THE DESIGN AND IMPLEMENTATION OF A STRUCTURED BACKEND FOR THE MCCAT C COMPILER

by

Christopher M. Donawa

School of Computer Science
McGill University, Montréal

March 1994

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

[redacted] DESIGN AND IMPLEMENTATION OF A [redacted]
BACKEND FOR THE MCCAT C COMPILER

Abstract

The McCAT system contains a highly optimizing, parallelizing C compiler that has been designed to support a high-level, structured intermediate representation, SIMPLE. Although the high-level abstraction enables sophisticated analyses such as pointer analysis, it limits the effective detection and exploitation of opportunities for fine-grained parallelism through low-level transformations like register allocation and instruction scheduling.

This thesis presents LAST, a low-level intermediate representation that exposes important architectural details, yet retains enough abstraction to simplify retargeting of the compiler.

LAST is structured, thus allowing easy access to information gathered by previous high-level analyses at SIMPLE, and also provides an elegant and simple framework for developing low-level analysis and transformation phases. To illustrate these features and their effectiveness, some example phases are presented, along with results from a small suite of benchmarks.

Résumé

Le système McCAT comprend un compilateur C qui optimise et parallelise, il est conçu pour permettre l'utilisation de SIMPLE, une représentation intermédiaire structurée de haut niveau. Bien que le niveau élevé d'abstraction permette des analyses sophistiquées, telle que l'analyse des pointeurs, il restreint le dépistage efficace et l'exploitation des possibilités du parallelisme à travers des transformations de bas niveau, telle que l'allocation des registres et l' ordonnancement des instructions.

Cette thèse présente LAST, une représentation intermédiaire de bas niveau qui met en évidence les détails importants de l'architecture, tout en conservant l'abstraction nécessaire pour permettre l'utilisation du compilateur en vue de plusieurs processeurs cibles.

LAST est structuré, permettant l'accès facile à l'information récoltée précedemment par les analyses SIMPLE de haut niveau, et fournit également une structure simple et élégante pour le développement d'analyses de bas niveau et de transformations. Les caractéristiques et leur efficacité sont illustrées par des exemples (d'analyses et de transformations) ansi que les résultats d'une série des tests de performance.

Genealogy of McCAT

This chapter presents the genealogy of the McCAT C compiler, from its origins in September 1990 to the present day. It is meant as a testament to all those who have worked on the project, an acknowledgment of their hard work to produce a system with which the author is proud to be involved.

The genealogy time line is broken down into individual semesters, as the components were roughly organized around the academic timetable.

September, 1990. The origins of McCAT started with a compiler course project implemented by Erik Altman. Professor Gao taught the course. The project consisted of implementing a list scheduler in a version of GNU's GCC C compiler, using the RTL intermediate representation.

January, 1991. In this winter term, two students joined Professor Laurie Hendren: Bhama Sridharan and Maryam Emami. Bhama started on deciphering the GCC front-end as part of her compiler course. It was during this course that the name McCAT(for the McGill Compiler/Architecture Testbed) was first coined.

May, 1991. As part of a combined compiler and architecture project, Bhama implemented McCAT's first version of loop unrolling. Working with her was another student, Chandrika Mukerji, who extended Erik's work on scheduling to include the Shieh-Papachristou algorithm. It was this work on scheduling that the author cloned to implement McCAT's current list scheduler.

A special note of appreciation is due to Ravi Shankar, who was always present to explain the many intricacies of C.

September, 1991. After a long cross-country trip, Chris Donawa arrived in Montreal, and along with a few other students (including Justiani and Mizuho Iwaihara) took the compiler course taught by Laurie. For the first time, the modified GCC front-end being worked on by Bhama was used for the course, and the assignments and projects were used as a testing ground for several ideas about SIMPLE. In particular, Mizuho's final project contributed heavily to some aspects of the simplify algorithm.

During this time, Maryam started her work on the call-graph and points-to analysis, while Bhama began development of the c-dump module.

January, 1992. After a very pleasant autumn, a frigid and severe winter gripped the city of Montréal. During this polar period, both Justiani and Chris began laying the foundations of their respective theses: the array dependence analyzer for Justiani, and the design and implementation of LAST for Chris. At the same time Ana Erosa, the next McCAT member, arrived to begin studies at McGill.

By March, Maryam had finished the call-graph framework, and the first version of the points-to analysis. Bhama finished implementing the SIMPLE intermediate representation.

May, 1992. After a late start, spring finally arrived in Montréal, and although the summer never really blossomed, the celebrations for the 350th anniversary of the city of Montréal made up for any lack of sun. During the festivities, Justiani started her implementation of the array dependence module, and Chris continued work on extending his course project to handle the entire SIMPLE C grammar, and generate code for the DLX architecture. Chris also started organizing the McCAT development environment.

The summer was also a flurry of activity for Bhama, as she wrote the McTAG module, implemented both reaching definitions and live variable analysis for SIMPLE, and graduated with a Masters degree.

September, 1992. After some construction delays, the ACAPS research group moved into their new lab (formerly two adjacent student offices). Into this new setup arrived the Rakesh Ghiya, who started work on extending Maryam's second version of points-to analysis to handle function pointers. Justiani finished an initial version of her array dependency analysis module.

For the second time, Laurie taught the compiler course with the McCAT compiler, and students projects developed into useful modules. Luis Lozano implemented Briggs' extension of Chaitin's graph-coloring register allocator. Matilda Leung wrote the initial version of the LAST interpreter, Sandro Mazzucato developed a SPARC code generator and Claudia Pateras and Mary Iarocci extended Bhama's original loop unroller. Clark Verbrugge began work on generalized constant propagation, and Ana on the McCAT restructuring module.

January, 1993. This winter started later that the previous year, and was fortunately not as bitter. Shielded from the milder-but-still-cold weather, V.C. Sreedhar finished the unnest module, and cut the last chains to the old GCC framework that had up to

then been a heavy weight, by replacing the memory management routines. In addition, Sreedhar implemented an initial version of the ALPHA intermediate representation.

Some of the more ambitions projects lingered on into this semester, as Clark, Luis and Ana finished the work on their projects. Rakesh began his research into practical heap analysis methods.

May, 1993. Montréal was treated to a splendid summer, with near perfect weather. Besides the arrival of the sunshine, two talented undergraduate students on NSERC summer scholarships arrived to work over the summer: Christopher Lapkowski and Patrick Betremieux. Christopher wrote the McCAT XWindows interface and function inliner, and Patrick developed the source file linker and second version of the LAST interpreter. Luis continued to fine-tune his register allocator, and Rakesh became a second McCAT administrator to help with module integration.

In July Maryam, after working night and day, submitted her massive Masters thesis on points-to analysis, which she had managed to cut down to 200 pages.

The end of the summer was marked by the beginning of a concerted effort to adopt 'serious' benchmarks for McCAT, made possible by the source linker.

September, 1993. The wonderful summer quickly gave way to a mediocre autumn, as clouds and chilly temperatures arrived in the city. Chris finished his work on integrating the list schedulers in McCAT, and handed over his duties as McCAT administrator to Rakesh and Patrick. In addition, a new position of benchmark administrator was taken by Ana, to coordinate the McCAT benchmark suite.

The third compiler course using McCAT was started, with a focus on code improving transformations.

The genealogy description ends here, but is not the end of the story, as many talented people continue to work on the compiler. The author looks forward to hearing the continuing unfolding saga of an interesting and exciting project.

Acknowledgments

What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that **software is hard**. *From now on I shall have significantly greater respect for every successful software tool that I encounter. During the past decade I was surprised to learn that the writing of programs for TEXand Metafont proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks.*

–Donald Knuth, Keynote address to 11th World Computer Congress (IFIP Congress 1989).

My guideline in the morass of estimating complexity is that compilers are three times as bad as normal batch application programs....

–Frederick P. Brooks, Jr., "The Mythical Man Month: Essays on Software Engineering".

The McCAT compiler is, to say the least, a complex software project, and its success is a testament not only to the hard work of the students involved, but also to Laurie Hendren's management skills. Laurie has provided a sharp focus to the project, and nurtured its growth from a collection of course assignments to an exciting project with which I am proud to have been involved. I am also deeply appreciative of her keen insight into problems and her tremendously clarifying abstractions, which have made this thesis significantly less difficult than it could easily have been.

I would also like to thank one of the 'originals', Bhama Sridharan, for the countless times she has helped both me and everyone else in the lab with SIMPLE, and especially for enlightening us on the many mysteries of the GCC front-end. Luis Lozano and Sandro Muzzacato's excellent work on the register allocator and SPARC code generator was also invaluable in helping me produce working programs, and Patrick Betremieux's LAST interpreter was a tool from the gods.

I'm also very appreciative of: Kristín Völundardóttir, Jill Ferrier and Siobhán Phelan for suffering me as a weekend roommate for several months; Helga Hermannsdóttir & Ian McAdam for their friendship and for keeping me from going insane; Rakesh Ghiya for many enlightening conversations; the incessant traveler, Ana Erosa, for enduring the constant teasing from Rakesh, Luis and myself; and Cécile Moura, for showing me my limits and what happens when they are exceeded.

I would also like to thank my co-workers Pierre Paulin, Francis Langlois, Cliff Liem, Trevor May and Shailesh Sutarwala at Bell-Northern Research for their patience and understanding while I was finishing this thesis, et merci à Isabelle Pot et Malte von Rüden pour la traduction.

Contents

List of Tables

List of Figures

Chapter 1

Introduction & Motivation

> *Because performance of a computer will be significantly affected by the compiler, understanding compiler technology today is critical to designing and efficiently implementing an instruction set.*—Hennessy and Patterson [HP90].

Compiler technology is of vital importance to modern processors. High-performance processors are increasingly dependent on compilers to tweak, massage and contort input programs to take advantage of characteristics specific to their architectures. Non-blocking pipelined architectures, for example, such as the MIPS, SPARC, RS/6000, DEC Alpha [DS90, OHM+90, Dig92] and most other RISC (Reduced Instruction Set Computer [Pat85]) processors [HP90] expect the compiler to schedule instructions to maximize functional unit utilization; load/store architectures expect frequently-used values to be cached in registers [GH86]. These optimizations can be performed in hardware, but by shifting performance-improving manipulations from run-time to compile-time, scarce hardware resources can be devoted to other beneficial purposes, thus improving the overall performance of the architecture [SLH90]. For example, with instruction scheduling, the buffer for out-of-order execution architectures could be reduced and the saved hardware real-estate devoted to a larger register set, or a larger on-chip cache. Also, the overall complexity of the chip can be reduced, allowing for simpler, faster designs [DS90].

Needless to say, the increased responsibility of the compiler requires that architectural features, such as pipelined functional units and branch-delay slots, be exposed to the compiler. Representing these features is the domain of the compiler's intermediate representation (IR).[1] The IR is a useful mechanism to abstractly represent the hardware, while representing some important characteristics (in terms of performance) of the underlying target processor. The user's program, written in some high-level language such as C, C++ or FORTRAN, is translated into the IR. The IR then becomes an interface between the

[1] Sometimes referred to as an intermediate language (IL) or intermediate code (IC).

code-generation phase of the compiler (where actual assembly code instructions are generated), and the analysis and transformation phases. Depending on the compiler, the IR can be either very abstract (*eg* model a simple machine), very processor specific, or something in between. The more specific the IR, the greater the information available to expose fine-grain parallelism to the analysis and transformation phases. However, the compiler will be more difficult to retarget to a different architecture: the less abstract an IR is, the harder the compiler is to retarget as more of the analysis and transformation phases rely on processor-specific features.

This thesis presents an intermediate representation used in the McGill Compiler Architecture Testbed (McCAT) C compiler. This IR, called Last(Low-level Abstract Syntax Tree), forms part of the backend developed as the focus of this thesis. Last strives to reveal enough architectural details to the analysis and transformations phases, while retaining some abstraction to simplify retargeting of the compiler to a collection of RISC architectures.

In addition, Last is designed as part of a family of intermediate representations [HDE+92] (Section 3.1), of which Last forms the the lowest level. The other, higher IRs facilitate high-level analyses and transformations, such as points-to analysis,[2] array dependency analysis, loop transformations, and inlining. Last is specially designed to utilize the data-flow information generated from these previous analyses phases for low-level transformation *ie* support a paradigm of *pervasive flow-information.*

1.1 Thesis Contributions

While several existing IRs can accomplish similar goals to Last, an experimental approach of making Last a hierarchical, structured IR, rather than unstructured, was taken. That is, instead of representing the flow-control of the program in terms of `goto` statements and labels, Last retains the notion of high-level control structures such as `for` and `while` loops, and supports nesting of these structures. The phrase 'structured IR' is used in this thesis to denote an intermediate representation which has the characteristic of representing nested control structures (*ie* hierarchical structures), without the use of goto statements.

A structured IR was chosen for Last in order to support structured analyses and transformations, a central paradigm of the McCAT compiler. A structured IR is conceptually cleaner to use, since the program is always represented at a high-level. A structured IR was also chosen so as to simplify implementation of the pervasive flow-information paradigm. By supporting high-level constructs such as `while` and `for` loops, it is very simple to relate flow-analysis information generated at the other high-level IRs.

In addition to describing Last, some example analyses and transformations are presented to illustrate the structured nature of the IR. And finally, the retargeting strategy of

[2] Comparable to alias analysis [ASU88] in other compilers.

McCAT using LAST is presented.

Specifically, the contributions of this thesis are itemized below.

- The design and implementation of LAST, a low-level tree-based retargetable intermediate representation that, while suitable for transformations such as register allocation and instruction scheduling, also retains the structured nature of programs, as well as supports the pervasive nature of data flow information in the compiler.
- The design and integration of a highly retargetable code generator (using the code-generator generator BURG[FHP92b]) for RISC machines.
- The implementation of various structured analyses on LAST, with a specific example of live variable analysis presented.
- The implementation of various code-improving transformations on LAST, including instruction scheduling and reducing the number of load and store instructions.
- Experimental results from a suite of benchmarks illustrating the benefits of various analyses and transformations performed on LAST.

1.2 Organization of Thesis

The rest of this thesis details the contributions listed above. The following chapter, Chapter 2, gives some background on intermediate representations, listing traditional, current and LAST-related IRs. Next, Chapter 3, is an overview of the McCAT compiler. The general framework of the compiler is presented, from the higher IRs to LAST to the code-generation phase. Following the overview is Chapter 4, which presents the individual nodes of LAST in detail, as well as the optimizations performed during the generation of LAST. Chapter 5 and Chapter 6 illustrate the structured nature of LAST by presenting some analyses and transformations, and highlight the simplicity and ease of using a structured IR such as LAST. Next is Chapter 7, which describes the code generation strategy used, and also how LAST simplifies retargeting of the compiler. Chapter 8 presents results of the transformations performed on LAST, with aid of both the LAST analyses and pervasive flow information gathered from previous analyses. Finally, Chapter 9 presents the conclusions of this thesis.

Chapter 2

Related Work

2.1 Motivation for Intermediate Representations

As mentioned in the introduction, intermediate representations (IRs), are used to represent the target architecture in an abstract form. In optimizing compilers the analysis and transformation phases are run on the IR, after which the transformed IR is used to generate assembly code for the target machine. However, intermediate representations are not an essential part of compilers; compilers have the option of generating assembly code (and sometimes machine code) directly, without the overhead of generating and manipulating an IR [FL88].

Such 'one-pass' compilers are designed to quickly generate assembly/machine code, often at the cost of the quality of the generated code, *ie* little effort is spent performing code-improving transformations. However, the focus of high-performance compilers is obviously to improve the speed of compiled programs, and intermediate representations are used so the compiler has something to actually analyze and manipulate.

In addition to facilitating analyses and transformations, intermediate representations also simplify software maintenance problems for compiler developers, since the complexity of the analyses and transformation phases requires a large investment in design and development of the compiler. Due to the nature of evolving architectures, compilers can expect to be frequently re-targeted. Compiler developers naturally wish to minimize the loss of their usually substantial code investment, and designing the compiler to use an abstract IR, common to many back-ends, is one method of minimizing the work and effort required to retarget a compiler. As Tannenbaum et al. note [TvSS82]:

> *[I]t is desirable to do as much optimization as possible on the intermediate code, because that optimizer can be written once and for all and used without change as a filter for subsequent front ends and back ends.*

An intermediate representation allows one to replace the code-generation part of the back-end when retargeting the compiler, and in some compilers, also the front-end [Sta92, CHKW86], so that different languages can be compiled reusing most of the existing compiler.

Intermediate representations are designed to represent abstract machines, but there remains the problem of how abstract a machine to model. An IR that closely models a specific architecture will allow more efficient code to be generated, as intricate machine details can be exposed to the analysis and transformation phases. However, the compiler then becomes more difficult to retarget as these same phases might need to be rewritten should the compiler be retargeted to a new, different target architecture.

On the other hand, if the IR is very abstract, then the compiler is more flexible, but fewer machine details are revealed and the code-generation part of the back-end must be quite sophisticated and optimize the IR itself in order to produce efficient code [CHKW86]. Such optimizing back-ends, called peep-hole optimizers, already exist for CISC (Complex Instruction Set Computers) machines. Peep-hole optimizers combine adjacent instructions to produce more complex but cheaper (in terms of execution time or resource utilization) instructions. This is an important optimization for CISC machines, and several retargetable peep-hole optimizers have already been developed [DF84, Kes84, BD88], but they are of lesser help to compilers for RISC architectures.

RISC architectures require optimizations such as instruction scheduling and removal of redundant loads, transformations which require extensive analyses over potentially large blocks of code. Research into retargetable instruction schedulers is in progress [Con93, CCDM93], but requires that the target machine be accurately modeled *ie* too abstract a model defeats the purpose. A balance must therefore be struck between a very abstract model and a too detailed a model in order to obtain a highly optimizing but retargetable compiler.

2.2 Overview of different IRs

There are a variety of intermediate representation forms that can be used in a compiler, but they fall into three main categories of tuples, linear forms, trees and directed acyclic graphs (DAGS).

