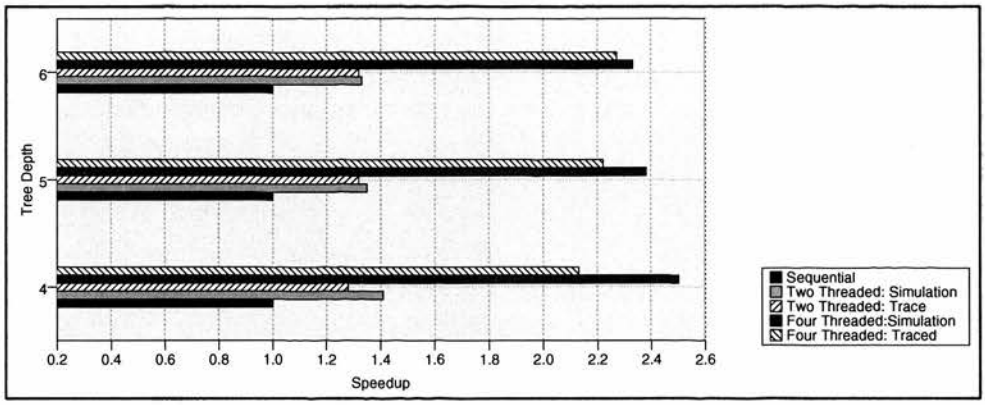


Figure 6.15: A bar chart comparing the predicted speedups from the threaded traces with that obtained from the simulations.

- Unequal sub tasks. The tasks spawned concurrently may be of significantly different execution time. When tasks are irregular and their size cannot be judged at compile time this can be a problem.

Referring to this trimesh example, since much of the synchronisation occurs at the start of the threads and the majority of the execution is completely dependence free, the only barrier to better speedup is the unequal size of the tasks. In this case, this is because for this balanced structure the threads chosen will be of a different size. This can be easily seen from inspecting the two- and four-threaded dependency DAGS we produce from the trace analysis. In fact for an unbalanced structure, where the resolution of the mesh varies at different points, the size of the tasks could be even more different.

The threaded trace information in Figure 6.13 agrees closely with the experimental results. This is illustrated in Figure 6.15, where the normalised execution times of the simulation and threaded trace predictions are plotted together.

This particular example has very small synchronisation delays, in that the threads only wait for a few statements at most. It is sufficiently unbalanced to not give very high speedup figures for larger numbers of threads.

The raw trace information suggests that there is the potential for much more parallelism, up to 20 times speedup for the depth of trees we consider here. An informal inspection of the conflict 2FSAs, suggests it is possible to exploit this by producing larger and larger numbers of threads with more complex sets of locking statements. This was not pursued any further, however.

```
int main(int argc, char *argv[]) {
  Tree * root;
  int c, i;
  root = newTree();
  allocate(5, root);
  join(root);
  OutputTime();
  spawn_pair(A_Tr, (void*) root->l,
             B_Tr, (void*) root->r);
  OutputTime();
}
```

Figure 6.16: Main function of this example.

```
void * A_Traverse(void *arg) {
  Tree * w = (Tree *) arg;
  if (w!=NULL){
  Update(w->l); //      C:
  if (w->p) w->data = 2 * w->p->data;   //      D:
  unblock3();
  A_E_Es_Traverse(w->r);      //      E:
  }
}
void A_E_Es_Traverse(Tree *w) {
  if (w!=NULL){
    Update(w->l);       //      C:
    if (w->p) w->data = 2 * w->p->data; //      D:
    unblock2();
    wait1();
    A_E_Es_Traverse(w->r);    //      E:
  }
}
Traverse(Tree *w) {
  if (w!=NULL) {
    Update(w->l);       //      C:
    if (w->p) {
      w->data =  w->p->data;    //      D:
      w->data = w->data + w->data;
    }
    Traverse(w->r);   //      E:
  }
}
```

```
void * B_Traverse(void *arg) {
  Tree * w = (Tree *) arg;
  if (w!=NULL) {
    B_C_Gs_Update(w->l);        //      C:
    wait3();
    if (w->p) w->data = 2 * w->p->data; //      D:
    Traverse(w->r);   //      E:
  }
}
void Update(Tree *t) {
  if (t!=NULL){
    if (t->p) t->data = t->data + 2 + t->p->data; //    F:
    Update(t->l);       //      G:
    Update(t->r);       //      H:
  }
}
void B_C_Gs_Update(Tree *t) {
  if (t!=NULL){
    wait2();
    if (t->p)    t->data = t->data + 2 + t->p->data; //    F:
    unblock1();
    B_C_Gs_Update(t->l);        //      G:
    Update(t->r);       //      H:
  }
}
```

Figure 6.17: Annotated source code with cloned functions

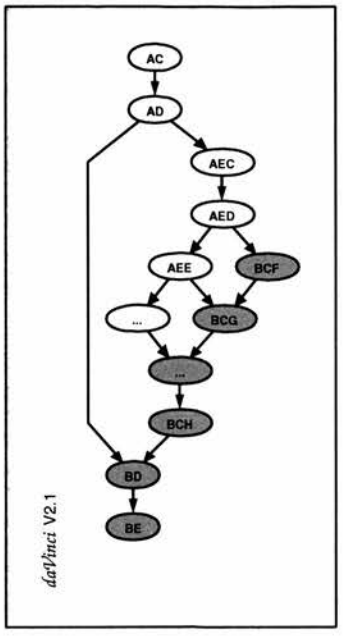


Figure 6.18: The tasks dependency for the bintree example

6.4.2 Example 2: 'bintree': Binary Tree

We now return to the binary tree example we followed in Section 5.4.1. By compiling it down to run on the simulator we can see if the rather more complex parallelisation discovered in that previous chapter actually gives some performance benefit.

The main function for this example is given in Figure 6.16. We parallelise by spawning the two `Traverse` function calls in `Main`. Next, we outline the dependencies that must be respected in order for this code to be concurrently executed. From the previous analysis we observe that we require the statement $B.D$ to wait for $A.D$ to complete, and each $B.C.G^k.F$ to wait for the corresponding one:$A.E.E^k.D$ (for each $k \geq 0$). A dependency graph for the tasks is given in Figure 6.18. The shaded nodes represent the second thread. The nodes containing ellipses represent a chain of nodes (length depending on the size of the tree) that have dependencies. This diagram suggests that there is parallelism, although the chain of dependencies between the tasks could easily limit it in practice.

The annotated source code including the cloned functions is given in Figure 6.17. The `Traverse` function is split into two functions `A_Traverse` and `B_Traverse`, one for each call A and B in `main`. Then functions `A_E_Es_Traverse` and `B_C_Gs_Update` are cloned from `Traverse` and `Update`. These keep track of when statements corresponding to $A.E.E^*.D$ and $B.C.G^*.F$ are executed.

Three locks are used. Lock 3 maintains the $B.D \rightarrow A.D$ dependency. The pair of locks $1,2$ ensure the remaining string of dependencies are respected correctly in the manner of Section 5.4.

	Depth 4	Depth 5	Depth 6
One	27861	58051	117670
Two	26651	56272	116030
Speedup	1.05	1.03	1.01

Figure 6.19: Timing results for the binary tree example.