2.2.1 Tuples

Tuples are a simple, straight forward representation that can have a variety of forms [FL88]. A tuple will represent a destination variable, operator(s) and operand(s). The most common form is known as 'three-address code', which means that each statement contains three variables: one destination and two operands (plus a single operator). It is also known as a

quadruple (three variables plus an operator). A multi-operand, complex instruction, might be decomposed into the three-address statements as shown in Figure 2.1.

```
                                    t1 = c * d
a = b + c * d -e        ⇒           t2 = b + t1
                                    a  = t2 - e
```

Figure 2.1: Three-address code example

As mentioned, there are different variations, some allowing several operators and some only two operands with no destination (a triple). A triple does not explicitly save intermediate values like quadruples, but instead refers to them by the number of the triple that created it as in Figure 2.2.

```
                                    (1): c * d
a = b + c * d -e        ⇒           (2): b + (1)
                                    (3): (2) - e
                                    (4): a = (3)
```

Figure 2.2: Two-address code example

Triples are more concise than quadruples, but are position dependent and so can create difficulties for transformations involving code motion.

Due to their potentially compact implementation, tuples were popular for compilers facing strict memory limitations, and allowed compilers to save the intermediate representations to files using this compact representation [Hor91]. Modern architectures allow more liberal use of their resources, so there are few compelling reasons for plain tuples. However, tuples, particularly three-address codes modified to hold flow-information, can be quite useful for modeling RISC machines, which are typically three-address architectures—a three-address IR can thus model a RISC architecture quite closely. However, three-address code IRs introduce complications through the addition of temporaries when modeling CISC architectures [BGM79], and so would be best suited for compilers that support RISC machines.

2.2.2 Linear Forms

There are two types of linear forms—prefix and postfix. They are two forms common in mathematics, useful for expressing parenthesis-free arithmetic operations. In prefix form, the operator precedes the operands, and in postfix form it succeeds them. Intermediate

values are implicitly saved on a stack, and so linear representations are well suited for stack-based architectures. The expression `a = b + c * d -e` is expressed in prefix as `=a-+b*cde` and postfix as `abcd*+e-=`.

Ganapathi and Fischer suggested an interesting variant of a linear form suitable for optimizing compilers—an attributed linear prefix form [GF84]. The prefix form allows analysis of variable-operand instructions, while minimizing the parsing of the IR itself. The 'attributed' part simply means associated flow information is stored with each linear prefix statement. The main advantage is that the IR can be mapped to an assembly instruction in one pass, whereas other IRs, such as trees, require multiple passes. However, this advantage is of less and less importance as other compiler phases, notably the analysis and transformation phases, become relatively more expensive, and the code-generation phase becomes relatively cheaper to run.

2.2.3 Trees & DAGS

The most general intermediate representations are based on parse trees [FL88]. A parse tree represents arithmetic operations with operands as leaves, and operators as interior (parent) nodes. Intermediate results would also correspond to parent nodes. Figure 2.3 illustrates a parse tree for the expression `a = b + c * d - e`.

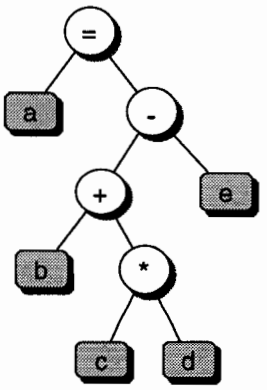


Figure 2.3: Parse tree example

Parse trees can directly represent hierarchy (*eg* nested structures such as nested `for` loops), as well as complex, multi-operator statements; other representations 'flatten' out the hierarchy and the complex instructions. For example, Figure 2.4 illustrates a parse tree consisting of an `if` statement with a sub-tree of instructions representing the condition expression, as well as a sub-tree for the body of the `if` statement. This body can contain any series of valid statements, such as other `if`, `while` or `for` constructs. Preceding and following the `if` statement are two arbitrary statements, stmt1 and stmt2.

In a non-hierarchical representation (right side of Figure 2.4), the condition and body statements are just part of a long list of instructions that also include stmt1 and stmt2: all sense of nesting or hierarchy has disappeared. Most low-level intermediate representations take this approach. By transforming high-level constructs to assembly-level statements, analyses and transformations are simplified since they need deal with only a subset of instructions. For example, all conditional constructs (`for`, `while`, `if`, `do while` and `switch`) can be transformed to a series of test and branch instructions.

However, this simplification also has a disadvantage, in that high-level information is lost, and must be regenerated. For example, to perform loop transformations (software pipelining [HP90] is included in such a transformation), loops must first be identified, requiring the calculation of dominators [ASU88, p. 602]. In some cases, the transformation to a low-level IR loses too much information that cannot be recovered. A structured IR retains all this information, and saves the cost of recalculation. In addition, dealing with structured IRs is conceptionally clearer, as it is much closer to the original program.

Trees also have the option of being structured or not. A structured representation, as explained before, does not represent `goto` statements. Since many source languages (including C) support `gotos`, many IRs are unstructured, as automatic structuring programs are complex, difficult to write (although possible, [EH94]), and can degrade performance of the resulting program. Structured analysis is, however, an elegant approach, and is straight forward to implement [ASU88].

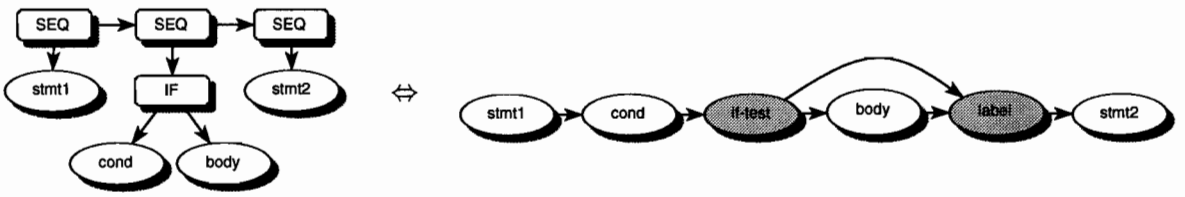


Figure 2.4: Hierarchy expressed in a parse tree

DAGS, or directed acyclic graphs, are variants of parse trees, where common parent and/or child nodes can be shared (Figure 2.5). DAGS can simplify some optimizations, such as common sub-expression elimination [ASU88], but some useful tools, such as the code-generator generator BURG[FHP92b], require trees and will not work with graphs.

The great power of parse-trees (and DAGS) is that all the information available at the source-level language is available in the representation. Parse trees, therefore, are an excellent IR when a compiler is performing source-to-source transformations. Parse trees are also good for generating code for CISC architectures, since complex instructions can (sometimes) map nicely onto subtrees in the parse tree [BGM79]. However, based on previous experience in building an optimizing compiler utilizing a parse-tree as an IR,

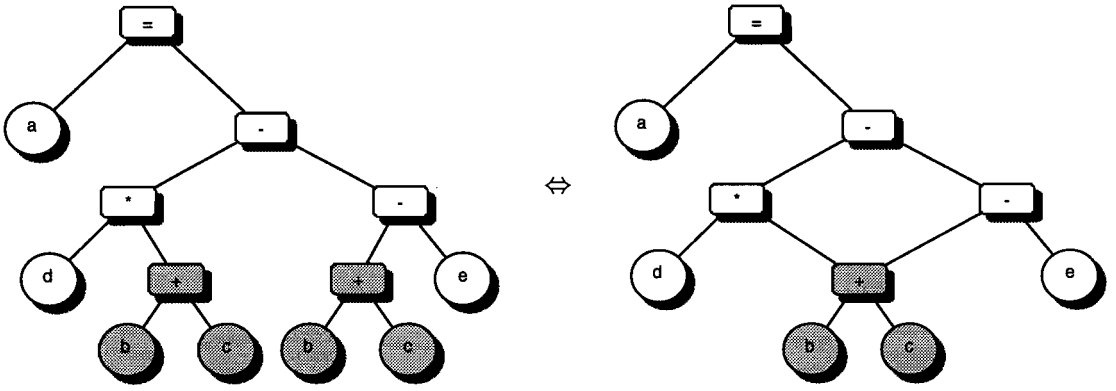


Figure 2.5: Tree and DAG IR example

the potential complexity of parse-trees, due to the hierarchy, can significantly complicate analyses and transformations. Trees and DAGS have the option of being structured, *ie* support flow-control without the use of `goto` statements and labels.

The McCAT compiler uses a simplified version of a parse-tree for one of its intermediate representations [Sri92, HDE+92], which provides the power of parse trees without the complexity (see Section 3.1).

2.2.4 Representing Flow of Control

Once the program has been parsed and an IR generated, the program's flow of control must be determined. In an unstructured IR, a control-flow graph (CFG) [ASU88] is generated to capture the possibly arbitrary changes in control flow. The IR on the right in Figure 2.4 shows two control-flow arcs from the test associated with an `if` statement. One goes to the beginning of the body of the `if` statement, and the other to a label, indicating the end of the body.

A structured IR, on the other hand, has flow-control represented explicitly, since each change in control is well defined (*ie* no `goto` statements). For example a `continue` will always transfer control to the inner-most enclosing loop.

The main advantage of a CFG is that it can handle `goto` statements. However, it must be generated, and the abstraction of high-level structures (like `while` loops) is lost, as all control structures are represented as low-level jumps to labels.

2.2.5 Comparison

Enumeration of the different types of IRs is, in some sense, superfluous, because the methods of implementation overlap (*eg* many would be combinations of pointers to structures [GF84])

so that there is sometimes very little that distinguishes the representations from one another. In addition, there are, of course, modified versions of each category which only blur the distinctions even more. The tuples, linear forms and trees/DAGS can all be converted into one another, although it may take some work. For example, a pre-traversal over a parse tree will generate a prefix linear IR, and a quadruple can be used to regenerate a tree. It is often the resulting abstract machine being modeled which decides the IR; LAST takes a three-address approach to model abstract RISC machines, in combination with a tree to provide a structured IR.

2.3 Specific Examples

The following subsections briefly describe some contemporary intermediate representations: U-code, RTL, and SUIF.

2.3.1 U-Code

U-code, or Universal PASCAL Code, is an extension of the PASCAL Code (P-code) intermediate language developed by Wirth [Wir71, NAJ+81] for PASCAL. U-code was extended to simplify global (intra-procedural) optimizing transformations for both PASCAL and FORTRAN programs [PS79], and was used to develop a retargetable, machine-independent global optimizer UOPT [Cho83]. The central idea of UOPT is to perform as many analyses and transformations as possible on the machine-independent IR in order to reduce re-implementation when retargeting the compiler to different architectures, an idea prevalent in many modern compilers.

U-code (and P-code) is an unstructured, three-address code IR that models an abstract stack-based architecture called the P-machine. The machine is composed of three stacks: one for runtime (to model parameter passing and memory allocation), one for address and integer operations, and a third for set and real type operations [KKM80]. U-code is thus quite retargetable, due to its highly abstract nature. It is not an effective representation for CISC-based architectures (due to its three-address approach), as a large number of extraneous temporaries are generated that take considerable effort to collapse [BGM79], an operation needed for effective instruction set selection.

However, U-code can be quite effective for RISC architectures: it is currently used in the MIPS compiler [Cho88, CHKW86], although by itself the IR is too abstract to generate good assembly code. The code-generation modules are extremely sophisticated, and must perform many complicated transformations, including instruction scheduling and dead code elimination [CHKW86]. In addition, the MIPS U-code representation was modified to represent some architecture specific features (*eg* function calls) [CHKW86]. The MIPS version of U-code is used primarily to perform machine-independent optimizations such

as copy propagation, function inlining, common sub-expression elimination and strength reduction [ASU88].

2.3.2 RTL

RTL, or Register Transfer Language, is the intermediate representation used in the popular GNU C compiler [Sta92, pp. 127-166], RTL is a variation of Jack Davidson's Register Transfer Lists [DF86]. There are many flavors of RTL, but the GNU version is perhaps the most widespread. The goal of RTL, like U-code, is to provide a machine-independent medium on which to perform analyses and transformations, and also be easily retargetable. However, RTL takes the opposite tack to U-code. Whereas U-code accomplishes portability by modeling a single, very abstract machine, RTL is configured to provide very machine-specific representations for different architectures. RTL provides actions that are at a lower level than corresponding assembly level operations. For example, RTL uses separate nodes to indicate register reads and register writes, whereas an assembly instruction will read the operands and write a result as an atomic operation *eg* the assembly instruction `add r1,r2,r3` reads registers `r2` and `r3`, and writes the result in `r1`. In addition, several RTL nodes are used to represent architecture-specific details, such as the size of character variables when performing a type-casting operation. Essentially, different sequences of RTL nodes are generated for different classes of architectures.

While some ideas from RTL were used in the design of LAST(see Section 3), there are two fundamental differences. First, LAST takes a higher-level approach, in that since its target architecture is limited to RISC, there are certain basic architectural assumptions that are made to simplify the design, whereas RTL must be flexible enough to handle a wider variety of architectures, which translates to a lower-level (and more verbose) approach than LAST. For example, LAST need not support CISC instruction sets, and so a simple, straight forward representation of operators suffices. GNU's RTL however, must support complex[1] instructions, which leads to a plethora of nodes flexible enough to handle the many possible types of CISC instructions.

The second difference to LAST is the unstructured nature of RTL. For instance, the operands to branch instructions are located by convention: they are the nodes immediately preceeding a conditional branch. This low-level, unstructured approach makes some low-level transformations, such as instruction scheduling, conceptually easy, but high-level analyses and transformations are more difficult.

2.3.3 SUIF

SUIF, or the Stanford University Intermediate Format [TWL+91], is a hierarchical intermediate representation that was designed in order to marry high and low-level analyses and

[1] And sometimes idiosyncratic!

transformations. Although SUIF is a hierarchical representation, it is unstructured, that is, SUIF supports arbitrary changes of flow control through `goto` statements. SUIF actually is polymorphic—there is a low-SUIF and a high-SUIF. The creation of two different forms was motivated by the two different types of parallelism available in programs: coarse- and fine-grain parallelism. Parallelizing (coarse-grain) transformations require a high-level view of the program, whereas scalar (fine-grain) optimizations prefer a low-level view. In many compilers this leads to two incompatible IRs, and thus duplicated analyses and transformation phases, since analysis information from the high-level IR is often unavailable to the low-level IR. This prompted the development of SUIF, which is a low-level IR with a superset of nodes that represent high-level structures and objects such as `for` loops and array references. The SUIF compiler first generates high-SUIF, performs coarse-grain transformations, and then transforms high-SUIF into low-SUIF, for scalar optimizations.

Low-SUIF represents the program in much the same way other unstructured IRs do: as a long linked list of assembly-level instructions (in three-address form with associated flow-information). However, for the parallelizing transformations, high-SUIF is used.

The main advantage of this multi-faceted approach is mainly in the software maintenance aspect of building the compiler. The common IR allows transformations to be implemented only once in the compiler (although they may be run several times after high-level transformations), saving time and effort.

Chapter 3

Overview

3.1 An Overview of the McCAT C Compiler

This section gives an overview of the McCAT C compiler, including the component intermediate representations, of which LAST forms the lowest level IR. McCAT is being developed by the Advanced Compilers, Architectures and Parallel Systems (ACAPS) research group at McGill as an optimizing and parallelizing C compiler, well suited to accurate points-to[1] [Ema93, EGH94] and dependency analysis [Wol82]. McCAT accomplishes this by utilizing a family of intermediate representations, which support pervasive flow information, *ie* flow information gathered from analysing one IR is available to lower-level intermediate representations for their transformations (see Figure 3.1).

The compiler uses a modified GCC[2] front-end for parsing the source files and generating a high-level *abstract syntax tree* (AST) [ASU88, p. 49], dubbed FIRST. This intermediate representation is then simplified into a similar AST called SIMPLE, upon which various high-level analyses and transformations are performed (for example, points-to analysis and loop unrolling).

SIMPLE is designed around the points-to analysis algorithm [HDE+92, EGH94], which requires a high-level view of the program, *ie* abstractions such as arrays, structures and pointer types are retained. This high-level information is required for alias analysis, but is inappropriate for low-level analyses and optimizing transformations such as register allocation or instruction scheduling: the architectural details required for these low-level optimizations are hidden in SIMPLE. LAST is designed to expose these details, yet at the same time retain some of the high-level features of SIMPLE such as abstract control flow structures (*eg* a `for` statement) so as to maintain a structured representation.

The McCAT compiler can be described as having four phases, with various analyses and transformations working on the two dominant IRs: SIMPLE and LAST. The first phase

[1] Analogous to alias analysis in other compilers.

[2] GNU's 1.37.1 version of GCC, to be specific.