Depth of structure	Total statements	DAG depth	Potential Parallelism
4	25	14	1.79
5	56	30	1.87
6	119	62	1.92
7	246	126	1.95

(a) The raw parallelism for various tree depths

Depth of structure	Total statements	DAG depth	Potential Parallelism
4	25	23	1.09
5	56	53	1.06
6	119	115	1.03
7	246	241	1.02

(b) The potential parallelism for the two thread strategy

Figure 6.20: The trace results for the binary tree example.

6.4.2.1 Results

The timings achieved for the one- and two-threaded versions are given in Figure 6.19. In summary: there is only a slight (5%) performance improvement of the two-threaded over the sequential. This is for the depth-4 tree, and there is a decrease to only one percent for the depth 6 tree.

The poor concurrency is mainly due to the imbalance in the size of the two threads. The trace information for this example is given in Figure 6.20. Comparisons between the predicted and actual speedup are shown in the barchart in Figure 6.21. These are given for trees of depth 4, 5 and 6. The threaded trace shows that the amount of genuine parallelism is small, and at around 6% agrees reasonably closely with the speedup observed from the simulation results. The decrease in parallelism trend as the depth of the tree increases we observe from the

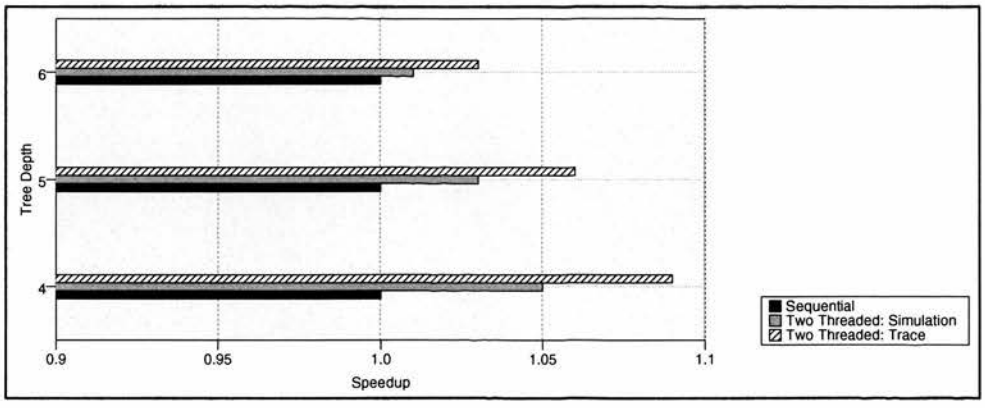


Figure 6.21: A bar chart comparing the predicted speedups of the binary tree example from the threaded traces with that obtained from the simulations.

trace predictions is also confirmed in the experimental results. Generally there is a reasonable correspondence between the predicted and actual, but for such slight amounts of parallelism we should not expect any closer agreement between the two.

The decrease in speedup and the tree depth increases can be explained by consideration of the original program, the derived waits-for information, and inspection of the trace DAG. The total execution time of the program increases exponentially with the depth of the tree, roughly doubling as would be expected for a binary tree. The concurrent section of this code is the phase where the locking and unlocking occurs. The number of locks increases by one as the depth increases by one, which can be seen from the locking scheme presented earlier in this section. Thus this portion only increases linearly in execution time as the depth increases. This doubling of the overall execution time, while the parallel section increases only linearly, causes the predicted and simulated speedups to decrease for deeper structures.

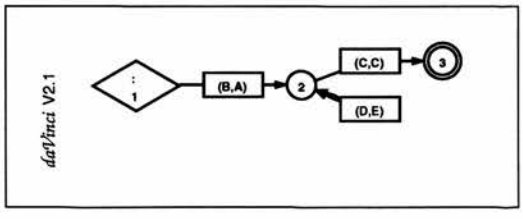
The raw trace shows that there is more parallelism in the code - potentially almost double speedup is possible. However this cannot be extracted using the function call approach we follow in this thesis. Inspection of the raw trace DAG shows that there are a number of unequally-sized independent threads. These threads appear to be composed of interleaved statements from a number of different function calls. As mentioned in Section 5.2.1, the lock generation process we employ makes the assumption that the partitions are not interleaved. As a result, this partitioning could not be exploited using this method.

6.4.3 Example 3: 'rectmesh': Rectangular Mesh

In this example, we take the algorithm of the preceding example, but execute it instead over the rectangular mesh structure. The code is shown in Figure 6.22, the only changes being

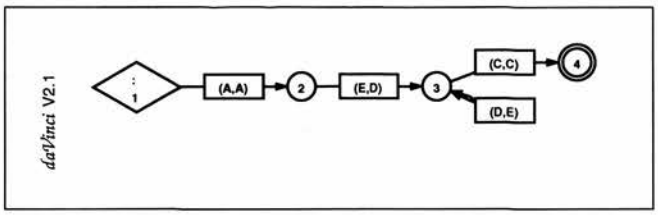
```
main (Tree *node) {
  if (node->d1!=NULL) {
    propagate(node->d1);        //      A:
    propagate(node->d3);        //      B:
  }
}
propagate(Tree *p) {
  if (p!=NULL) {
    p->data = p->l->data + p->r->data + p->u->data
      + p->d->data;     //      C:
    propagate(p->d2);   //      D:
    propagate(p->d4);   //      E:
    sweep1(p->l);       //      F:
  }
}
sweep1(Tree *x) {
  if (x!=NULL) {
    x->l->data = x->l->data + x->data;  //      G:
    sweep1(x->l);       //      H:
  }
}
```

Figure 6.22: The example program that recurses over the rectangular mesh

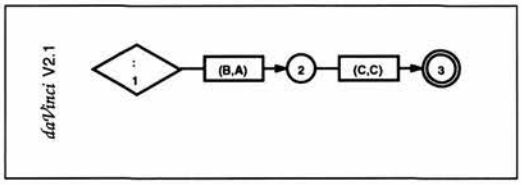


(a) Between thread 2 and 1

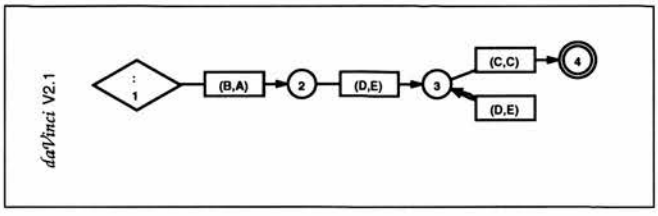
Figure 6.23: The waits-for information between pairs of threads in the two-threaded version of the rectangular mesh code.



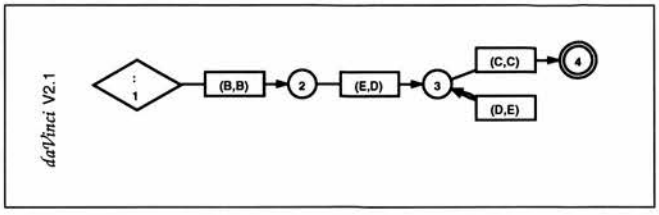
(a) Between thread 2 and 1



(b) Between thread 3 and 1

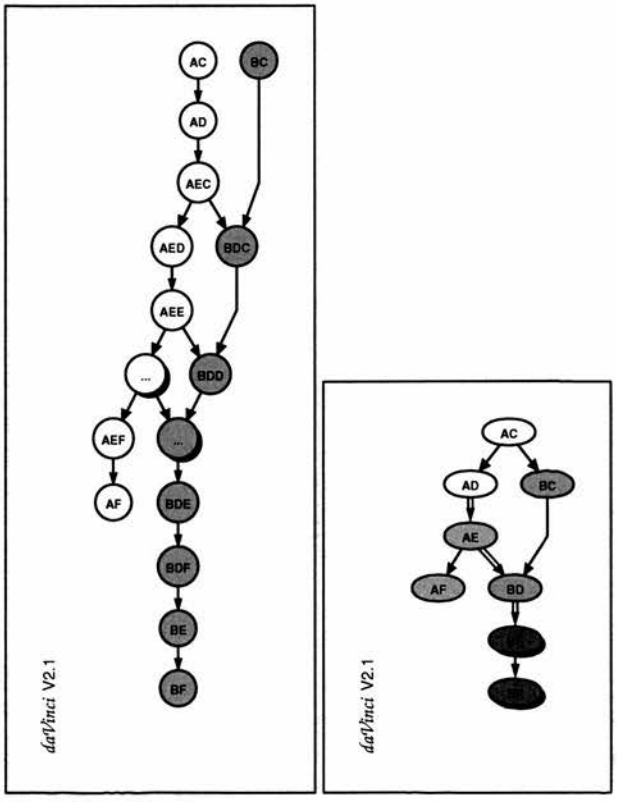


(c) Between thread 3 and 2



(d) Between thread 4 and 3

Figure 6.24: The waits-for information between pairs of threads in the four-threaded version of the rectangular mesh code.



(a) Two threads (b) Four threads

Figure 6.25: Task dependency diagrams for the rectmesh example. The tasks in each thread are shaded differently. Arrows between tasks indicate dependency. In the two thread diagram the ellipsis indicates a chain of dependencies. In the four thread a chain of dependencies between sub tasks of tasks A and B is represented by a double arrow between A and B.

recursing over a different pair of sub-trees and the statement C in the `propagate` function now takes values from the four adjacent nodes (in the l, r, d and u directions), rather than the three l, r and d for the triangular mesh.

We will attempt to build two- and four-threaded versions. The two-threaded will spawn function calls A and B, the 2FSA descriptions for the dependencies between these are given in Figure 6.23. There are dependencies between $B.D^k.C$ and $A.E^k.C$, for all $k \geq 0$. These can be synchronised using a pair of locks, as before.

Similar to the trimesh structure, the four-threaded version will subdivide the '`propagate`' function further into half:

- C and D
- E and F

This yields four threads, and the 2FSA descriptions of the dependencies between them are given in Figure 6.24. In regular expression notation this gives, for all $k \geq 0$ (where applicable):

- Between 2 and 1: $A.E.D^k.C$ must wait for $A.D.E^k.C$
- Between 3 and 1: $B.C$ must wait for $A.C$
- Between 3 and 2: $B.D.D^k.C$ must wait for $A.E.E^k.C$
- Between 4 and 3: $B.E.D^k.C$ must wait for $B.D.E^k.C$

This yields a complex system of locks, and uses seven locks in total. A pair of locks is required for each of threads 2 and 1, threads 3 and 1 and threads 4 and 3. One lock is additionally required for the dependency between thread 3 and thread 1.

The task dependencies for the two- and four-threaded versions are shown in Figure 6.25. The two-threaded one has a chain of synchronisations, yet appears to offer some potential for two-way parallelism. The four-threaded does not appear to have much additional potential. It may still show some speedup compared to the two-threaded version.

6.4.3.1 Results

The experimental timings for the one-, two- and four-threaded versions of the rectangular mesh example are given in Figure 6.26. Also included are the predictions from the threaded and raw trace runs. As with the trimesh example, the raw trace prediction suggests potential for a large amount of parallelism; for this example 20-way parallelism is theoretically possible.

The predicted and experimental speedups are shown again in Figure 6.27. For the two-threaded version they are in close agreement. For the four-threaded version the predictions are

No. of Threads	Sim. Time (μ s)	Sim. Speedup	DAG Depth	Trace Speedup
1	839	1	611	1.00
2	804	1.04	586	1.04
4	662	1.27	537	1.14
Raw	N/A	N/A	30	20.4

Figure 6.26: The timing results for the rectangular mesh example

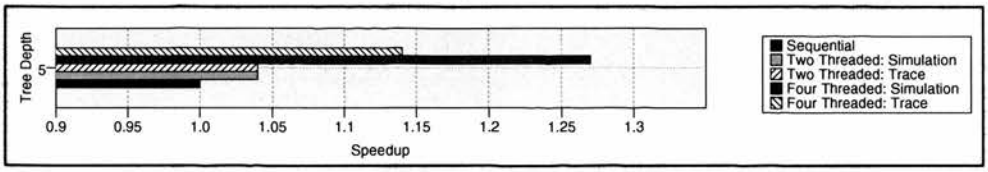


Figure 6.27: A bar chart comparing the predicted speedups of the rectmesh example from the threaded traces with that obtained from the simulations.

reasonably good, the simulations offer slightly more parallelism than the trace predicts. We would not generally expect to do better than the predictions, but this discrepancy is probably due to the fact that the predictions use a very crude model of the execution time. Again, very close agreement cannot really be expected. This result does at least suggest that the overhead of the locking mechanism is not significantly slowing the implementation.

6.5 Summary and Conclusions

The triangular mesh example was run in two- and four-threaded versions with good speedups of 1.4 and 2.4 respectively. The binary tree example was run as two threads and showed a very modest speedup of 3 to 5%. The rectangular mesh example gave a slight speedup (4%) over two threads, and slightly better over four (27%).

The rectmesh and trimesh examples are worth comparing, since they are essentially the same algorithm run over data structures with different linkage patterns. Indeed if we ignore the additional structure descriptions they would present almost identical code to the compiler. However the dependency information extracted is very different. This leads to two quite different locking strategies to give correct execution in two and four-threaded versions. The trimesh gave better speedups, although the threaded trace results suggested that there was more parallelism available.

The trimesh example could have given further parallelism, indeed it would have been possible to extract larger numbers of threads by subdividing the existing ones and using our method to generate the appropriate locking mechanisms. The rectmesh example did not appear to have much additional parallelism that could be extracted using our method, anything beyond four

threads would have been fruitless.

The dependency analysis we perform can produce approximation in the presence of recursion in link directions. This occurs for the trimesh and rectmesh examples, but not for the bintree. Additionally computation of waits-for information that is used to generate the positioning of synchronisation lock can involve some approximation when pruning infinite chains of dependencies. Without these approximations, assuming a perfect set of synchronisation points, we would expect the amount of parallelism discovered in the experimental runs to depend only on the particular partitioning chosen. To discover how much effect the analysis approximation had on the extracted parallelism, we need only to compare the experimental speedups with the threaded trace speedup predictions. Since these were found to be consistently close in value. Therefore we can say that for these examples the approximation is not contributing much to the amount of parallelism observed; it is mainly due to the particular choice of partitions.