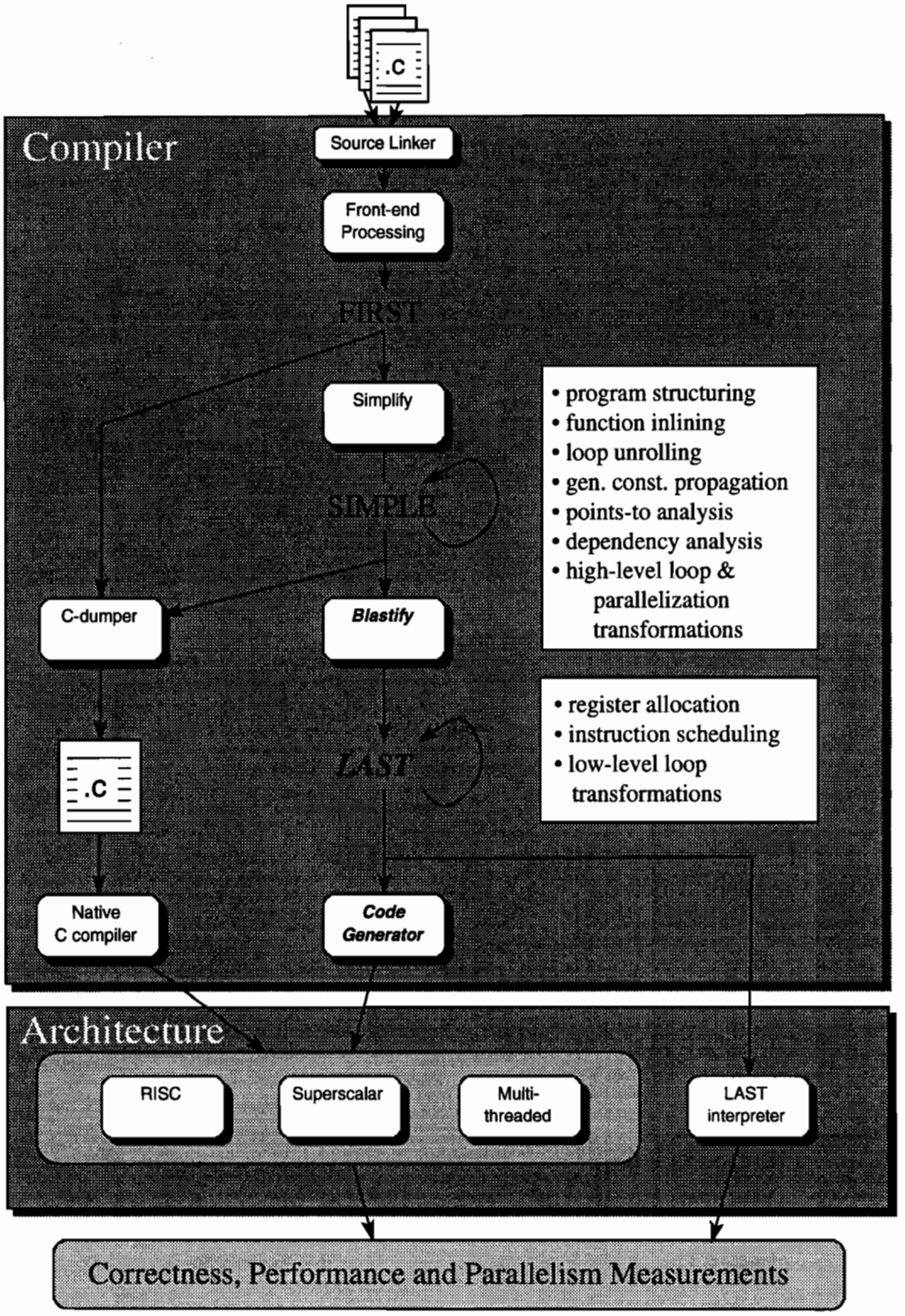


Figure 3.1: Overview of McCAT

is the creation of FIRST and the second its transformation to SIMPLE. The third phase transforms SIMPLE to LAST, and the fourth, final phase generates assembly code from LAST. Figure 3.1 illustrates the overall process.

3.1.1 Front-end Processing and Simplification Phases

The McCAT compiler parses C source files and produces an AST.[3] In McCAT, this AST is called FIRST, and is the first phase of compilation. This IR is immediately transformed, or *simplified* into SIMPLE(the second phase), upon which the first set of analyses is performed. As the name would suggest, the simplify stage takes C, in all its gory glory, and simplifies it to a grammar corresponding to that in Appendix A. Typical simplifications include compiling complex statements into a series of basic statements, simplifying all conditional expressions in `if` and `while` statements to simple expressions with no side-effects, simplifying procedure arguments to either constants or variable names, and moving variable initializations from declarations to statements in the body of the appropriate procedure. Figure 3.2 captures the transformation of a complex arithmetic operation to a series of basic statements; the figure illustrates how a FIRST tree is transformed into a sequence of two SIMPLE trees. High-level abstractions such as array and structure references remain, but are simplified. Bhama Sridharan provides an excellent description of SIMPLE in her masters thesis [Sri92].

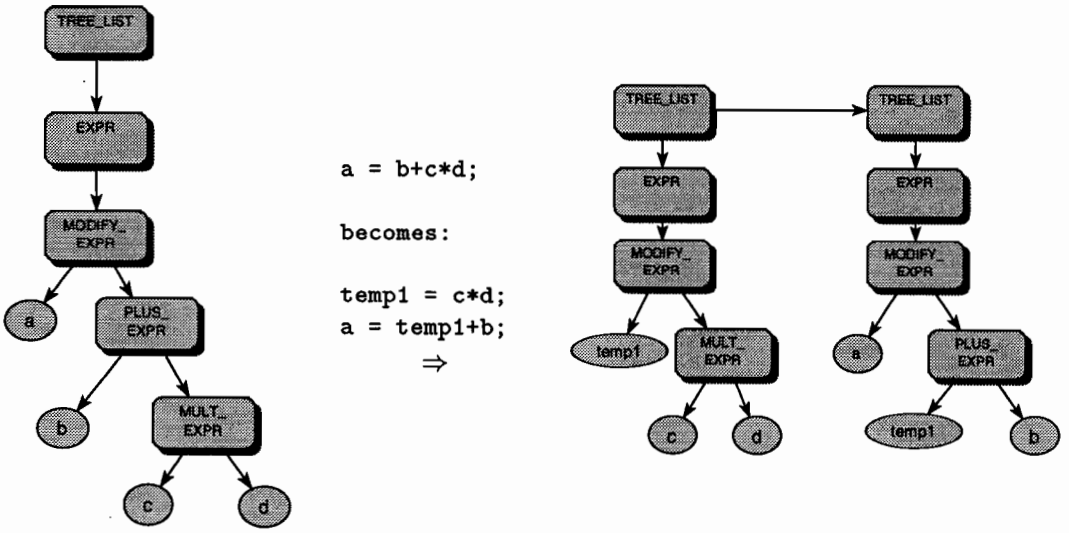


Figure 3.2: FIRST to SIMPLE conversion

[3] McCAT also has the ability to parse multi-file programs and produce a combined AST.

3.1.2 The Blastify Phase

The blastify phase generates LAST from SIMPLE. Whereas SIMPLE hides the memory hierarchy, LAST exposes it by representing register loads and memory stores, as well as representing array and structure references as a series of arithmetic operations. In other words, the LAST IR is very close to assembly language: in most cases there is a one-to-one correspondence between LAST statements and assembly language instructions. At the same time, however, high-level constructs such as `while` loops are still represented, in order to maintain the structured representation and simplify access to flow-information stored in SIMPLE(explained in Section 3.2.1). Figure 3.3 illustrates the translation of the SIMPLE tree in Figure 3.2 to its LAST counterpart, and Figure 3.4 shows the C code (on the left hand side) for SIMPLE, and the corresponding LAST pseudo assembly code (on the right hand side). Essentially, variables `c` and `d` are loaded from memory into registers, multiplied, and the result placed in a register `temp1`. The variable `b` is then loaded, added to `temp1`, moved into a register and then stored in the memory location reserved for `a`.

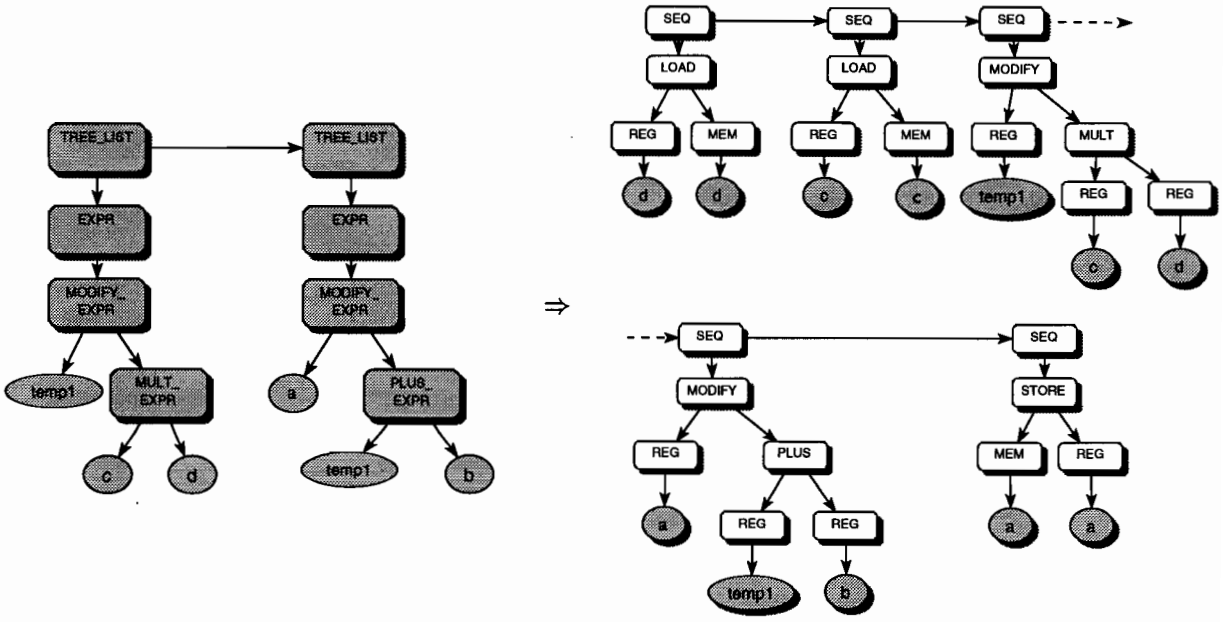


Figure 3.3: SIMPLE to LAST Transformation

3.1.3 Code Generation Phase

The final and simplest of phases in McCAT is code generation. Since the focus of research for McCAT is on IR transformations, rather than generating object code, the compiler

```
temp1 = c * d;
a = temp1 + b;
```

⇒

```
REG(d)(0) <-LOAD(Int)- MEM(d)
REG(c)(0) <-LOAD(Int)- MEM(c)
REG(temp1) := REG(c) * REG(d)
REG(b)(0) <-LOAD(Int)- MEM(b)
REG(a) := REG(temp1) + REG(b)
MEM(a)(0) <-STORE(Int)-< REG(a)
```

Figure 3.4: Code for SIMPLE to LAST Transformation

produces assembly code, and uses existing simulators/assemblers for running the input programs.

The design philosophy of LAST is to push as much complexity into the IR, to use transformations on LAST to perform optimizations such as instruction scheduling and to make the code generation (the actual printing of assembly instruction to a file) as trivial as possible. This is accomplished by a code-generator generator called BURG[FHP92b], which allows simple templates of assembly code to be written for various LAST constructs. BURG constructs a tree traversal routine that traverses LAST, and when it finds a group of LAST nodes that match a particular pattern, the corresponding code template is printed. No complicated analyses or transformations are performed within the code generator, and so retargeting is greatly simplified as most the work consists of only rewriting simple templates for each new machine targeted. Figure 3.5 gives an example of the overall strategy. Blastify generates a LAST subtree corresponding to a division statement `x = y / z`, which the BURG-generated code generator matches with one of its patterns. In the example there are only two patterns to match: addition and division. Once the match is made, the appropriate code template is used to generate assembly code. Templates for two machines (machine A and machine B) are shown, with machine A's template being used (so `div ra,rb,rc` is generated).

3.2 Overview of LAST

McCAT was constructed utilizing the GNU C compiler. The parsing phase was kept, but its entire back-end, including the analysis, transformation and code generation phases, was removed. The GCC back-end utilizes RTL [Sta92], but the code was judged too complicated to modify. One major contributor to the GNU C and GNU C++ compilers considers the "common back-end of these compilers very difficult to comprehend and/or modify" [Gui94]. As a result, SIMPLE was developed and serves its function well as a structured IR for high-level analysis.

However, SIMPLE is too abstract a representation for low-level transformations such as instruction scheduling and register allocation—high-level constructs such as arrays and

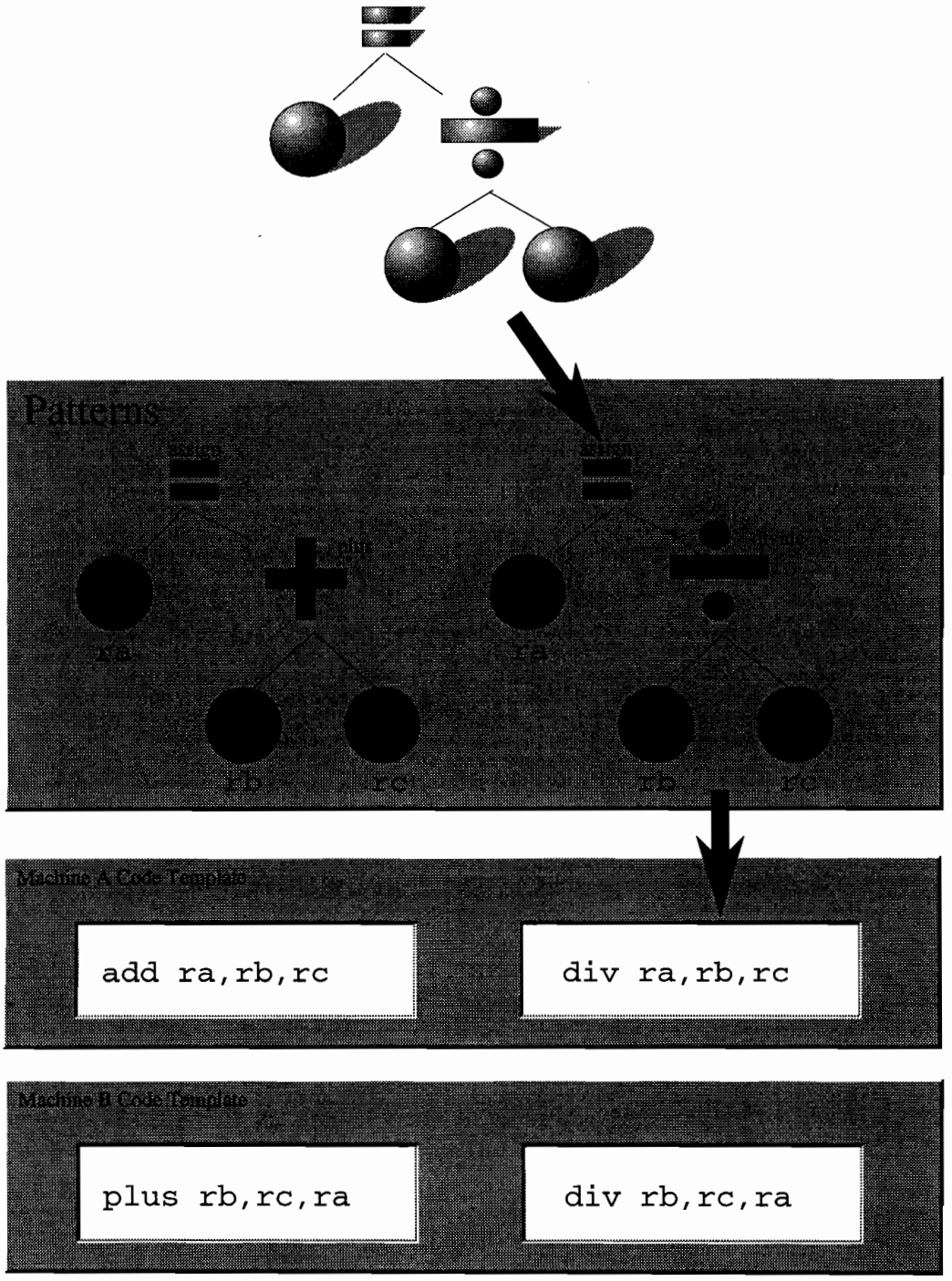


Figure 3.5: Code templates used by BURG to simplify retargeting

structures still exist. As a result, another low-level IR was required, and so the mandate for LAST was born.

3.2.1 Design Mandate for LAST

Based on the desire to utilize high-level flow information (gathered at the SIMPLE level) to aggressively exploit fine-grain parallelism in target machine architectures, the following six criteria were developed to guide the design and implementation of LAST.

Support for structured analysis: A structured, compositional representation, where control flow is regular and explicit (*ie* no `goto` statements) simplifies analysis tremendously, as control structures can be analyzed compositionally. For instance, if analyzing a `while` loop, only the conditional expression and body of the loop under analysis are inspected; surrounding control constructs are disregarded.

A compositional representation enables program analysis to be abstracted to a simple model, and thus implemented in a straight forward fashion, and allows for the use of automated tools. Also, by maintaining the program structure, it becomes easy to find and transform groups of loop nests. Our register allocation algorithm, based on hierarchical cyclic interval graphs [HGAM92], uses this compositional property.

The critic might point out that `goto` statements can and do appear in popular programs, particularly in interpreters and automatically-generated code. An unstructured program must therefore be converted into an equivalent structured program, an achievable target [WO75, Bak77, Amm92]. Currently, the McCAT compiler has a restructuring module that converts all programs with `goto` statements into equivalent structured programs [EH94].[4]

Support for pervasive analysis: LAST is only one in a family of intermediate representations. Each IR has its own advantages. For example, SIMPLE, the high-level representation, is suitable for points-to analysis. Such high-level information is crucial to determining available fine-grain parallelism, and is needed at the lower-level representation LAST. Therefore, propagation of information from the higher IRs to LAST was an essential design criterion.

Support for load/store machines: Reduced instruction set computer (RISC) architectures are perceived as having a significant performance advantage over complex instruction set computer (CISC) architectures [DS90, HP90]. One of the distinctive features of a RISC architecture is that it is a load/store architecture, that is, all memory references are through explicit stores(loads) to(from) memory, and arithmetic operations take only register operands.

[4] With the exception of `setjmp` and `longjmp`, which are currently unsupported.

Considering the gravitation of high-performance computer systems towards RISC architectures, LAST was required to support them, and thus load/store architectures.

Expose opportunities for transformations: Obviously, since McCAT is an optimizing compiler, it should aggressively seek to improve the efficiency of generated code. As a corollary, LAST should expose all opportunities for code and performance-improving transformations, such as exposing the use of all registers to the register allocator, filling branch delay slots, removing induction variables (plus strength reduction), and exposing a function's prologue and epilogue to the instruction scheduler. This means that LAST must be able to represent individual assembly level instructions.

Support simplistic code generators: Since McCAT is a research and pedagogical tool, it has a mandate to investigate a wide variety of real and experimental architectures. McCAT therefore needs to be highly retargetable, and so should require minimal intelligence of its code generation module. That is, as much complexity as possible should be embedded in the non-machine specific intermediate representation, and the IR should be generic enough for load/store architectures so that the compiler is easily retargeted between such machines.

Support for high-level tools: In order to maximize productivity on interesting research ideas by minimizing time spent on laborious and repetitive tasks, as many facets of the compiler as possible should be relegated to automated tools. For code generation, McCAT employs the code-generator generator BURG(see Section 7.2) to ease the task of retargeting McCAT to different machines.

3.2.2 Design Influences on LAST

There were three main influences in the design of LAST. First, LAST's design was heavily influenced by experience gained in analysing the compositional form of SIMPLE. This structured form simplifies analysis by guaranteeing a strict flow of control, so that language constructions can be analysed and transformed compositionally. In addition, the compositional approach allows easy identification and transformation of loop nests, which is useful for the McCAT cyclic interval graph register allocator [HGAM92] and the dependence testing framework [JH94, Jus94].

The second influence was through the author's experience in an introductory compiler course, which demonstrated the usefulness of the code-generator generator BURG, despite its limitations [AH91]. One of these limitations is that BURG can traverse only binary trees, even though some AST constructs are more naturally represented as n-ary trees, where n is greater than two (such as `for` loops or `if` statements). However, it was later discovered that this binary representation made for a fast traversal mechanism for other analysis phases, due to the regular structure of the AST.

The third influence was a paper by a compiler group from the University of Illinois. A article by Johnson, McConnell and Lake from University of Illinois on RTL [JML91] proved quite useful in identifying what information should be represented at the LAST level, such as labels for branch chain elimination.

3.2.3 Abstract Machine Model

Selecting the appropriate abstract model for an intermediate representation is a major design decision. As discussed previously, there is a choice, and in fact a tradeoff, between retargetability and generating highly efficient code. As part of a high-performance compiler, LAST must obviously generate high quality assembly code. To simplify retargetability, it was decided to limit McCAT's architectural targets to RISC machines.[5]

RISC architectures have evolved, after extensive study [Pat85, HP90], from CISC machines which were developed in a more ad hoc approach [HP90]. That all the successful modern high-performance workstations use RISC processors, such the MIPS, DEC Alpha, RS/6000, PA-RISC and SPARC chips, is a testament to this research. So, while the commercial market place may demand compiler support for CISC chips,[6] for a research oriented, high-performance compiler, limiting potential targets to RISC machines is a reasonable decision.

This limited focus has simplified the design of LAST by allowing it to model an abstract RISC machine with features that are expected to be on most, if not all, compiler targets for McCAT. By exposing these features, machine-independent analyses and transformations can be written and reused for all the targets. Since the features are general across the targets, retargeting is dramatically simplified. Of course, an abstract model cannot capture all the features found in each specific architecture, but in practice these additional features have been relatively minor and easily accommodated within the LAST framework.

The abstract RISC architecture that LAST expects consists of four main characteristics, as described below.

Load/Store architecture: All memory accesses are through explicit references to memory, rather than implicit references as in the Motorola 68000 or VAX architectures [HP90].

In load/store architectures, loads and stores are considered expensive, 'to-be-avoided' operations. On some architectures load latencies can take dozens of cycles, leaving the processor under-utilized [CKP91]. As a result, the blastify translation (and subsequent transformations) attempt to keep as many variables as possible in registers *ie* load and store instructions are reduced where ever possible.

[5] To be precise, McCAT is oriented towards scalar, superscalar and multi-threaded RISC machines.
[6] Such as the Intel 80x86 series [DS90].

General purpose register set: Explicit manipulation of registers is represented at the LAST level. It is assumed that the registers are relatively general purpose *ie* that the integer register set can also hold addresses. It is also implicitly assumed that a RISC architecture will have a large[7] number of general purpose registers, although, strictly speaking, this is more an issue for the register allocator than LAST. The assumption is implicit since *blastify* tries to keep as many variables in registers as possible, and so assumes that register pressure is not so severe as to cause the register allocator to generate an excessive number of register spills and reloads.

Reduced instruction set: In keeping with the RISC strategy, LAST expects a relatively simple instruction set. A complex high-level operation, such as an array reference, is translated to a sequence of simpler low-level operations.

In addition, it is assumed that most instructions will have the same latency (*eg* one clock cycle), so little effort need be expended on instruction set selection. For those instructions that have longer latencies, a non-blocking pipelined architecture is assumed.

Pipelined architecture: Long latency operations, such as a floating-point multiplication, are assumed to be non-blocking, so delay slots can be filled by other instructions. In addition, LAST sub-trees are designed to simplify rearrangement due to instruction scheduling.

3.2.4 Retargeting LAST: A Configurable IR

Despite the architectural features that are common to RISC machines, there will always be variations and different approaches to these RISC features. For example, the MIPS R2000 architecture requires the operands of an integer multiplication operation be contained in floating point registers, whereas the SPARC architecture does not [DS90]. An integer multiplication on the MIPS therefore requires a move of operands from integer registers to floating point, and a move of the result from a floating point register to an integer one in addition to the multiplication instruction, whereas the SPARC has no need of these register moves. A generic approach hides the intricacies of a MIPS integer multiplication, whereas a performance hungry approach exposes it.

Again, there is a tradeoff between ease of retargeting the compiler (and the not insubstantial reuse of code) and generating efficient code. LAST attempts to strike a balance between the two extremes by supporting different configurations. LAST is configurable to various architectures, similar to how GCC's RTL works [Sta92]. The configurations available are grouped in several classes, so that machines in similar classes will use identical IR's, even if the actual assembly code produced is slightly different. With this approach,

[7] As compared to traditional CISC architectures with typically eight registers.

the analysis and transformation phases need not be re-implemented. There is, however, a penalty to be paid in terms of generating different forms of LAST for different architectures, but the cost is minimized by supporting only classes of architectures rather than specific machines. These classes can be thought of as refinements of various abstract RISC machines that provide greater description of a target architecture, while still maintaining a degree of abstraction.

The following architecture classes are supported:

- Explicit(implicit) condition codes. Condition codes for various operations are (not) held in registers.
- Multiplication/division of integer values in integer (floating-point) registers. Some architectures, such as MIPS, must move integer values into floating-point registers to perform multiplication or division, and then must move them back to integer registers. To gain maximum benefit from the register allocator and instruction scheduler, the extra instructions required for the register moves are explicitly represented in the IR. Conversely, if the architecture does not have this requirement, such as SPARC, then these unnecessary moves are not generated during the blastify process.
- Register windows versus the traditional stack paradigm. Architectures with register windows such as the SPARC require a different register allocation strategy than the traditional stack-oriented architectures. Again, to maximize benefits from instruction scheduling, the pushing and popping of arguments onto and off of the stack during a procedure call must be exposed. The register windows paradigm can also be used for architectures that do not have register windows, in order to easily support the passing of parameters via registers, an elegant approach to minimizing procedure call overhead [Cho88, Wal88].
- Architectural support for structure copies. Some architectures include instructions to perform block copies of memory. In some machines the assembler will support such instructions, even if the actual architecture does not. LAST considers the assembler to be the target architecture, with the assumption that the assembler can generate the equivalent instructions at least as efficiently as LAST. However, for architectures (assemblers) that do not provide such a copy operation, it is important to expose the multitude of operations required.
- Architectural support for exclusive or operators. For architectures that do not support exclusive or operators, equivalent instructions using simpler operators are generated.
- Architectural support for negation. As with exclusive or operators, simpler instructions are generated in lieu of this support.

- Specific type conversion routines. Various architectures have their own approaches for performing type conversions *eg* converting a variable of type integer to double. Some architectures require integer doubles to be in floating point registers, others allow them to remain in integer registers. In order to expose these instructions to the instruction scheduler and register allocator, they are all generated, specific to each architecture. For the SPARC and MIPS machines, the differences are trivial, but for the RS/6000 it is more complex.

Currently the compiler generates assembly code for three architectures: SPARC, DLX (a simplified version of MIPS' R2000 assembly code) and RS/6000, in addition to pseudo assembly code, an easier-to-read approximation of RISC assembly code. It is used primarily for pedagogical purposes, but also for the LAST interpreter to display the interpreted LAST code. Figure 3.6 shows the corresponding pseudo assembly generated for a trivial C program.

```
void main(void){
int a;
    a = 1;
    foo(a);
}
void foo(int c){
int a,b;
    a = c * 4;
    b = a;
}
```

⇒

```
;==============================
;   Function body for "main"
;==============================

  <<Save Registers>>
  REG(a) :=  1
  Function Call to foo
  delay slot: nop
  Arguments:
      Parameter: REG(a)

  <<Restore Registers>>
  end of sequence

;==============================
;   Function body for "foo"
;==============================

  <<Save Registers>>
  REG(c)(0) <-LOAD(Int)- MEM(c)
  REG(a) := REG(c) <<  2
  REG(b) := REG(a)
  <<Restore Registers>>
  end of sequence
```

Figure 3.6: Example of Pseudo assembly code

The `<<Save Registers>>` and `<<Restore Registers>>` represent the saving and restoring of registers for architectures without register windows, and are explained in Section 4.2.3.

Otherwise the example is straight forward; the only difference is the optimization of replacing a multiplication by 4 (`a = c * 4`) with a bitwise left shift of two bits (`REG(a) := REG(c) << 2`). Notice that in the body of `foo`, the value of `c` is loaded from memory (a stack parameter passing paradigm is assumed) and placed in a register, denoted by `MEM` and `REG` respectively. Once the value of `c * 4` is saved in the register for `a`, it is not stored, but simply copied to the register for `b`. The register holding this value is not stored, and is thrown away since it is subsequently unused.[8]

3.2.5 Overview of Transformations and Analyses on LAST

Figure 3.7 shows the analyses and transformations currently performed on LAST, starting with SIMPLE (the shaded node). There are eight phases before assembly code is finally emitted, starting with the register use phase. This is an analysis of SIMPLE, whose information is used during the blastify process to determine what variables are in registers. It is optional, and is used in conjunction with the naive spilling transformation only when the compiler is being conservative with register-memory consistency. Register-memory consistency is the problem of ensuring that when a value resides in both memory and a register, that the values are both consistent.

The next phase is blastify itself, the generation of LAST from SIMPLE. During this process, several code-improving transformations are performed as LAST is generated, including alias substitution, multiplication replacement [Ber86] and folding of constant array indices. Next, one of either the naive or improved spilling transformations is performed, which determines what is optimistically kept in registers (given an infinite number of registers), and what must be stored to memory. The naive algorithm loads (stores) globals and aliased variables on every use (definition), and stores local variables at the end of basic blocks. The improved spilling algorithm attempts to minimize reloads and unnecessary stores.

After spilling, the McCAT list scheduler is invoked, and instruction scheduling performed. After scheduling, live variable analysis is performed, generating information for the next phase, register allocation [LJ92]. After allocation, offsets for parameters and local variables are calculated, in time for the eighth phase—assembly code generation. There is also the optional phase of interpreting LAST[Bet94], occurring after code generation.

3.3 Comparison of LAST with SIMPLE

Superficially, LAST is quite similar to SIMPLE; both LAST and SIMPLE express most C statements in three-address form. However, LAST is otherwise significantly different from SIMPLE.

[8] In future generations of the compiler, a dead-code elimination phase will remove such useless instructions.

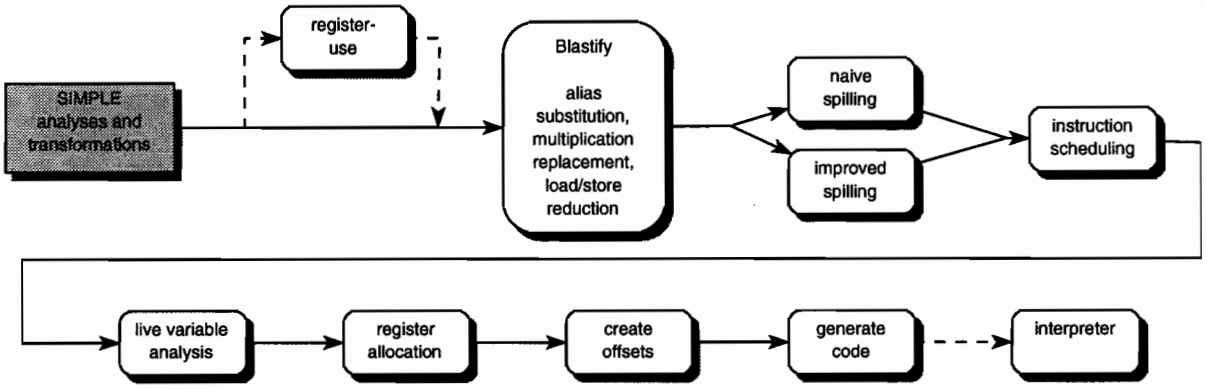


Figure 3.7: Analyses and transformations performed on LAST

1. LAST statements, including complex statements such as array and structure references, are represented in terms of three-address code *ie* address calculations are exposed.

2. LAST statements are modeled after a load/store architecture.

3. LAST nodes corresponding to variables are unshared *ie* have only one parent, whereas in SIMPLE the same node is re-used to represent a specific variable.

4. LAST maintains a parent/child relationship with SIMPLE—it is not a replacement, but rather an augmentation of SIMPLE.

3.3.1 Three-address Code

All expressions and variable references, including references to structures and arrays, are represented in LAST in terms of three-address code. For example, the blastify transformation replaces a SIMPLE sub-tree representing an array with a series of LAST sub-trees that represent the various array offset calculations required to access the array value.

Figure 3.8 gives an example of an array reference, including a load of the base address.[9] Note that SIMPLE temporary variables, such as `_t1`, are analogous to registers and are therefore never loaded,[10] and are always defined before being used.

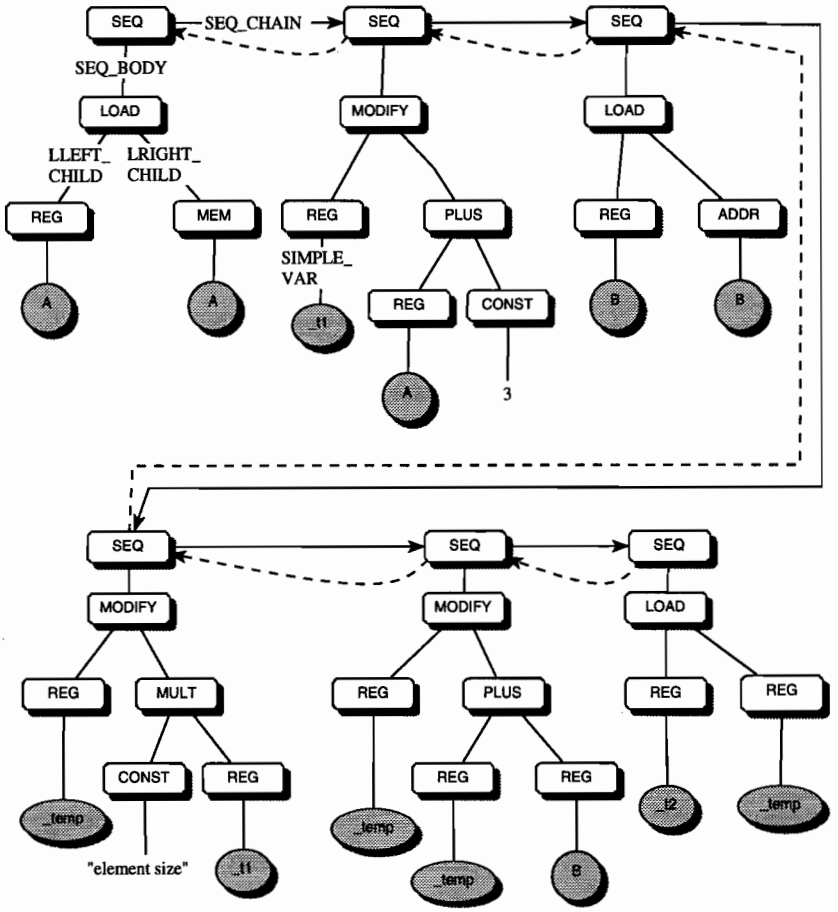
The first subtree in the figure represents the load of the value of variable `A` from memory, denoted `MEM(A)`, into a symbolic register *associated* with `A`, denoted `REG(A)`. This is the notation used to differentiate the memory hierarchy.

The second subtree represents the addition of `A + 3`. The value is placed in `_t1`, a temporary register-only variable. Next, the address of `B` is loaded into a register associated with `B`. Variable `_t1` is then multiplied by the element size of the array, and the result

[9] For the purposes of the example, all the variables are assumed to reside in memory.

[10] Unless of course the register allocator selects them for spilling.

stored in another temporary variable `_temp`, which is then added to the value of `B`. These two operations constitute the calculation of the array reference. The final subtree loads the value found at the address that has been calculated placed in the register for variable `_temp`.



_t1 = A+ 3;
_t2 = B[_t1]

Figure 3.8: LAST representation of an array reference

Structures, like arrays, are represented differently than in SIMPLE. Except for bit-fields, structure field references indicate a load or store to an offset from a base address. LAST represents such a reference as a load or store, with the appropriate offset kept in the *offset* field (accessed via the *LS_OFFSET* macro) of the load and store nodes (Section 4.2.2) *ie* for each field reference, there is a corresponding load or store LAST node, plus the initial

load of the base address.