Actual simulation of these codes on a multithreaded architecture was valuable in exposing some issues. Firstly, the variability on the quality of parallelism on the relative sizes of sub tasks in the threads. Secondly, the granularity of the synchronisation points: how much computation occurs between the locking and unlocking in the situation where there is a long chain of dependencies.

The trimesh example highlighted an important limitation of this approach. The quality of parallelism depends heavily on the relative sizes of the sub tasks. With irregular-sized data structures it is impossible to do much to detect this statically, too much depends on the actual balance of the data structure. For wholly balanced trees (as this example was), some analysis is possible to determine the sizes of the sub tasks. Generalising somewhat, if we knew in advance or could specify in the description the degree to which the underlying tree was balanced this could be used to aid the parallelisation process.

Comparison of the two-threaded bintree and rectmesh examples is worthwhile since each had interlocking chains of dependencies. Both the bintree and rectmesh had these chains, and neither gave good quality parallelism.

The threaded trace results shed some light on this. Generally the poor performance was not not due to the multiple synchronisations, but because one thread was forced to wait nearly until the other was finished before it could begin its execution. Thus only a small proportion of the statements could actually execute concurrently.

We will return to these issues when we draw wider conclusions in the next chapter.

Chapter 7

Conclusions and Future Work

This thesis has attacked some of the issues surrounding automatic parallelisation in the presence of recursive structures. 2FSAs are a useful formalism for manipulating structure linkage information, and program dependency information.

In this chapter we will first take a broader look at some of the issues raised by this work. We will then discuss some of the shortcomings of our work and suggest future work that could improve and extend on what has been achieved so far.

7.0.1 Structure descriptions

First we will look critically at our 2FSA structure description approach. If we assume that these 2FSA descriptions must be supplied by the programmer this is an additional burden and uses a formalism which most will be unfamiliar with. However, this approach may not be as onerous on programmers as it first appears:

- Link directions can often be built up from expressions in simpler descriptions. This involves expressing one link direction in terms of others, for example in Section 3.3 the *previous* pointer in our ongoing binary tree example was given as the inverse of the *next* direction. Also the descriptions of the rectangular mesh in Section 3.4.3 are given as reversals of a 2FSA which was directly derived from a simple set of pointer relationships.

- Many of the link descriptions we consider are quite similar in construction. The trimesh and rectmesh have quite similar descriptions and the reversal approach we used for deriving them works for both. Thus it is possible to gain quickly some 'expertise' in producing these descriptions.

- The rich set of operations available for 2FSAs (see Section 3.2.2) means that simple tests can be constructed to check for correctness of the supplied descriptions. For example we can test that the link pointer has a unique target by checking the 2FSA describes a one-valued relation. We can test whether two nodes link to the same node via a pointer. The

presence of a pointer that is always null can be detected since this is an empty 2FSA relation. Also we can expose pathological linkage descriptions. An example would be a pointer that can link a node to any nodes and thus would be insufficiently precise to give useful dependency analysis.

- We can extend the use of the description to automate some messy data structure housekeeping tasks. For example, we showed how to generate code for initialising these structures and producing the join code in Section 3.6. This could be extended to other tasks, such as checking the pointer linkages are correct after pointers have been altered during a structural update. This means that time can be invested in producing accurate link descriptions which can automate other programming tasks in addition to the main task of aiding the parallelisation process.

- We can derive simple properties directly from the descriptions that can be used in dependency analysis. In Section 3.7.1 and Section 4.5.1.1, simple questions about possible aliasing of pointers can be answered by forming compound 2FSA descriptions.

- 2FSA formalism is powerful enough to describe most data structures that other methods in the literature do. It may not be as powerful as certain other formalisms in expressing relations between pointers see, for example shape types [80] or graph types [79] for strictly more powerful formalisms. However, for most example data structures it has been shown that a 2FSA description is sufficient to specify any aliasing properties within the structure.

7.0.2 Dependency analysis

We have shown how to use the descriptions to produce dependency analysis for simple recursive programs. Even for such a simple language there was a particular complexity arising from functions that recurse in non-generator directions. This was resolved using a process of approximating closures of 2FSA relations. The process of constructing the closures was a heuristic approach, but could be made rigorous by use of a test that determined whether the derived approximation could be used safely in dependency analysis.

The alternative option of using a more powerful formalism for holding dependency information was rejected. A more powerful transducer language would have required complex approximation and heuristic techniques to be applied later in the process when we were dealing with the dependency information. By approximating earlier to 2FSAs we could apply more general manipulations directly when we came to use this information.

7.0.3 Extracting Parallelism

The dependency analysis produces complex 2FSA information, which cannot in general be understood by simple inspection. We followed a method whereby once the programmer had partitioned a program into separate threads the dependency information for that threaded version could be drastically simplified. This meant that extracting the information required to synchronise that set of threads was possible.

7.0.4 Architecture Simulations

We used a simulation of a multithreaded architecture to investigate whether parallel versions of a set of example programs showed worthwhile speedup. We also used a trace-based technique to extract information about how much parallelism was available in the programs, and also in the particular thread schemes we chose.

The general conclusion here was that once a set of threads was chosen, the synchronisation scheme generated by our dependency analysis permitted speedups close to the theoretical maximum. The application of raw trace analysis to each example showed that frequently there was far more potential parallelism in the program than our chosen partitioning exploited. This thesis did not attempt to address this partitioning phase in great detail, to do so could have improved these results significantly.

7.1 Further Work

We will take each step of the approach in turn and see what limitations it had and thus what direction future research in that area could follow.

7.1.1 Structure Description Language

Since the 2FSA descriptions seem to be powerful enough to capture most data structure dependencies, adopting a more sophisticated description language would be counter-productive. Instead, the main improvements would come from further attempts to simplify the process of building a set of descriptions for a data structure.

Some assistance in the specification process would be useful. This could be in the form of some automatic verification of these specifications. Alternatively an Interactive Development Environment could show diagrams of real example structures from directly manipulated specifications. This would allow an interactive debugging of structure descriptions.

Another approach would perform a derivation of 2FSA linkage diagrams from concrete example structures. The programmer could write code that creates and links a structure and a search of possible 2FSA representations that give rise to those linkages could be performed. A

choice of possible matching 2FSA descriptions could then be presented to the programmer and a selection of the correct one could be made.

Taken to its logical conclusion we would perhaps ideally like to extract such descriptions directly from program code. Rather than use the 2FSA description of a link to build up the join code as we do in Section 3.6 we could reverse this process. A restricted language could be designed for writing pointer linkage code, and this would be used to generate 2FSAs descriptions directly.

Extracting pointer aliasing information from C code is an ongoing research area, which is still far from solved (see [42] for a recent survey). Here, the 2FSA descriptions could be the representation used by other automatic shape analysis techniques. Since it can cope with a wide variety of structures and permits program dependency analysis it would make a good back-end to any automatic shape extraction.

7.1.2 Analysis

The main restriction of our program analysis would have to be the simplicity of the programs to which we applied our approach. Extending this to more general codes would be an important component of future work in this area.