Representing bit-field references, however, involves additional representation. Since the target architecture is usually byte addressable, and no finer, accessing individual bits requires bit masking. First the appropriate byte/halfword/word is loaded into memory, and then the bits of interest are isolated via a bit mask. If the left hand side of an assignment is a bit field, then any unaffected bits must be saved, and also the right-hand side be appropriately masked as well. Figure 3.9 gives an example bit-field reference. First, `a` is assigned 1. In preparation for assignment to `str.field`, the bit is isolated by first shifting the contents five bits left, and then masked. Meanwhile, the address of `field`, containing the bit, is loaded into a register. This value is then bit masked to preserve the surrounding bits, including the four bytes used to represent the `x` field. The result of the mask is or'ed with the isolated bit held in the register `REG(lst-sAddr2)`. This result is then stored back to the structure in memory.

Note that bit-fields are assumed to reside in unsigned 32-bit (4 byte word) integers.

```
void main(void){
int a,b;
struct {
        int x;
unsigned int field:1;    ⇒
} str;
    a = 1;
    str.field = a;
}
```

```
REG(a) :=  1
REG(lst-sAddr0)(0) <-LOAD(Addr)- ADDR(str)
REG(lst-sValue0)(0) <-LOAD(Int)- REG(lst-sAddr0)
REG(lst-sAddr2) := REG(a) <<  5
REG(lst-sAddr2) := REG(lst-sAddr2)  &  0x00010000
REG(lst-sAddr1) := REG(lst-sValue0) &  0x11101111
REG(lst-sAddr1) := REG(lst-sAddr2) | REG(lst-sAddr1)
REG(lst-sAddr0)(0) <-STORE(UI)-< REG(lst-sAddr1)
```

Figure 3.9: Example of a bit-field reference

In addition to structure references, the mechanics of type casting are explicitly represented. For instance, converting a character to an integer requires that the appropriate eight bits are read and stored in an integer.

Usually architectures provide load and store commands that perform this implicit masking. However, one of the goals of the McCAT compiler is to keep as many values in registers as possible, and so loads and stores should be avoided where ever possible. As a result, instructions that perform the appropriate masking are generated and the result kept in a register instead of utilizing the load/store commands. Figure 3.10 shows the conversion of the character 'a' to an integer. The mask is loaded into a register and a bit-wise and performed.[11] The result is kept in a register. The astute reader may notice an apparent discrepancy: the previous example (Figure 3.9) did not load its masks into a register. This is actually an optimization. For each target architecture, LAST is aware of the range of

[11] Although modelled in LAST as a load, because an integer constant is being used, the code generator actually replaces the load with a cheaper instruction.

integers that can be represented as an immediate constant. For those values that lie within this range, such as -64 (0x11101111) and 128 (0x00010000), they can be represented as immediate constants and thus appear as operands to simple integer arithmetic operations. Larger constants, like 65535 (0xffff), must be placed in a register.

```
void main(void){
char ch;                          REG(ch) :=  97
int i;                            REG(Addr0) <-LOAD- Addr(0x0000ffff)
    ch = 'a';          ⇒          REG(CnvtTmp0) := REG(ch) & REG(Addr0)
    i = ch;                       REG(i) := REG(CnvtTmp0)
}
```

Figure 3.10: Example of a type conversion

3.3.2 Explicit Support of Load/Store Architectures

The second difference between LAST and SIMPLE is, as mentioned previously, that LAST explicitly supports a load/store architecture, whereas SIMPLE has no provision to differentiate between registers and main memory. The consequence of this approach is that all operands, with the exception of load and store operators (Section 4.2.2), are registers or constants and not memory variables. Two corollaries for LAST follow from this approach: variables are explicitly loaded into registers, and are explicitly stored to memory. These two corollaries are explained in the following sections.

Explicit Loads

In a load/store architecture, variables are explicitly loaded into registers. Subsequently, all references to the variable will use that register. There are three situations that cause a variable to be loaded into a register:

1. It is the first use of the variable *ie* there is no existing register holding the value of the variable. This is calculated using a *register use* analysis (Figure 3.7) as one phase of the compiler. The phase consists of a forward traversal of LAST, and the first time a variable is either used or defined, it is marked as being in a register for all subsequent uses. The only exception is when the next use is in an outer block *ie* the first use/definition occurred inside a conditional body. In that case, all of these first used variables that were defined must be stored to memory at the end of the block, and reloaded at the next use. If variables are used in all arms of conditionals (*cf* both `if` and `switch` statements), the information is merged for each arm—Figure 3.11 demonstrates this algorithm.

Inside the **if** body, variable **a** is already in a register and so is not loaded, since it is defined in the first line of the program, whereas **parm** must be loaded. Variable **parm** is again loaded when being assigned to **b**, as there is no guarantee that the **if** statement then body is executed. As a result, **parm** must be stored at the end of the conditional, otherwise the following load would overwrite the value in REG(parm) with its old value.

```
void
foo_lish(int should_incr,
    int parm){
int cond,a,b;
    a = 1;
    if(should_incr){
       a = 2;
       parm = parm + 1;
    }
    b = a+parm;
}
```

⇒

```
;================================
;   Function body for "foo_lish"
;================================
   <<Save Registers>>
    REG(a) :=  1

    REG(should_incr)(0) <-LOAD(Int)- MEM(should_incr)
If Statement:
Conditional:    REG(should_incr)
    end of sequence
Branch delay slot: nop

Then Statements:
    REG(a) :=  2
    REG(parm)(0) <-LOAD(Int)- MEM(parm)
    REG(parm) := REG(parm) + 1
    MEM(parm)(0) <-STORE(Int)-< REG(parm)
    end of sequence
Else Statements:
    end of sequence
end of If Statement

    REG(parm)(0) <-LOAD(Int)- MEM(parm)
    REG(b) := REG(a) + REG(parm)
    <<Restore Registers>>
    end of sequence
```

Figure 3.11: Example of register-use algorithm

2. The variable is an array or structure field. For array dereferences, the memory location is usually computed at run-time, and so the easiest and safest approach is to load the value from memory. Structures, including their fields, have their addresses calculated at compile time, but since structure fields can be arrays, field dereferences are treated conservatively and considered identical to array dereferences (future extensions to LAST may wish to remove this restriction).

3. The variable is a global variable. In the basic, conservative approach all uses of a global are loaded, and all definitions are stored to ensure correctness. Such an approach is necessary since there is no guarantee that the global is left unmodified by a called procedure, and so memory consistency must be maintained. However, with the accurate points-to analysis available in McCAT, this conservative approach can optionally be replaced by another one: loading a global once at the beginning of a basic block, and storing it only at the end (see Section 5.1).

   Because of our analysis algorithms, local static variables are transformed into global static variables with unique names that are used in one function only.[12]

4. The variable is pointed-to. A pointed-to variable is one whose address has been taken at some point in the program, and whose memory location can therefore be accessed via indirect references. Any pointed-to variable is loaded on use and stored on definition. Again, with the points-to analysis available, the compiler can optionally replace all definitely pointed-to variables, and thus reduce the loads and stores to one per basic block. The below example demonstrates the substitution. On the left, the variable `*aptr` is essentially replaced by the variable `a`, as on the right the register `REG(b)` is assigned `REG(a)`.

```
{                                   REG(_aAddr0)(0) <-LOAD(Addr)- ADDR(a)
int a,b,*ptra;                      REG(ptra) := REG(_aAddr0)
ptra = &a;                    ⇒     REG(a) :=  10
a = 10;                             REG(b) :=  REG(a)
b = *aptr;                          MEM(b)(0) <-STORE(Int)-< REG(b)
}
```

Note that temporary variables generated in the *simplify* and *blastify* process are analogous to registers and so are never loaded nor stored.[13]

Explicit Stores

Just as there are explicit loads, so too are there explicit stores. Stores are generated after assignment statements under the following conditions:

1. If the variable is pointed-to.
2. The variable is a global variable (or a static local variable, as explained above).
3. The variable being written to is an array or structure. Although there will be some cases where the same array reference is being read in the immediate future, Last is not the place to optimize this case. There should be a higher level transformation at

[12] The original name is given a unique suffix and prefix.
[13] Unless specifically spilled by the register allocator.

the SIMPLE level to replace the array reference with a scalar variable [CCK90]. For example, Figure 3.12 shows three array references on the left, and a corresponding transformation for them on the right. Note that once scalar variables (like `t_1`) are introduced, LAST automatically keeps them in registers.

Figure 3.12: Optimization of array references

The array reference must be stored because, like the load, the memory location is calculated at run time. So every array reference is either a load or store. Note that the base address and all indexes will be in registers, and are treated like any other variable regarding whether they should be loaded or stored.

It is easy to imagine a situation where loads outnumber the stores (*eg* a program with many scalar variables). If this difference is too large, the register allocator will run out of registers, as the current allocators attempt to keep as many values in registers as possible. It is the task of the register allocator to decide which variables should be retained in registers and which should be stored [Bri92, HGAM92]. A register allocator has been implemented [LJ92], but further discussion is beyond the scope of this thesis.

3.3.3 Unique Variable Nodes

The third difference between SIMPLE and LAST is the nodes used to represent variables. In the implementation of LAST, each leaf node parent is unique *ie* is allocated a different memory location. SIMPLE, on the other hand, reuses nodes to maintain consistency and save space. In LAST, this specific consistency is actually a characteristic to be avoided, otherwise the implementation of chameleon registers becomes complex. The use of chameleon registers is a technique to substitute register moves for register spills [HGAM92]: in this case a variable may be allocated different registers at different points in the program.

To uniquely identify the register nodes, they have pointers to their corresponding SIMPLE variable declaration nodes. See Section 4.2.1 and Figure 3.13.

3.3.4 Parent/Child Relationship

The fourth and final difference is the overall structure and relationship between LAST and SIMPLE. The actual AST structure is different; the leaf nodes of LAST are actually SIMPLE

nodes. Each leaf node parent has a pointer to a SIMPLE node such as a variable, parameter or constant (in SIMPLE known as *VAR_DECL, PARM_DECL* or *CONSTANT* nodes), see Figure 3.13. This approach is primarily a space-saving optimization, but is also useful when storing information with variables (as in the stack reduction optimization—see Section 3.4.4).

In addition, this parent-child relationship allows analyses at the LAST level to access information created and stored at the SIMPLE level, in support of the persistence of flow information.

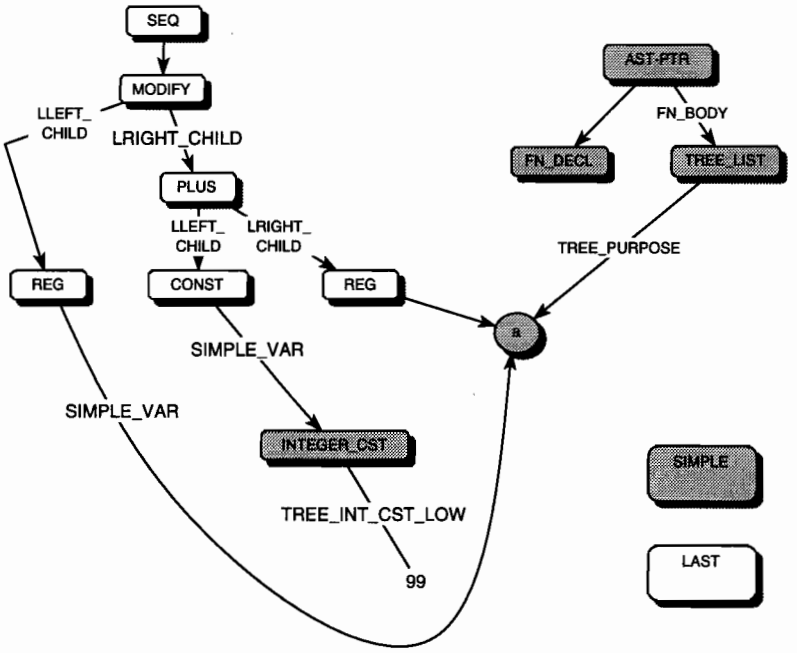


Figure 3.13: Storage optimization

3.4 Optimizations Performed During Generation

Optimizations are performed not only by separate passes on the SIMPLE and LAST IRs, but during the generation of LAST. There are various motivations for incorporating these optimizations during the translation process, but the two main ones are: 1) some information is lost during the transformation, especially regarding arrays and structures, and 2) the transformation is trivial to implement, and so can easily be integrated in the generation

of LAST without significantly increasing its complexity.

Four such transformations are described below. The first is the pre-calculation and folding of offsets, the second the loading of addresses outside of loops, the third the substitution of integer multiply instructions by cheaper operations, and fourth the reduction of space reserved for local variables.

The first three are classical, well-known optimizations; their motivation was not so much to further increase McCAT's code quality, but to keep up with the 'minimum standard' of code presented by other compilers such as GCC [Sta92] or lcc [FH91]. The last was motivated by the simplify algorithm, that can generate an inordinate number of temporary variables.

3.4.1 Pre-calculation and Folding of Offsets

Regular fields[14] in structures are accessed as an offset from a base address. In RISC architectures, memory is usually addressed as a register plus some constant offset, in a form similar to `12(r4)` *ie* access the twelfth byte from the address contained in register `r4`. Rather than placing the offset of some field in a register and adding it to another one containing the structure's base address, the indexed register mode is used for efficiency's sake. An extra field (*LS_OFFSET*) in the *LOAD* and *STORE* nodes (see Figure 4.7) enables this optimization.

In addition, array indices that are constants are folded into the address calculation. For instance, given the array declaration `int a[5][10]`, the address calculation for the dereference `a[i][3]` is : `&a + i` $\times$ `10` $\times$ `4 + 3` $\times$ `4`. The constants are folded together to reduce the number of multiplication operations, producing `&a + i` $\times$ `40 + 12`.

3.4.2 Loading of addresses outside of loops

As shown in the previous optimization, the address calculation for arrays includes adding the array's base address to the offset. The base address is a constant value, and so is a perfect candidate for code motion, and a potentially profitable one, considering that arrays are frequently used inside of loops.

When the address of a variable is loaded in LAST, the load instruction is placed before the outermost enclosing loop, unless of course the address is needed before the loop, where it is just loaded before being needed. All subsequent references to the variable's address refer to the loaded value.

[14] That is, any field other than a bit-field.

3.4.3 Multiplication by Integer Constants

Even in a pipelined architecture, long latency operations such as multiplications can cause problems since finding instructions to execute during the wait can be difficult, especially in non-numerical programs with characteristically small basic block sizes[LW92]. For integer multiplication, where one of the operands is a constant, the multiply operation can be replaced by a series of shifts, additions and subtractions.

However, in performing this substitution, one must ensure that the resulting code is no more expensive than the multiply it replaces. The blastify process generates the best sequence of shifts/adds/subtracts it can find, using a version of Bernstein's algorithm [BH93, Ber86], and if the cost of these instructions is less than a multiply, the multiply is replaced. Figure 3.14 shows a pseudo code sequence generated for the statement `a = 97 * b`. Code for a conventional multiply is on the left hand, and a corresponding sequence using shifts on the right. The code is for the DLX architecture, where an integer multiply operation expects its operands to be in floating point registers.

Comparing the two methods, ignoring identical instructions, the arithmetic operation itself is 10 cycles, the move-integer-to-float (MVI2F) and corresponding move of the result back (MVF2I) operations constitute 3 cycles, plus one cycle for the initial load of the constant, giving a total of $14 + 2 = 16$ cycles.

Using Bernstein's algorithm, four simple arithmetic one cycle instructions plus a register move are used to calculate the result. Total cycles are $5 + 2 = 7$ cycles, thus saving 9 cycles.

There are two more advantages to using this transformation. Firstly, the naive method uses a total of three integer registers and two floating point, whereas the second method uses only four integer registers. The second advantage occurs when the integer constant is large. Usually immediate constants are represented by no more than 16 bits *ie* their values range from -32767 to 32768.[15] This is because many RISC architectures have an instruction size of 32 bits, so encoded immediate integer constants are necessarily much smaller than 2^{32} bits. When an integer constant exceeds these values, a more expensive operation (typically taking an extra machine cycle) is used to load the value into a 32 bit register and then multiplied, adding the potentially high cost of a load and the extra use of a register. Since the largest immediate constant used in a shift would be 31, the second method does not suffer from this problem.

3.4.4 Reducing Stack Space

For every invocation of a function, stack space must be reserved for the local variables of that function. Unfortunately, during the transformations to SIMPLE and LAST, a large number of temporary variables can be generated—for some benchmarks they are in the hundreds. Even though these variables may have short lifetimes and never be spilled to

[15] On SPARC architectures, constants are limited to 14 bits.

```
REG(temp1) := CONST(97)                          REG(b)(0) <-LOAD(Int)- MEM(b)
REG(b)(0) <-LOAD(Int)- MEM(b)                    REG(temp1) := REG(b) <<  1
REG(temp2) <-MVI2F REG(temp1)                    REG(temp1) := REG(temp1) + REG(b)
REG(temp3) <-MVI2F REG(b)                   ⇒    REG(temp2) := REG(temp1) <<  5
REG(temp4) := REG(temp2) * REG(temp3)            REG(temp2) := REG(temp2) + REG(b)
REG(b) <-MVF2I REG(temp4)                        REG(a) := REG(temp2)
MEM(a)(0) <-STORE(Int)-< REG(a)                  MEM(a)(0) <-STORE(Int)-< REG(a)
```

Figure 3.14: Replacing multiply with shifts for a = 97 * b

the stack, a naive compiler will allocate stack space for all of these variables, which can be extremely wasteful, especially for recursive functions.

The solution taken was to mark all variables that are either loaded or stored to memory, or had their address taken (in case they were referenced indirectly) *ie* any child node of a MEM or ADDRESS node, determined during the blastify process. Then, when creating offsets (see the create offset module in Figure 3.7), only those variables so marked are allocated stack space.

3.5 Implementation Restrictions

The final section of this chapter briefly describes the limits of the implementation of LAST.

All facets of SIMPLE C are handled, except for

- Type conversions involving `long long` integers.
- Bit fields using any type other than unsigned integers.
- The interprocedural `goto` instruction `longjmp()` (and its companion `setjmp()`).
- Stack based memory allocation using `alloca()`.

In addition, there are the following restrictions:

- Only 32 bit wide registers are supported. For type conversions, 32-bit float and integer registers are assumed, with pairs of floating point registers giving 64-bit precision. A specific precision is required for the various bit masks generated, although switching to a 64-bit register architecture should be relatively painless. Currently, the majority of commercial RISC architectures are 32 bits, with DEC's new Alpha chip being an exception [Dig92].
- Individual structures are limited in size. Since an offset to a register is used to access structure fields, structures are limited to be only as large as the maximum value an offset field can represent, which can range from 2^{14} to 2^{16}bits.

Chapter 4

Detailed Description of LAST

The LAST intermediate representation is implemented as a doubly linked-list of (usually) binary trees. There are many different LAST nodes, but they can be classified into four main categories: structural nodes, architectural nodes, operator nodes and control-flow nodes.

In the illustrations of LAST nodes and trees that follow, a specific convention is followed. The nodes are either shaded or unshaded, with shaded nodes representing SIMPLE nodes, and unshaded being LAST nodes. Connecting the nodes are three types of arcs: solid, dashed and dotted. Solid arcs represent the 'normal' links that connect nodes, and are labeled with the C macro used to access the field (for the sake of brevity only the first instance of each type of arc named). The dashed arcs are also labeled, and represent connections that are invisible to the normal traversal, but are still accessible when specifically addressed. They are used only to circumvent the binary restriction placed on LAST. The last arc type, the dotted arc, represents arcs that are used only for implementation purposes, but are meant to be ignored in the conceptual model of LAST. Figure 4.15 and Figure 4.16 illustrates these two types of arcs.

4.1 Structural Nodes

The first category of nodes in LAST is structural nodes. These types of nodes provide LAST trees with their tree-like structure (for analysis), and to aid manipulation of them (for transformations). There are three sub-types of structural nodes: common, sequence and anchor nodes.

4.1.1 Common Nodes

A common node is not really a node, but is part of all nodes. Every LAST node has a common component, and illustrated in Figure 4.1.

There are several fields for this node, as described below.

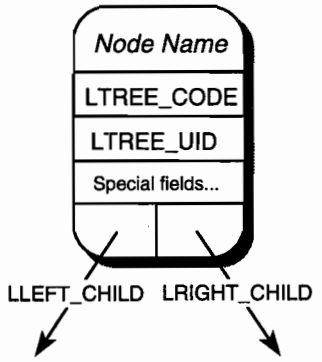


Figure 4.1: The fields common to every LAST node

code: an eight-bit field that contains the node's name.

uid: an unsigned integer uniquely identifying each LAST node. Although of limited usefulness in various optimizations/analyses, the uid has proved very useful for debugging purposes.

children (left and right): two pointers to two other LAST nodes, usually used to represent children. While not all nodes use both these fields, they simplify the traversal of LAST.

There are also some 'special fields', which are reserved fields shared by different nodes for space reasons.

4.1.2 Sequence Nodes

Sequence nodes are the glue of LAST, and hold all the trees together. Programs are represented as doubly linked-lists of sequence nodes (represented by SEQ) with various LAST trees as children—SEQ nodes are parents of every LAST sub-tree (Figure 4.2). LAST trees under a SEQ statement can be of any type, except another SEQ node, an arithmetic node (except for MODIFY–explained in Section 4.3) or a EOSEQ node (Section 4.1.3).

For saving flow information generated by analyses run on the LAST IR, SEQ nodes have a pointer, SEQ_FLOW, used to point to an arbitrary structure for containing flow information (Figure 4.2).

SEQ nodes maintain a pointer back to the corresponding SIMPLE sub-tree that generated the particular LAST sub-tree. This arc, called EXPR_STMT_PTR (see Figure 4.3), is used to access the flow information deposited by various SIMPLE analyses, and is the mechanism to support the pervasive flow information in McCAT. It points back to the root of a SIMPLE expression tree, such as an assignment statement. In the case where an abstract data type,

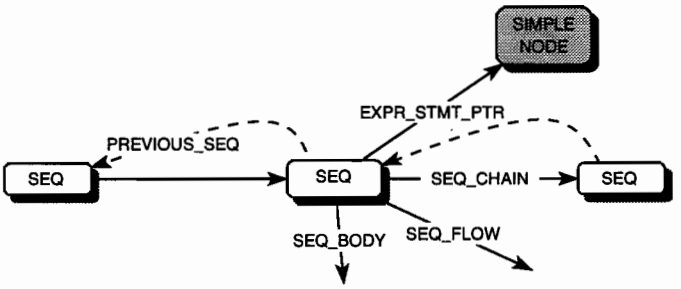


Figure 4.2: SEQ node

such as an array or structure, is referenced, then a series of LAST sub-trees are generated, with all their EXPR_STMT_PTR arcs pointing to the same SIMPLE expression node (a many-to-one relationship). There are also situations where LAST subtrees are generated independently of the SIMPLE IR, especially during register allocation if spill and reload code is generated. In such cases, the EXPR_STMT_PTR has no corresponding SIMPLE expression tree to point to, and so is set to null.

4.1.3 Anchor Nodes

These are special nodes that are used exclusively to support instruction scheduling. Their function is to *anchor* a sequence of nodes, so the instruction scheduler is guaranteed to have nodes that remain immobile, pointing to the beginning and end of basic blocks. There are two such nodes, called begin body (BEGIN_BODY) and end-of-sequence (EOSEQ). Begin body nodes are the child of the first sequence node in every body, (except for conditional and delay slot bodies[1]), and EOSEQ nodes terminate SEQ lists.

EOSEQ nodes also serve a dual function in LAST as a terminator of SEQ chains. The code-generator generator used in McCAT, BURG, requires that terminal nodes be used for trees it traverses, rather than simply having a null terminated list.

4.2 Architecture-exposing Nodes

The second category of LAST nodes is perhaps what most differentiates LAST from SIMPLE. These nodes expose the underlying architecture and directly represent it: they differentiate register and memory references, expose function prologue and epilogue, parameter passing, and explicitly represent delay slots. These nodes are designed to expose optimization opportunities of the target architecture.

[1]Since the scheduler does not attempt to reorder instructions in either conditional or delay slot bodies, the begin body node is unneeded in these bodies, and so is left out to save space.

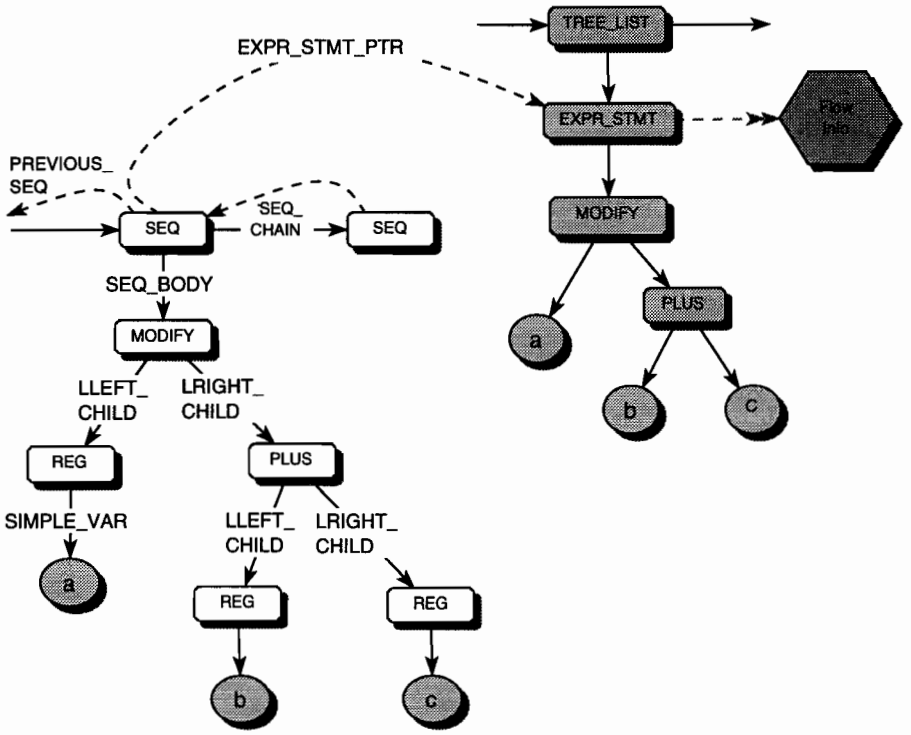


Figure 4.3: How a SEQ accesses the flow information in an EXPR node

4.2.1 Variables, Addresses, Constants and Labels

Because variables can reside in both registers and memory, LAST has two different representations for a variable: one for when it is in a register (REG), and one for when it is in memory (MEM). There is also a representation of the address of a variable (ADDRESS), and constant node (CONSTANT), and a label (LABEL), used for denoting the name of functions.

These five nodes are all similar in that they are all parent nodes, and point to SIMPLE nodes. This is a result of the storage optimization discussed previously, and shown in Figure 3.13.

REG: The REG node corresponds to a register holding the value associated with the SIMPLE variable it points to (written `REG(a)`, where `a` is the SIMPLE variable—see Figure 4.4). For example, if variable `a` is assigned the constant 1, then REG(a) would be assigned the value 1. The REG node can access the memory location information in the SIMPLE nodes in case the register allocator decides it must be spilled to memory. There can be an infinite number of REG nodes; at register allocation time they will be mapped to an appropriate real register.

```
a = 1;        ⇒        REG(a) = 1;
```

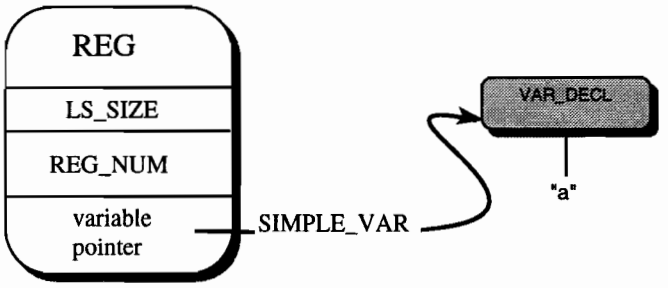


Figure 4.4: REG node

MEM: The MEM node indicates the value contained in the actual memory variable. The node is used only with LOAD or STORE operators. In Figure 4.5, the value of `b` is loaded from memory and placed in a register associated with `b`. The register for `a` (REG(a)) is assigned the value of `b`, and then stored to memory.

ADDRESS: The ADDRESS (ADDR in the figures) node indicates the *address*, not value, of the associated SIMPLE variable. This node replaces the C address operator `&`. In addition, besides holding the contents of variables, a REG can also contain the address of some memory location like a pointer variable, structure or array reference.

```
a = b;          ⇒    REG(b)(0) <-LOAD(Int)- MEM(b)
                     REG(a) := REG(b)
                     MEM(a)(0) <-STORE(Int)- MEM(a)
```

Figure 4.5: Example of load and store in LAST

In Figure 4.6, the address of `b` is assigned to `REG(ptrb)` and `REG(ptra)`. A new register is created, called `REG(iref-ptra)`, to hold the value of the dereferenced variable `*ptra` (which is the value 10). The value in this register is then stored at the memory location to which `ptra` points.

```
int b,*ptrb,*ptra;                  REG(addr-b)(0) <-LOAD(Addr)- ADDR(b)
ptrb = &b;                  ⇒       REG(ptrb) := REG(addr-b)
ptra = ptrb;                        REG(ptra) := REG(ptrb)
*ptra = 10;                         REG(iref-ptra) :=  10
                                    REG(ptra)(0) <-STORE(Int)-< REG(iref-ptra)
```

Figure 4.6: Example of pointer dereference in LAST

CONSTANT: The CONSTANT node points to the corresponding SIMPLE CONSTANT node, and is used for all constants (Figure 3.13).

LABEL: The final leaf parent is the LABEL node, which points to the variable declaration node of a function.

4.2.2 Load and Store Nodes

As mentioned previously, LAST models a RISC machine, one of whose characteristics is being a load/store architecture. LAST therefore has nodes to explicitly represent both loads and stores of variables. Loads and stores take two children: a 'source' child node and a 'destination' child node. There is also an LS_OFFSET field, which is used to take advantage of the indexed register addressing mode usually used in RISC architectures (Figure 4.7).

LAST abstracts whether a local or global variable is being loaded or stored. Determining a global variable's address is more expensive than a local, but because calculating this address varies widely between architectures, it is left up to the code generator to handle.

Load Nodes

Load nodes always have a REG node as their destination child, but can have one of the following four types of nodes as their source (see Figure 4.7).

1. REG(ptra): a register associated with variable 'ptra', holding the address of a memory location. This node is most commonly used for accessing the contents of arrays, structures and pointer variables. In Figure 4.6, the register `REG(ptra)` is used to hold the address of variable `b`.

2. MEM(a): a memory location holding the value associated with 'a'.

3. ADDRESS(a): the address of the variable 'a'. MEM(a) is a value, ADDRESS(a) (or ADDR(a)) is an address. Note that while it is represented in LAST as a load, the code generator substitutes cheaper instructions.

4. CONST(99.0): the location of a non-integer constant (*eg* a real constant 99.0). CONST can represent strings and reals as well as integers that are too large for the immediate constant mode, *ie* larger than 2^{16} for the DLX architecture.[2] Note that when loading such large integer constants, the code generator actually does not generate a load, but detects this special case and substitutes cheaper instructions than a load.

Store Nodes

Store nodes are similar to load nodes, but with the children reversed. The source must be a register, and the destination node must be either a MEM node, or a REG node (see Figure 4.7). The destination REG node will hold an address identical in function to the corresponding source REG in the load instruction.

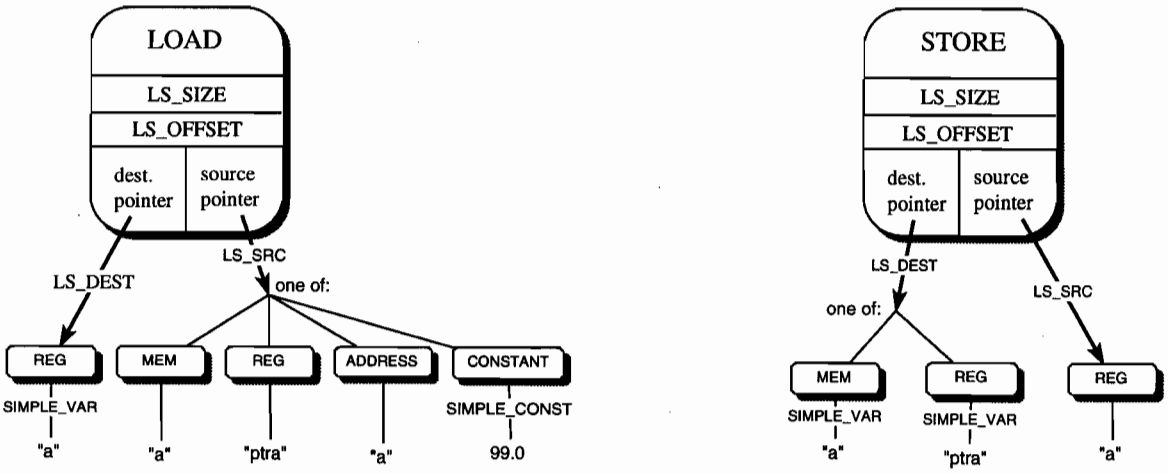


Figure 4.7: LOAD and STORE nodes

[2] 2^{14} for SPARC.

4.2.3 Function Declarations

Another set of nodes that help expose the architecture to the compiler are those dealing with function declarations. Function declarations reuse most of the structure of SIMPLE declarations; the function name and the parameters are unchanged. The only difference is an extra pointer to a LAST version of the function body (see Figure 4.8).

The LAST function body is bracketed by two special nodes: SAVE_REGISTERS and RESTORE_REGISTERS. SAVE_REGISTERS represents the function prologue code that saves all registers used in the function body to the stack, and RESTORE_REGISTERS the function epilogue that restores them. In addition, the RESTORE_REGISTERS is in effect the target label of all **return** statements, and so at code generation time a label is also emitted when this node is processed.

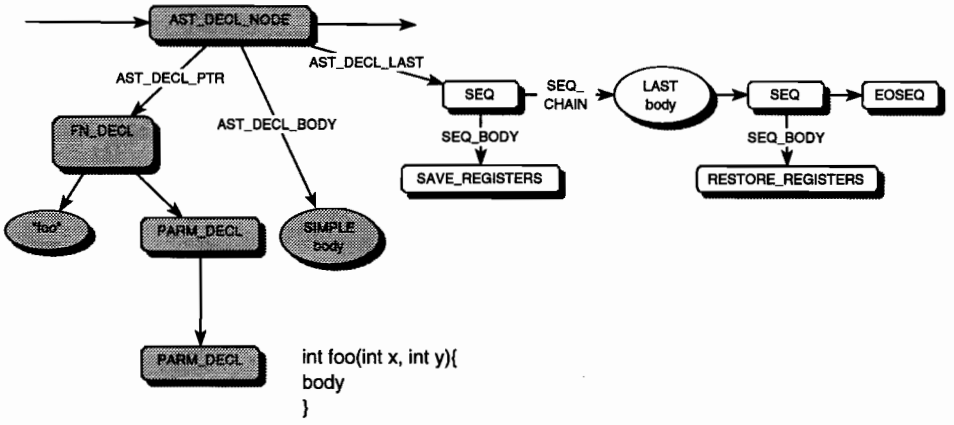


Figure 4.8: LAST function declaration

Since load/store architectures tend to have many registers, the save and restore sections are expensive (since many registers may have to be saved and restored around function calls), and are an important part of function call optimization. During register allocation, the register saves and stores are inserted for only the registers used in the function body (only the register allocator knows exactly what registers are used). The SAVE_REGISTERS and RESTORE_REGISTERS nodes are left untouched and used as a reference point for **return** statements. Subsequent instruction scheduling optimizations can further improve program performance by interleaving function body instructions with the save/restore instructions.