In terms of extending the program model, we can identify the following as important restrictions that would be worth addressing.

Control Structures Throughout this thesis we have considered programs to be sets of recursive functions. This could be extended to include loop structures (`for` and `while` loops). This would require extending the control words so that the iteration number of each loop instance was encoded. This is fairly straightforward; we just append a symbol representing the loop head statement to the control word for each loop iteration, an approach that would extend to handle nested loops. This permits us to treat loop bodies much like function calls. Induction variables updated by the body of the loop would have to be resolved correctly, just as the function parameter in recursive functions. In general this would require some approximation approaches similar to the ones we use in Section 4.4.

Pointer Assignments The main restriction here is that we have only considered code that reads and writes into data values of a structure. Extending this so as to allow writing to the pointers within a structure (with statements such as '`x->p = y;`') would increase the applicability of our approach. This is a significant change, as it would mean that the very structure and linkages could change dynamically . If we want to permit this in general we have to consider the following two situations depending on whether '`p`' is a link direction or a generator:

Link direction: As part of the analysis we would need to check that the assignment does not invalidate the given description for that link. This can be done directly from the information we extract. Assuming that this link's 2FSA description permits linking to a number of nodes, it may also be possible to take account of the information about exactly where the pointer now points in subsequent analysis.

Generator direction: Assignment here changes the data structure. We could disallow such assignments since they contradict the no-sharing property of the generator directions. Alternatively, we could make sense of this for dependency purposes by assuming:

1. The entire sub tree at 'p' is destroyed.
2. All of the target nodes subtree 'y' has been copied over, or shared by that assignment.

These changes would significantly alter the dependency analysis of the programs.

Multiple structures Currently we only consider one global data structure. We could extend this to multiple structures by combining them into one structure that we treat in the manner as suggested in Section 4.1. For example, if one structure is nested inside another the linkage descriptions can simply be composed. The problems of efficiency would become more significant with this as it could greatly increase the number of generator directions. The complexity of the 2FSA manipulations we use are sensitive to the size of the underlying alphabet. We could counter this by using the kind of alphabet manipulations that are suggested for control words in Section 4.8 to reduce this.

7.1.3 Multithreaded simulation

The performance simulations on the multithreaded target were useful in that they revealed a number of possible problem areas where improvements could be made to our parallelisation process. However, they were restricted in that in order to keep the simulation running times to practical levels [1] only short runs over small (depth ≤ 6) were possible.

7.1.3.1 Relative sizes of sub tasks

The issue of handling relative sizes of sub tasks is strongly related to knowledge about how the data structure is balanced. If we recurse through a balanced structure then we can be more confident that sub recursions will be of equal size. Ultimately, to solve this one has to collect some information about the data structure at run time. However, some assumptions can be made at compile time if the structure is assumed to be balanced, or nearly balanced.

[1] Many of the runs took 50 to 70 hours to complete.

7.1.3.2 Granularity of synchronisations

The method supplies a set of synchronisation points for each thread in the partition. In these situations it is difficult to predict in advance how much parallelism is actually available. The quantity of computation that is done between each synchronisation point is key to this. A static technique that could be used to make some prediction would need to estimate the execution time of such blocks of code. A fairly crude model using estimated execution times for a small set of features including function calls, control constructs and arithmetic statements could give sufficient estimates. In our architecture the overhead of a synchronisation is about the time to execute 20 or 200 instructions, depending on the synchronisation type. An improved prediction scheme would want to check that there were many times this number of instructions between synchronisation points.

7.1.3.3 Trace Analysis

The use of traces to produce estimates of speedups for a particular partitioning aimed to give a measure with which to compare the efficiency of the locking scheme on the microthreaded architecture. It was not designed to estimate the parallelism in the synchronisation scheme, only in the chosen partitioning. However, the estimates produced agreed closely with simulation results. These traced programs executed in less than a second, the analysis of the traces to build dependency DAGs took no longer than a few seconds for the examples we looked at. This is significantly shorter than the simulation times.

This suggests that extending this trace approach to take account of the locking scheme could give realistic speedup estimates. This would be a good alternative to the simulation approach following in this thesis, and would permit examples with larger depth structures to be explored.

In addition the trace approach could be further exploited. It could be used to aid the partitioning process by allowing experimentation with a large number of candidate partition schemes. The process of parallelising a program could have many iterations between partition generation and parallelism prediction, with use of a trace tool to predict parallelism between iterations.

7.1.3.4 Partitioning

In this thesis we have assumed that the programmer supplies the partitioning directly. However it would be an improvement if some programmer assisted partitioning of programs could also be achieved. For example, the programmer could be responsible for partitioning the data structure rather than the program. The analysis tool could derive program partitions directly from this based on the kind of access information we derive.

The partitioning could be almost completely automatic. The tool would create partitions, checks dependencies between them and then alternate between splitting or merging partitions until something sensible is created. It could perhaps start from the point of having the whole program in one partition and repeatedly splitting until a granularity appropriate for a particular machine is achieved. Alternatively it could split all statements into separate threads and merge them based on estimated communication overhead. Both approaches could use a programmer supplied partition as a starting point.

Together with the methods in this thesis of producing synchronisation points, this could give a powerful method of extracting parallelism from programs that use complex pointer-based data structures.

Bibliography

[1] M. Fröhlich and M. Werner, "Demonstration of the interactive graph visualisation system daVinci," in *Proceedings of the DIMACS Workshop on Graph Drawing*, vol. LNCS No. 894, (Princeton, USA), 1994.

[2] D. K. Arvind and T. Lewis, "Static analysis of recursive data structures," in *Languages and Compilers for Parallel Computing, 10th International Workshop LCPC'97* (Z. L. et al, ed.), no. 1366 in Lecture Notes in Computer Science, (Minneapolis, Minnesota, USA), pp. 423–426, Springer-Verlag, Aug. 1997.

[3] D. K. Arvind and T. Lewis, "Dependency analysis of recursive data structures using automatic groups," in *Languages and Compilers for Parallel Computing, 11th International Workshop LCPC'98* (S. C. et al, ed.), Lecture Notes in Computer Science, (Chapel Hill, North Carolina, USA), Springer-Verlag, Aug. 1998.

[4] D. K. Arvind and T. Lewis, "Safe approximation of data dependencies in pointer-based structures," in *Languages and Compilers for Parallel Computing, 13th International Workshop LCPC'00* (S. M. et al, ed.), no. 2017 in Lecture Notes in Computer Science, (Yorktown Heights, NY, USA), Springer-Verlag, aug 2001.

[5] M. Schmidt, "Time-bounded Kolmogorov complexity may help in search for extra terrestrial intelligence (SETI)," *Bulletin of the European Association for Theoretical Computer Science*, vol. 67, pp. 176–180, 1999.

[6] Beberg, A. L.Lawson, and J.McNett, "The distributed.net project, located at `http://www.distributed.net`."

[7] D.K.Arvind and R. Rangaswami, "Asynchronous multithreaded processor cores for system level integration," in *IP*, 1999.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[9] J. Hummel, L. J. Hendren, and A. Nicolau, "A framework for data dependance testing in the presence of pointers," in *Proceedings of the 1994 International Conference on Parallel Processing (ICPP)* (K. C. Tai, ed.), (North Carolina, USA), August 1994.

[10] A. Asenov, A. Brown, S. Roy, and J. R. Barker, "Developement of a parallel 3D finite element power semiconductor device simulator," *IEE Colloquium on Physical Modelling of Semiconductor Devices*, vol. 1995/064, 1995.

[11] S. R. A. Asenov, A. R. Brown and J. R. Barker, "Topologically rectangular grids in the parallel simulation of semiconductor devices," *VLSI Design*, vol. 6, no. 1–4, pp. 91–95, 1998.

[12] A. Wieland, R. Leighton, and W. Morgart, "Aspirin for MIGRAINES," in *Proceedings of the 1988 International Neural Network Society Conference*, 1988.

[13] S. H. Bokhari and D. J. Mavriplis, "The Tera multithreaded architecture and unstructured meshes," Tech. Rep. ICASE No. 2000-13, NASA, December 1998.

[14] J. Barnes, "A modified tree code," *Journal of Computational Physics*, vol. 87, pp. 161–170, 1990.

[15] R. L. Lewis, *Vortex Element methods for fluid dynamic analysis of engineering systems*. Cambridge University Press, 1991.

[16] G. J. Pringle, *Numerical Study of Three-Dimensional Flow using Fast Parallel Particle Algorithms*. PhD thesis, Department of Mathematics, Napier University, Edinburgh, Feb. 1994.

[17] U. Banerjee, *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[18] Y. A. Lui and S. D. Stoller, "Eliminating dead code on recursive data," in *Proceedings of the 6th International Static Analysis Symposium*, no. 1694 in LNCS, (Venezia, Italy), September 1999.

[19] Y. A. Liu, "Dependence analysis for recursive data," in *Proceedings of the IEEE 1998 International Conference on Computer Languages*, (Chicago, USA), May 1998.

[20] M. S. Hecht, *Flow Anlysis of Computer Programs*. Programming Languages series, Elsevier, Holland, 1977.

[21] R. Book, *Formal Language Theory*. Academic Press, 1980.

[22] A. Cohen and J.-F. Collard, "Applicability of algebraic transductions to data-flow analysis," in *Proceedings of Parallel Architectures and Compilation Techniques, PACT'98*, (Paris, France), Oct. 1998.

[23] A. Cohen and J.-F. Collard, "Applicability of algebraic transductions to data-flow analysis," Tech. Rep. 9, PRiSM, U. of Versailles, Jan. 1998.

[24] A. Cohen, J.-F. Collard, I. Djelic, and P. Feautrier, "A transduction-based framework for recursive programs analysis," tech. rep., PRiSM, U. of Versailles, 1998.

[25] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Principles of 22nd ACM SIGPLAN-SIGACT symposium on Programming Languages*, (San Francisco, California), 1995.

[26] F. E. Allen, "Interprocedural data flow analysis," in *Proceedings of IFIP Congress*, pp. 398–402, North-Holland Publishing Co., Amsterdam, 1974.

[27] G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, PLDI*, pp. 72–84, 1998.

[28] A. Cohen and J.-F. Collard, "Fuzzy array data-flow analysis – part II: Recursive programs," Tech. Rep. 96-036, PRiSM, `http://www.prism.uvsq.fr`, dec 1996.

[29] D. Barthou and J.-F. Collard, "Fuzzy array dataflow analysis," *Journal of Parallel and Distributed Computing*, vol. 40, pp. 210–241, Feb. 1997.

[30] J.-F. Collard, D. Barthou, and A. Cohen, "Maximal static expansion," in *Symposium on Principles of Programming Languages*, (San Diego, CA, USA), pp. 98–106, Jan. 1998.

[31] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by constuction of approximations of fixpoints," in *Conference Rec. of Principles of Programming Languages, POPL-4*, (Los Angeles, California), pp. 238–252, ACM Press, New York, Jan. 1977.

[32] J.-F. Collard and M. Griebl, "Array dataflow analysis for explicitly parallel programs," in *EURO-PAR'96, Vol. I* (L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, eds.), Lecture Notes in Computer Science 1123, pp. 406–413, Lyon, France: Springer-Verlag, 1996.

[33] P. Feautrier, "Dataflow analysis of scalar and array references," *International Journal of Parallel Programming*, vol. 20, pp. 23–53, Feb. 1991.

[34] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array dataflow analysis and its use in array privatization," in *Proceedings of ACM Conference on Principles of Programing Languages*, (Charleston, South Carolina), pp. 2–15, Jan. 1993.

[35] A. Cohen and P. Wu, "Dependence testing without induction variable substition," in *Ninth International Workshop on Compilers for Parallel Computers*, (Edinburgh, Scotland, UK), 2001.

[36] C. Lengauer, "Loop parallelization in the polytope model," in *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings* (E. Best, ed.), vol. 715 of *Lecture Notes in Computer Science*, pp. 398–416, Springer-Verlag, 1993.

[37] M. Griebel and C. Lengauer, "The loop parallelizer LooPo," in *Proc. Sixth Workshop on Compilers for Parallel Computers* (M. Gernt, ed.), pp. 323–334, Aachen, Germany: Forschungzentrum Jülich GmbH, December 1996.

[38] A. Cohen, J.-F. Collard, P. Feautrer, M. Barreteau, D. Barthou, and V. Lefebvre, "The interplay of expansion and scheduling in PAF," tech. rep., PRiSM, U. of Versailles, 1998.

[39] C. Ancourt, F. Coelho, B. Creusillet, and R. Keryell, "How to add a new phase in PIPS," in *Sixth Workshop on Compilers for Parallel Computers" (CPC 96)*, (Aachen, Germany), pp. 19–30, December 1996.

[40] R. Seater and D. Wonnacott, "Polynomial time array dataflow analysis," in *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1-3, 2001. Revised Papers* (H. G. Dietz, ed.), vol. 2624 of *Lecture Notes in Computer Science*, Springer, 2001.

[41] A. Cohen, J.-F. Collard, and M. Griebl, "Array data-flow analysis for imperative recursive programs," Tech. Rep. 96/035, Laboratory PRiSM, University of Versailles, France, 1996.

[42] M. Hind, "Pointer analysis: haven't we solved this problem yet?," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 54–61, ACM Press, 2001.

[43] J. Larus and P. Hilfinger, "Detecting conflicts between structure accesses," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, vol. SIGPLAN Notices 23(7), (Atlanta, Georgia), July 1988.

[44] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," Tech. Rep. CS-2000-12, Department of Computer Science, University of Viginia, 2000.

[45] J. L. Ross and M. Sagiv, "Building a bridge between pointer aliases and program dependences," *Nordic Journal of Computing*, vol. 8, pp. 361–386, 1998.

[46] Y.-S. Hwang, P.-S. Chen, J. K. Lee, and R. D.-C. Ju, "Probabilistic points-to analysis," in *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1-3, 2001. Revised Papers* (H. G. Dietz, ed.), vol. 2624 of *Lecture Notes in Computer Science*, Springer, 2001.

[47] Y.-S. Hwang and J. Saltz, "Compile-time analysis on programs with dynamic pointer-linked data structures," Tech. Rep. UMCP-CSD CS-TR-3744, Department of Computer Science, University of Maryland, Nov. 1996.