4.2.4 Passing Parameters

LAST explicitly represents the passing of parameters to a function. Implementing these nodes compromises the retargetability mandate of LAST, but the benefits are significant

```
main(){
   int a = 10;
   printf("%d",a);
}
```

⇒

```
REG(a) :=  10
REG(_aAddr0) <-LOAD- Addr("%d")
REG(temp_0) := REG(_aAddr0)
Adjust stack pointer
Pass parameter REG(temp_0)
Pass parameter REG(a)
Function Call to printf
delay slot: nop
Arguments:
    Parameter: REG(temp_0)
    Parameter: REG(a)
Pop parameters
```

Figure 4.9: Passing parameters via the stack

enough to justify their existence. Parameter passing is such an expensive operation that it has motivated the design of register windows specifically to cheaply pass parameters to functions (*eg* SPARC [DS90]). If a more retargetable approach is taken by abstracting the function call and passing of parameters to be one LAST node, then a significant optimization opportunity is lost.

Two parameter passing methods are thus required: the traditional pass-by-stack method, and via register windows. The solution to supporting both parameter passing paradigms was to create specific nodes for each method. For the former, the nodes *ADJUST_SP*, *PASS_PARM* and *POP_PARMS* are used to represent adjusting the stack pointer, passing parameters, and restoring the stack pointer respectively (Figure 4.9).

As a brief review of a register window architecture, such as the SPARC [DS90, pp. 307-315], there is a circular buffer of registers used to pass parameters to functions. At any one point, there is an active 'window', consisting typically of 24 registers: eight *in*, eight local, and eight *out* registers. The *in* registers contain the parameters passed into the function call, and the *out* registers the parameters to be passed to the next function call. When a call is made, the active window is moved forward by 16 registers, so the *out* registers become the *in* for the next function. For a fuller description, the reader is directed to Dewar and Smosna [DS90, pp. 301-340] or Hennessy and Patterson [HP90, pp. 450-454].

Note that since functions with more parameters that *out* registers exist, architectures with register windows must also sometimes pass parameters via the stack. To support register windows, two additional nodes are used: REG_WIN_OUT and REG_WIN_IN. REG_WIN_OUT moves a parameter into a specific *out* register before the function call, and REG_WIN_IN is used inside the called procedure to retrieve the parameter from its *in* register. If the register allocator is clever it can initially allocate REG_WIN_OUT registers to prospective parameters. Otherwise some additional register moves to the *out* registers

may be required.

Since register windows are supported in the intermediate representation, it is possible use registers to pass parameters, even on machines without register windows: by simply utilizing a convention for which registers to reserve for passing parameters, the cost of function calls can be reduced significantly [Wal88, Cho88].

4.2.5 Delay Slots

Another set of nodes that help expose the target architecture are those dealing with delay slots. Delay slots are not specific nodes, but rather a classification of nodes. They are simply regular subtrees of LAST nodes that are isolated and identified as being part of a delay slot.

There are two types of delay slots present in RISC machines: load delay slots and branch delay slots [HP90, pp. 265-268,273-276]. On some architectures, load delay slots are implicitly inserted by the hardware (*eg* SPARC [CKDK91, p. 294]), while in others the compiler must explicitly handle them (*eg* MIPS [DS90, p. 296]). Branch delay slots, on the other hand, are generally present in RISC architectures. For these reasons, LAST models only branch delay slots, and lets the instruction scheduler add nops as appropriate for load delays. If no instruction scheduling is performed, then a nop instruction is emitted as appropriate at code generation time.

LAST also exposes the branch delay slots present after all conditional and unconditional jumps, and are labeled in the diagrams by an oval. The oval represents a sequence of instructions, terminated by an EOSEQ node, but are initially only one SEQ with a nop as a child. The flexibility is there for delay slots greater than one cycle, as might be present in super-pipelined architectures [JW89].

4.2.6 Looping Nodes

To implement a loop in assembly language, there is usually an unconditional branch to a test condition *ie* a `continue` statement at the end of a loop. Since there will be a branch delay slot associated with this unconditional branch, this implicit `continue` is explicitly represented, along with its branch delay slot. There are three such nodes: JUMP_OVER_ELSE, JUMP_TO_WHILE and LOOP_TO_FOR, for the unconditional branch in `if`, `while` and `for` statements. There is no similar unconditional branch for `do while` loops.

4.3 Operator Nodes

Operator nodes are trivial nodes, and usually correspond directly with assembly instructions. The abstract RISC machine to which LAST is targeted is assumed to have an instruction to implement each of these operations, although if a particular machine does not, then

blastify can be altered to generate the equivalent behavior using other arithmetic nodes. The operator nodes consist of arithmetic, logical and conversion operators, and take either one or two arguments (children), which are either a REG node, or in some special cases a CONSTANT node.

4.3.1 Arithmetic Nodes

Table 4.1 lists the arithmetic nodes, with the MODIFY node being special because it can also take any other single arithmetic or logical node as its child, except another MODIFY node. The TRUTH_NOT operator, while supported in LAST, never appears in programs since the GCC front-end removes it through boolean algebra.

Name	Equiv. C code	Description
MODIFY	a = b	assign
PLUS	a + b	addition
MINUS	a - b	subtraction
DIVIDE	a / b	division
MOD	a % b	modulus
MULTIPLY	a * b	multiplication
BIT_AND	a & b	bit-wise and
BIT_IOR	a | b	bit-wise inclusive or
BIT_XOR	a ^ b	bit-wise exclusive or
BIT_NOT	~a	bit-wise not
NEGATE	-a	negation
TRUTH_NOT	!a	boolean zero/not zero
LSHIFT	a << b	shift left
RSHIFT	a >> b	shift right
NOP	a + 0;	no operation (wait 1 cycle)

Table 4.1: Arithmetic nodes

4.3.2 Logical Nodes

Table 4.2 lists all the logical operators, which are usually found in the conditional subtree associated with flow-control nodes (described below). They can also be the child of a MODIFY node.

Name	Equiv. C code	Description
GE	a >= b	greater or equal to
GT	a > b	greater than
EQ	a == b	equals
LT	a < b	less than
LE	a <= b	less or equal to
NE	a != b	not equal to

Table 4.2: Logical nodes

4.3.3 Conversion Nodes

Conversion nodes are used to implement type casting, and support architectural restrictions such as certain arithmetic operations requiring float/integer only registers. Table 4.3 lists these nodes. On some architectures, such as the RS/6000, the equivalent of some of these nodes do not exist, and so the Blastify phase generates their equivalent using arithmetic nodes.

Name	Equivalent type cast	Description
FD2S	(float)a = (double)b	double to single precision
FD2I	(int)a = (double)b	double to integer
FI2S	(float)a = (int)b	integer to float
FI2D	(double)a = (int)b	integer to double
FS2I	(int)a = (float)b	float to integer
FS2D	(double)a = (float)b	float to double
FIXUD2S	(unsigned short)a = (double)b	double to unsigned short
MVI2F		move from integer to float register
MVF2I		move from float to integer register

Table 4.3: Conversion and register move nodes

Appendix B contains a full description of the LAST grammar, and details the use of operator nodes.

4.4 Flow of Control Nodes

The fourth and final class of nodes in LAST are the flow-of-control nodes, which represent control-flow nodes like `while`, `do while`, `if`, `for`, `switch`, and function calls. Implementing change of flow control implies the use of (un)conditional branches, so all of these nodes have delay slots.

4.4.1 While and Do-While Statements

The `while` and `do while` statements are structured similarly to one another. Figure 4.11 illustrates both `while` and `do while` structures. All `while` statements have labels for the beginning of the loop (LWHILE_STARTLBL for `continue` statements and loop iteration), and the end (LWHILE_ENDLBL, for `break` and a failed condition).

In addition, there is the JUMP_TO_WHILE node, which represents an unconditional jump back to the LWHILE_STARTLBL label, to repeat the loop. In keeping with the compositional approach it has its own branch delay slot for instruction scheduling (see Figure 4.10 and Figure 4.12).

```
cond=10;
while(cond)
{
   cond--;
}
```

Figure 4.10: `while` loop example

The `do while` statements are quite different, since they consist of only one conditional branch, versus `while`'s conditional and unconditional branch. The condition sub-tree is still attached to the DO_STMT node to make analysis inexpensive, but it is really associated with the conditional branch that is generated after the code for the loop body.

In addition, the DO_STMT also contains an extra jump label, DO_CONDJUMP. The LWHILE_STARTLBL is used for jumping backwards to repeat the loop, but cannot be used for the `continue` statement since the condition body is at the end of the loop; thus DO_CONDJUMP contains the jump target for `continue`. LWHILE_ENDLBL works as before as the `break` target.

Figure 4.14 provides an illustration of the three labels required for `do while` statements. On the left is the pseudo code, and on the right the equivalent DLX code for the C code in Figure 4.13.

The first label `LWStart1` is the target for the `bnez`[3] instruction, used when beginning

[3] *B*ranch if argument is *n*ot *e*qual to *z*ero.

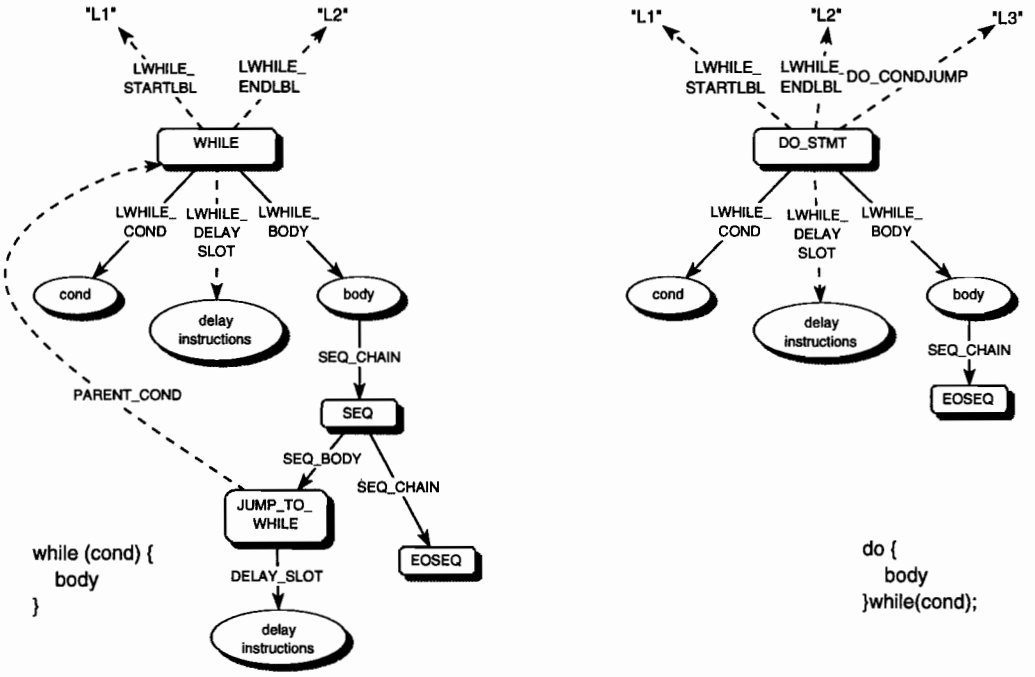


Figure 4.11: `while` and `do while` statements

```
REG(cond) :=  10

While Statement:
Conditional: REG(cond)
end of sequence
Branch delay slot: nop
While Body
REG(cond) := REG(cond) -  1
Jump-to-while branch delay slot: nop
end of sequence
```

⇒

```
addi r2,r0,#10  ; cond = 10;

    ; While Statement:
LWStart0:           ; continue label
    beqz r2,LWend0  ; cond !=0
    nop
    ;While Body
    subi r2,r2,#1   ; cond--;
    j LWStart0
    nop
LWend0:             ; break & end label
```

Figure 4.12: Pseudo and DLX assembly code illustrating labels in a `while` statement

```
cond=10;
do
{
   cond--;
}while(cond);
```

Figure 4.13: `do while` loop example

another loop iteration. The second label, `LWStart2`, is the target of any `continue` statement, and the third label, `LWend1`, is the target for any `break` statement in the loop.

```
REG(cond) :=  10                          addi r2,r0,#10    ; cond = 10

Do While Body                             ;Do While Statement:
REG(cond) := REG(cond) -  1           LWStart1:
end of sequence                  ⇒        ;Do While Body
Do While Statement:                       subi r2,r2,#1     ; cond--
Conditional: REG(cond)                LWStart2:             ; continue label
end of sequence                           bnez r2,LWStart1  ; cond != 0
                                          nop
Branch delay slot: nop                LWend1:               ; break & end label
```

Figure 4.14: Pseudo and DLX code illustrating labels in a `do while` statement

4.4.2 If Statements

If statements in LAST are peculiarly constructed because the code-generator generator BURG allows only binary tree representations. This constraint requires the use of the IF_ELSE_HACK node, whose left child is the then body, and right the else body—see Figure 4.15. In the figure, the solid lines, as in the previous diagrams, represent arcs that are normally visible to the compiler (and are labeled with the macros used to access them). The dashed and dotted lines are used for special purposes: they represent the LAST AST structure, where they are required to handle the binary tree requirements of BURG. However, the macros IF_ELSE_BODY and IF_THEN_BODY make this constraint transparent, as they allow access to the then and else bodies from the parent `if` node. The dashed lines are back edges that are used to access jump labels, which are stored in the `if` statement node for easy modification by a branch chain elimination transformation. Like the `while` statement, a conditional branch slot is associated with the `if` node. This slot is invisible to BURG's traversal mechanism, since BURG only recognizes nodes with an arity less than three, but the slots are accessible when specifically addressed.

There is always a then body and else body, but either may contain only the EOSEQ node *ie* be empty. When the else body does contain statements, then the JUMP_OVER_ELSE node is inserted into the then body. This node represents an unconditional jump of the code of the else body (as occurs in the assembly generated for an `if` statement). This node has a delay slot, and an arc pointing back to the `if` node to enable access to the labels.

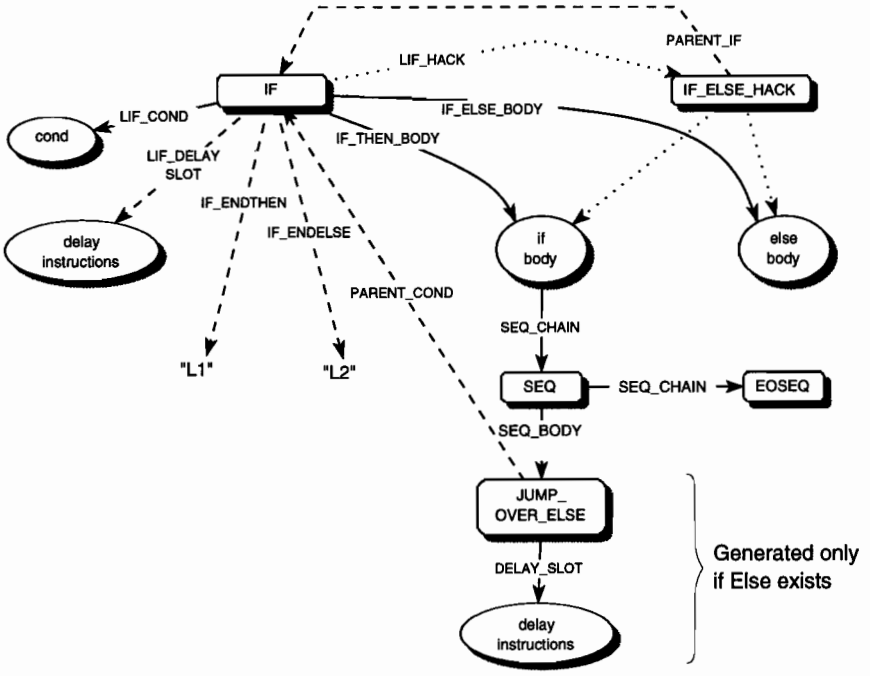


Figure 4.15: If statement

4.4.3 For Statements

The `for` loops in LAST are a mix of `if`, `while` and `do while` statements; because of BURG's constraints, the initialization, iteration condition and increment statement must be split up into binary trees, as illustrated in Figure 4.16.

Like the `while` loop, there is an unconditional branch at the end of the body, as well as a conditional branch. There is also a similarity with `do while` statements, as a third label is needed to handle `continue` statements (DO_CONDJUMP).

4.4.4 Switch Statements

In LAST, `switch` statements are divided into two types of subtrees. The first correspond to the labels of each case statement, and the second to the instructions to be performed should its case label be chosen. LAST assumes that when implemented, there will always be a test of the `switch` expression, and then a conditional jump to the appropriate case label. The implementation of the logic for determining the correct `case` label is not exposed at the LAST level. It is felt that there is little opportunity for instruction scheduling, and there are several different approaches to generating the appropriate code, and should thus be handled abstractly by the code generator. For example, if the case label density is sufficient, either branch tables [Ber85, HM82, Sal81] can be generated, or a more space-efficient method, *ie*

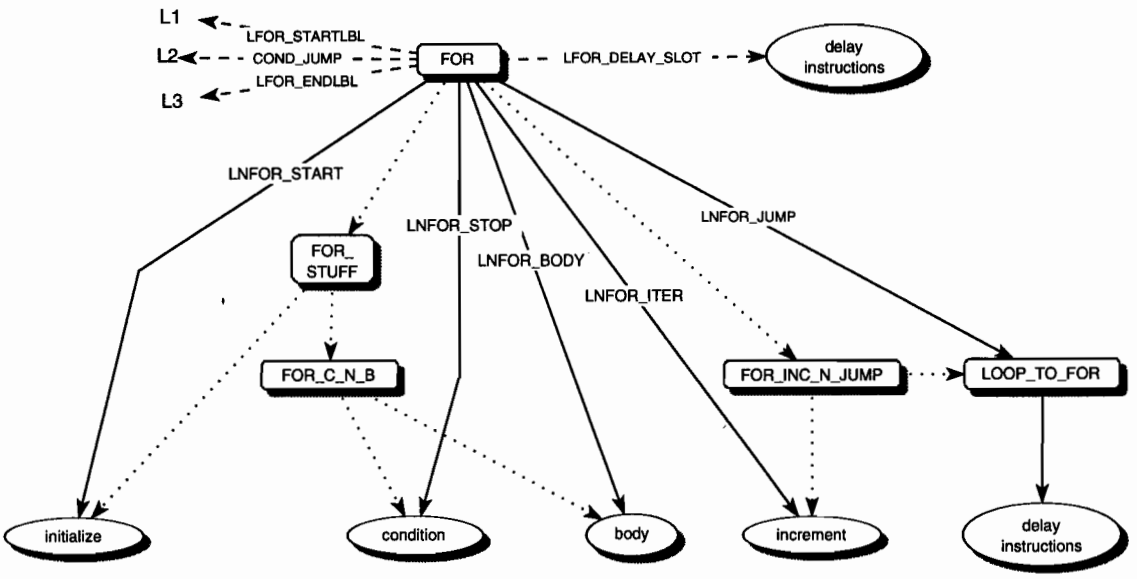


Figure 4.16: For loop statement

a series of test-and-branch instructions. However, whatever the method, there will be at least one conditional branch, with an associated branch delay slot. This slot is filled by the instructions in the LSWITCH_DELAY_SLOT.