[48] R. Rugina and M. C. Rinard, "Pointer analysis for multithreaded programs," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia, USA), May 1999.

[49] R. Ghiya and L. J. Hendren, "Putting pointer analysis to work," in *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (San Diego, CA, USA), pp. 121–133, Jan. 1998.

[50] A. Cohen, J.-F. Collard, and M. Griebl, "Data flow analysis of recursive structures," Tech. Rep. 96/018, Laboratory PRiSM, University of Versailles, France, Sept. 1996.

[51] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," in *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Denver, Colorado, USA.), Nov. 1999.

[52] R. Ghiya, L. J. Hendren, and Y. Zhu, "Detecting parallelism in C programs with recursive data structures," in *Proceedings of the 1998 International Conference on Compiler Construction*, vol. LNCS 1383, pp. 159–173, Mar. 1998.

[53] Y.-S. Hwang and J. Saltz, "Applying traversal-pattern-sensitive pointer analysis to dependence analysis," Tech. Rep. CS-TR-3848, Department of Computer Science, University of Maryland, Nov. 1997.

[54] N. D. Jones and S. Mucknick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Ninth Annual ACM Symposium on Principles of Programming Languages (POPL)*, (Albuquerque, New Mexico), pp. 66–74, ACM Press, January 1982.

[55] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (New York, NY), pp. 230–241, ACM Press, 1994.

[56] A. Deutsch, "A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations," in *Proceedings of the IEEE International Conference on Computer Languages*, (Oakland), pp. 2–13, IEEE Press, Apr. 1992.

[57] L. J. Hendren and G. R. Gao, "Designing programming languages for analyzability: A fresh look at pointer data structures," in *Proceedings of the 1992 International Conference on Computer Languages*, (Oakland, CA, USA), pp. 242–252, Apr. 1992.

[58] P. Feautrier, "A parallelization framework for recursive tree programs," tech. rep., Laboratoire PRiSM, May 1998.

[59] P. Fradet, R. Gaugne, and D. L. Metayer, "An inference algorithm for the static verification of pointer manipulation," Tech. Rep. 980, IRISA, June 1996.

[60] M. I. Schwartzbach, J. L. Jensen, M. E. Jorgensen, and N. Klarlund, "Automatic verification of pointer programs using monadic second order logic," in *Proceedings of the ACM SIGPLAN Conference on Programming Language design and implementation*, (Las Vegas, Nevada, USA), ACM, June 1997.

[61] R. Gaugne, "Static debugging of C programs: detection of pointer errors in recursive data structures.," Tech. Rep. 1119, INRIA, Aug. 1997.

[62] D. C. Oppen, "Reasoning about recursively defined data structures," Tech. Rep. STAN-CS-78-678, Stanford Artificial Intelligence Laboratory, July 1978.

[63] P. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of Computer Science and Logic*, no. 2142 in LNCS, pp. 1–19, 2001.

[64] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, (Los Alamitos, California), IEEE Computer Society, 2002.

[65] P. Jouvelot and R. Triolet, "Newgen: A language-independent program generator," Tech. Rep. A-191, ENSMP/CRI, July 1989.

[66] I. D. Macdonald, *The theory of groups*. Oxford University Press, 1968.

[67] C. C. Sims, *Computation with Finitely Presented Groups*. Cambridge University Press, 1995.

[68] J. Giavitto, O. Michel, and J. P. Sansonnet, "Group based fields," in *Parallel Symbolic Languages and Systems, International Workshop PSLS* (I. T. et al., ed.), vol. LNCS 1068, (Beune, France), pp. 209–215, Springer-Verlag, October 1995.

[69] O. Michel, "Design and implementation of $8\frac{1}{2}$, a declarative data-parallel language," Tech. Rep. 1012, Laboratoire de Recherche en Informatique, Dec. 1995.

[70] J.-L. Giavitto and O. Michel, "Declarative definition of group indexed data structures and approximation of their domains," in *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pp. 150–161, ACM Press, 2001.

[71] O. Michel, D. De Vito, and J. P. Sansonnet, "$8\frac{1}{2}$: data-parallelism and data-flow," in *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming* (E. Ashcroft, ed.), World Scientific, May 1996.

[72] N. Klarlund and M. I. Schwartzbach, "Graphs and decidable transductions based on edge constraints," in *Trees in Algebra and Programming - CAAP'94, 19th International Colloquium* (S. Tison, ed.), vol. 787 of *Lecture Notes in Computer Science*, (Edinburgh, U.K.), pp. 187–201, Springer, April 1994.

[73] L. Hendren and J. Hummel, "Abstractions for recursive pointer data structures, improving the analysis and transformation of imperative programs," in *SIGPLAN Vol 27, 7-8*, pp. 249–260, 1992.

[74] J. Hummel, L. J. Hendren, and A. Nicolau, "Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs," Tech. Rep. 93-8, Department of Information and Computer Science U-C Irvine, Feb. 1993.

[75] J. Hummel, L. J. Hendren, and A. Nicolau, "Applying an abstract data structure description approach to parallelizing scientific pointer programs," Tech. Rep. 92-10, Department of Information and Computer Science U-C Irvine, June 1995.

[76] J. Hummel, L. J. Hendren, and A. Nicolau, "A language for conveying the aliasing properties of dynamic, pointer-based data structures," in *Proceedings of the 8th International Parallel Processing Symposium*, (Cancun, Mexico), Apr. 1994.

[77] J. Hummel, L. J. Hendren, and A. Nicolau, "A general data dependence test for dynamic pointer-based data structures," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. SIGPLAN Notices 29(6), (Orlando, Florida, USA), pp. 100–104, June 1994.

[78] N. Klarlund and M. I. Schwartzbach, "A domain-specific language for regular sets of strings and trees," in *DSL '97 - First ACM SIGPLAN Workshop on Domain-Specific Languages*, (Paris, France), January 1997.

[79] N. Klarlund and M. I. Schwartzbach, "Graph types," in *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, (Charleston, South Carolina), pp. 196–205, Jan. 1993.

[80] P. Fradet and D. L. Metayer, "Shape types," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Paris, France), pp. 27–39, Jan. 1997.

[81] J. Engelfriet and V. van Oostrom, "Logical description of context-free graph languages," *Journal of Computer and System Sciences*, vol. 55, pp. 489–503, 1997.

[82] M. Benedikt, T. Reps, and M. Sagiv, "A decidable logic for describing linked data structures," in *ESOP '99: European Symposium on Programming, Lecture Notes in Computer Science* (S. D. Swierstra, ed.), vol. 1576, pp. 2–19, Springer-Verlag, Mar. 1999.

[83] C. B. Jay, "Partial Evaluation of Shaped Programs: Experience With FISh," in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99) San Antonio, Texas, January 22-23, 1999: Proceedings* (O. Danvey, ed.), pp. 147–158, BRICS, 1999.

[84] C. B. Jay and J. R. B. Cockett, "Shapely types and shape polymorphism," in *Programming Languages and Systems - ESOP '94* (D. Sannella, ed.), vol. 788 of *Lecture Notes in Computer Science*, (Edinburgh, UK), pp. 302–316, Springer, April 1994.

[85] R. Ghiya and L. J. Hendren, "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C," in *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Florida, USA), pp. 1–15, Jan. 1996.

[86] F. Corbera, R. Asenjo, and E. Zapata, "Accurate shape analysis for recursive data structures," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing* (S. M. et al, ed.), no. 2017 in Lecture Notes in Computer Science, (Yorktown Heights, NY, USA), 2000.

[87] F. Corbera, R. Asenjo, and E. L. Zapata, "New shape analysis techniques for automatic parallelization of C codes," in *ACM International Conference on Supercomputing (ICS'99)*, (Rhodes, Greece), ACM Press, june 1999.

[88] F. Corbera, R. Asenjo, and E. Zapata, "Towards compiler optimization of codes based on arrays of pointers,," in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, (Washington, DC, USA.), 2002.

[89] Y.-S. Hwang and J. Saltz, "Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures," in *Languages and Compilers for*

Parallel Computing, 10th International Workshop LCPC'97 (Z. L. et al, ed.), no. 1366 in Lecture Notes in Computer Science, (Minneapolis, Minnesota, USA), pp. 131–145, Springer-Verlag, Aug. 1997.

[90] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," in *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, (St. Petersburg Beach, Florida, USA), pp. 16–31, ACM Press, New York, Jan. 1996.

[91] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, (San Antonio, TX, USA), pp. 105–118, ACM Press, Jan. 1999.

[92] D. Chase, M. Wegman, and F. Zadeck, "Analysis of pointers and structures," in *SIGPLAN Notices Vol. 26 No. 6*, p. 296, 1990.

[93] R. Wilhelm, M. Sagiv, and T. Reps, "Shape analysis," in *Proceedings of CC2000, 9th International Conference on Compiler Construction* (D. A. Watt, ed.), vol. 1781 of *Lecture Notes in Computer Science*, (Berlin, Germany), Springer, Apr 2000.

[94] R. Gupta, "SPMD execution of programs with pointer-based data structures on distributed memory machines," in *Journal of Parallel and Distributed Computing*, vol. 16, pp. 92–107, 1992.

[95] D. B. A. Epstein, J. W. Cannon, D. E. Holt, S. V. F. Levy, M. S. Paterson, and W. P. Thurston, *Word Processing in Groups*. Jones and Bartlett, 1992.

[96] C. Froughny and J. Sakarovitch, "Synchronized rational relations of finite and infinite words," *Theoretical Computer Science*, vol. 108, pp. 45–82, 1993.

[97] D. Holt, "Version 2 of a recent package for Knuth-Bendix in monoids, and automatic groups." `http://www.maths.warwick.ac.uk/~dfh/`.

[98] D. B. A. Epstein, D. F. Holt, and R. S. E., "The use of Knuth-Bendix methods to solve the word problem in automatic groups," *Journal of Symbolic Computation*, 1991.

[99] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebra," *Computational problems in abstract algebras*, pp. 263–297, 1970.

[100] M. Greenlinger, "Dehn's algorithm for the word problem," *Communications on Pure and Applied Mathematics*, vol. 13, pp. 67–83, 1960.

[101] P. L. Chenadec, "Analysis of dehn algorithm by critical pairs," Tech. Rep. 438, INRIA, august 1985.

[102] J. R. Buchi and D. Siefkes, *Finite automata, their algebras and grammars : towards a theory of formal expressions*. Springer-Verlag, 1989.

[103] J. E. Hopcroft and U. J. D., *Formal Languages and their relation to automata*. Addison-Wesley, 1969.

[104] A. Cohen, *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, University of Versailles, 1999.

[105] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

[106] G. Ramalingham, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, pp. 1467–1471, Sept 1994.

[107] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[108] W. J. Savitch, *Abstract machines and grammars*. Little, Brown & Company, 1982.

[109] S. Ginsburg and S. Greibach, "Abstract families of languages," *Memoirs of the American Mathematical Society*, vol. 87–90, pp. 1–32, 1966.

[110] S. Eilenberg, *Automata, languages and machines*. Academic Press, 1974.

[111] S. Ginsburg and M. A. Harrison, "One-way nondeterministic real-time list-storage languages," *Journal of the ACM*, vol. 15, pp. 428–445, 1968.

[112] A. V. Aho, "Nested stack automaton," *Journal of the ACM*, vol. 16, pp. 383–406, July 1969.

[113] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "A general theory of translation," *Mathematical Systems Theory*, vol. 3, p. 193, 1969.

[114] F. Pereira and R. Wright, "Finite-state approximation of phrase structure grammars," in *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, (Berkley, CA, USA), pp. 246–255, 1991.

[115] E. G. Evans, "Approximating context-free grammars with a finite-state calculus," *Proceedings of ACL-EACL*, 1997.

[116] W. Kelly, W. Pugh, E. Prosser, and T. Shpeisman, "Transitive closure of infinite graphs and its application," *International Journal of Parallel Programming*, vol. 24(6), pp. 579–598, Dec. 1996.

[117] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[118] P. J. Koopman and P. Lee, "A fresh look at combinator graph reduction," in *Proceedings of the SIGPLAN '89 Conference on Programming Language and Design*, vol. 24 of *SIGPLAN Notices*, pp. 110–119, ACM Press, June 1989.

[119] A. J. Bennett, *Parallel Graph Reduction for Shared-Memory Architectures.* PhD thesis, Imperial College, sept 1993.

[120] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Montreal, Canada), 1998.

[121] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Palo Alto, CA: Morgan Kaufmann, 2002.

[122] D. R. Butenhof, *Programming with POSIX Threads.* Addison Wesley, ISBN 0-201-63392-2, 1997.

[123] L. Oliker and R. Biswas, "Efficient parallelization of a dynamic unstructured application on the Tera MTA," Tech. Rep. 43190, Lawrence Berkeley National Laboratories (LBML), May 1999.

[124] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello, "Micronets: A model for decentralising control in asynchronous processor architectures," in *Proceedings of the International Workshop on Asynchronous Design Methodologies*, (London, England), pp. 190–199, IEEE Computer Society Press, May 1995.

[125] J. Hennessey and D. Patterson, *Computer Architecture, a Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1996.

[126] The Stanford SUIF Compiler Group, "The SUIF compiler system," June 2004. `http:\\www-suif.stanford.edu`.

[127] A. Stanley, "Power estimation of micronet-based multi-threaded architectures." Fourth Year Project Report, Department of Computer Science, University of Edinburgh, 2001.

[128] J. E. Hossell, "Compiling Java byte-code for multi-threaded architectures.," Master's thesis, Department of Computer Science, University of Edinburgh, 1999.

[129] D.K.Arvind, J. Hossell, A. Koppe, T. Lewis, R. Rangaswami, J. Schneiders, A. Stanley, and S. Zhong, "A design framework for asynchronous multithreaded architectures," in

Proceedings of the 2001 Asynchronous Forum (S. Furber, ed.), (Edinburgh, Scotland, UK), 2001.

[130] G. Wilson and R. Irvin, "Assessing and comparing the usability of parallel programming systems," Tech. Rep. CSRI-321, University of Toronto, March 1995.