Figure 4.17 illustrates a typical switch statement. Note that the condition expression of the switch statement is kept in a register (labeled **result** in the figure). This register is used in evaluating the expression when calculating to which **case** statement to jump.

4.4.5 Return, Continue and Break Statements

The **return**, **continue** and **break** statements are all handled similarly. They all have an unconditional jump delay slot, and generate jump instructions to an appropriate label. The return node is different in that it can also have a "body" of instructions. Figure 4.18 illustrates a **return** statement.

4.4.6 Function Calls

The function call node can have two types of right children: a register holding the address of a jump target, or a LABEL node, indicating a jump target in the form of a label (see Figure 4.19). In the first case, `foo` is called directly as `foo(tmp0,tmp1)`, and so a LABEL node would be used (with the label name `foo`). In the second case, `foo` is called via a function pointer as `fooptr(tmp0,tmp1)`, in which case a REG node is used to hold the address of `foo`.

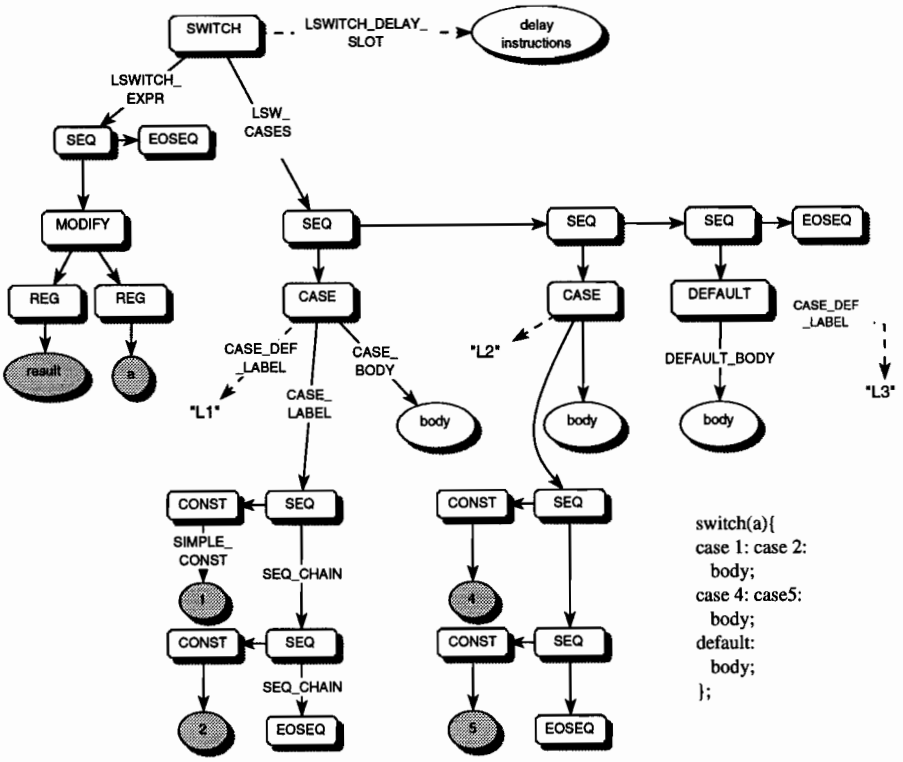


Figure 4.17: Switch statement

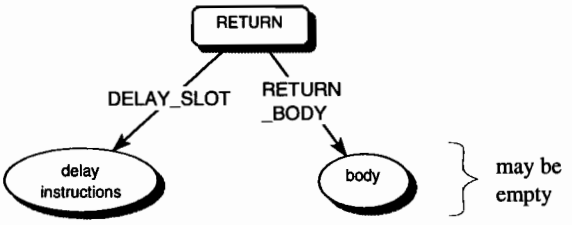


Figure 4.18: Return node

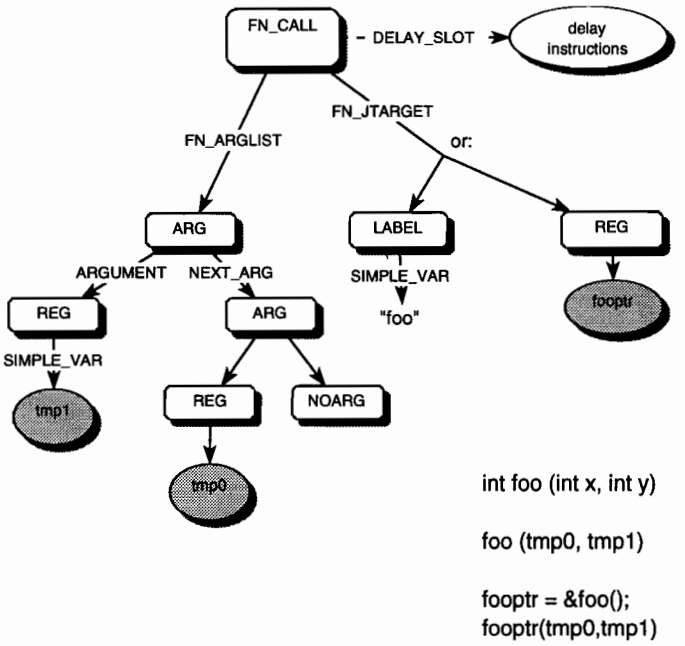


Figure 4.19: LAST function call

The left child points to a list of arguments, which are kept as a chain of ARG nodes. The left child of an ARG node is the function parameter. The leftmost ARG node contains the last parameter (they are stored in reverse order). The list of ARG nodes is terminated by a NOARG node. Since parameters are passed explicitly, one may question the utility of associating them with the function call node. The ARG nodes are used for the analysis phase in order to explicitly represent the dependency between parameter passing nodes and the arguments. When using the stack passing paradigm, this relationship is irrelevant, but if passing parameters in registers using a simple convention of reserving registers (on machines without register windows) then it is important to indicate a particular mapping of registers.

Chapter 5

Transforming LAST

Usually transformation phases follow analysis phases (otherwise there is no information to guide the transformation). However, following the flow and phases in Figure 5.1, the first transformation on LAST(spilling) occurs before the first analysis phase (an analysis to generate dependencies is performed at the same time as the transformation), so the example LAST transformations are presented in this thesis first, with the example analysis on LAST presented in the next chapter.

The ability to perform aggressive transformations is the ultimate objective of the McCAT compiler. This chapter presents two example transformations performed on LAST(see the unshaded nodes in Figure 5.1). Other transformations are performed on LAST, but the two examples are enough to provide a basic understanding of how transformations work on a structured IR such as LAST.

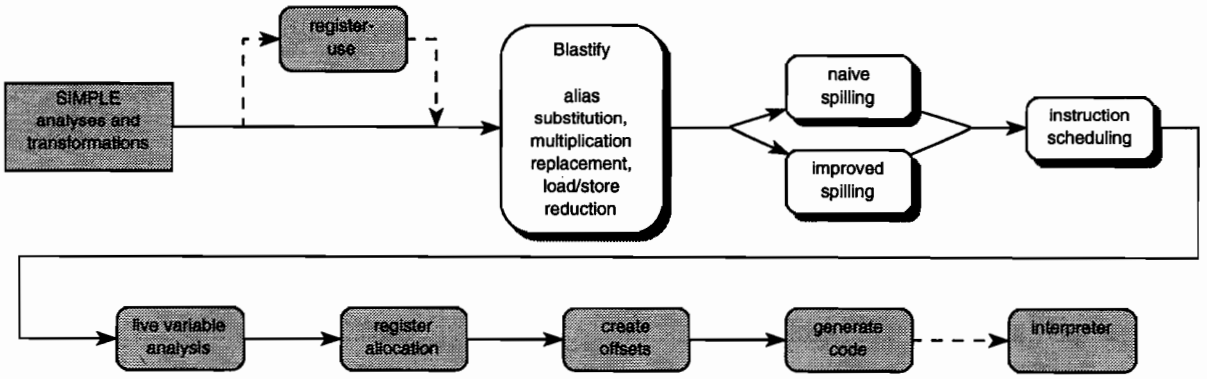


Figure 5.1: Transformations performed on LAST

The first code-improving transformation example is the reduction of the number of loads and stores, and the second is intrabasic block instruction scheduling.

5.1 Reducing the Number of Loads and Stores

Removing superfluous instructions is generally beneficial,[1] so there is an obvious improvement in reducing the number of instructions executed.

However, load and store instructions manipulate a critical resource—the load/store pipeline [SLH90]. The bandwidth available for input-output communication to/from the central processing unit is a limited resource [GH86], and should therefore be minimized. In addition, load instructions are long latency operations, and depending on the speed of main memory and the effectiveness of any cache (if present), a load instruction can take from between several to tens of cycles to complete [CKP91]. Thus reducing loads and stores reduces the pressure on this precious resource.

The transformation is equivalent to keeping variables in registers for as long as possible. In effect, the register set is treated like a small, visible (*ie* manipulable) cache [GH86], and the transformation tries to keep everything in this 'cache'. If the register pressure is too high and there are not enough registers to maintain all the variables in this cache, then the register allocator applies the information from its analysis of the program to select the most 'appropriate' value to remove from the cache (to be replaced by a new value). The register allocator has a very sophisticated algorithm for determining the 'appropriate' values to be kept in registers, and thus will utilize the registers more efficiently than an algorithm in LAST that attempts to portion out the registers—unless of course a LAST algorithm as complicated as a register allocator were to be implemented, which would make the allocator obsolete.

5.1.1 Handling the Register-Memory Consistency Problem

A load/store architecture is motivated, in part, by current technological limitations [SC91]—accessing on-chip data is much faster than off-chip. Therefore, loading data into registers and manipulating them there results in faster access time, and thus faster program execution, but creates a consistency problem: the copy contained in a register can be altered and thus differ from the original in memory. If the memory value is referenced instead of the modified register value, the program will most likely produce incorrect results. The solution is to store the changed value, and reload the updated variable on its next use; the challenge is to reduce these updates.

There are four types of variable references that cause loads and stores in McCAT[2]: references to variables whose address is calculated at run time, references to global and local static variables, references to pointed-to and dereferenced variables, (variables that

[1] Ignoring the need in some architectures for 'filler' instructions. For example the Alpha architecture needs basic blocks to be filled with a minimum number of instructions, even if they are redundant [Dig92].

[2] Ignoring resource limitations, which can cause additional loads and spill *ie* register spills and reloads caused by the register allocator.

refer to the same memory location, also known as *aliased* variables), and references to unaliased local and parameter variables whose first use or definition is inside a conditional body.

These four situations cause a level of uncertainty for the compiler, and the compiler must store and reload due to its overriding mandate to produce correct code, rather than risk a memory-register consistency problem. The transformation presented in the following section decreases this level of uncertainty, so that more variables can be kept in registers over longer sequences of instructions. However, as a starting point, and for comparison, McCAT can be conservative about memory-register consistency. When in its 'conservative mode', McCAT stores and reloads when referencing the following four types of variables.

Run-time address calculated variables: Each reference to either a structure or array causes a load on a use, and a store on a definition. The reason is that without sophisticated analysis, the compiler cannot determine what memory location is being referred to, and given this uncertainty the compiler must ensure correctness and thus generate loads and stores. An array dependence tester exists for McCAT[Jus94], but is unused for the naive approach. A more sophisticated approach could use the array dependence tester to detect opportunities for scalar replacement [CCK90].

Global and local static variables: Global variables have global scope, and except where the name is hidden by a local object (*eg* a local variable), can be referenced at any point in the program in any procedure, as shown in the C code fragment in Figure 5.2.

```
int global_a=9;
void main(void){
   global_a++;
   foo();
   printf("%d",global_a);
}
```

```
void foo(void){
  global_a++;
}
```

Figure 5.2: Referencing a global variable across function boundaries

Globals therefore cannot be kept in registers across function calls without extensive support from and coordination with the register allocator. The allocator would have to allocate the same register to a global variable across all uses, and remember not to include this register in the list of callee saved registers in a function's prologue and epilogue. In McCAT, local static variables are treated identically to global variables (which happen to be referenced in only one function).

Pointed-to and dereferenced variables: A pointed-to variable is one whose address has been taken, and whose memory location may be modified or read via a dereferenced

variable. For example, in the C code fragment in Figure 5.3, `a` is points-to by `ptra`, and `*ptr` is the result of dereferencing `ptra`. Obviously `*ptra` and `a` refer to the same memory location, and are considered to be *aliased* to one another. An unaliased variable is one whose address has never been taken.

```
{
int a,*ptra;
   ptra = &a;
   *ptra = 10;
   a = a + 1;
}
```

Figure 5.3: Pointer dereference example

Since this binding happens at run-time and is unknown without sophisticated points-to analysis, the compiler loads/stores (as appropriate) each reference to either a pointed-to or dereferenced variable.

Unaliased local and parameter variables: The only types of variables left are unaliased local and parameter variables. Usually they are kept in registers and are never loaded nor stored, except in one special case. That case is when the first use or definition of the variable is in a conditional body (see Figure 5.4). If so, then subsequent uses (such as the `printf` in the example) cannot know for sure that the variable was loaded or not, and so will reload the variable. Since all uses after the definition will load the variable, all definitions must store the variable to maintain consistency.

```
void foo(int cond){
int a;
    if(cond) a = 10;
    printf("hello ");
    if(cond) printf("there %d",a);
}
```

Figure 5.4: Example C code showing the first definition of a variable in a conditional body

5.1.2 Algorithm for Reducing Loads and Stores

This section presents two algorithms for reducing the number of loads and stores in a program. The first reduces them for references to unaliased local and parameter variables, and the second for references to global and aliased variables. They are orthogonal, and

```
void foo(int cond){
int a,b;
  if(cond) a = 10;
  b = 100;
  if(cond) a++;
}
```

Figure 5.5: First reference of a variable in a conditional

together deal with three of the four types of references that cause loads and stores in McCAT. References to arrays and structures are still loaded/stored for each use/definition, as a sophisticated high-level transformation (such as scalar replacement [CCK90]) is required, and is beyond the scope of this thesis.

Reducing Local Loads

The first algorithm presented, reducing loads and stores for references to local and parameter variables, is trivial to implement given the high-level information available from the SIMPLE IR, but pays rich dividends in producing superior code. The transformation is based on a simple observation about unaliased local variables:

Unaliased local variables are constrained to exist completely inside the function body in which they are declared. Therefore, in a semantically correct program, for any particular variable there must be a definition of that variable that dominates all uses and subsequent definitions of it, in any execution of the program. Moreover, a variable used before being defined has an unspecified value.

From this observation, one can safely assume that an unaliased variable always resides in a register, and thus neither loads nor stores are generated for any such variable. An aliased variable can be determined in one of two ways. When the McCAT points-to analysis [Ema93, Ghi92, EGH94] is employed, aliased variables are clearly and easily identified. Otherwise, a more conservative approach is used: any variable whose address is taken in the program is considered to have an alias.

Figure 5.6 shows the pseudo code for the trivial C function in Figure 5.5. On the left is the pseudo code for the naive approach, and on the right the approach guided by the above observation. Notice that the variable `a`, first placed in a register in the initial `if` statement, must be stored at the end of the then body block, and again reloaded in the second `if` statement. The pseudo code on the right, however, has no loads or stores for `a`, as a semantically correct program requires that either both `if` then bodies are entered, or neither. In the case of a use occurring before a definition, the C language does not guarantee it a specific value and so leaving a random value in the register is acceptable.

When considering parameter variables, the observation is simply extended—they are

```
<<Save Registers>>
If Statement:
Conditional:
REG(cond)(0) <-LOAD(Int)- MEM(cond)
REG(cond)
end of sequence
Branch delay slot: nop

Then Statements:
REG(a) :=  10
MEM(a)(0) <-STORE(Int)- REG(a)
end of sequence

Else Statements:
end of sequence
end of If Statement

REG(b) :=  100

If Statement:
Conditional: REG(cond)
end of sequence
Branch delay slot: nop

Then Statements:
REG(a)(0) <-LOAD(Int)- MEM(a)
REG(a) := REG(a) +  1
MEM(a)(0) <-STORE(Int)- REG(a)
end of sequence

Else Statements:
end of sequence
end of If Statement
MEM(b)(0) <-STORE(Int)- REG(b)
<<Restore Registers>>
```

⇒

```
<<Save Registers>>
If Statement:
Conditional:
REG(cond)(0) <-LOAD(Int)- MEM(cond)
REG(cond)
end of sequence
Branch delay slot: nop

Then Statements:
REG(a) :=  10
end of sequence

Else Statements:
end of sequence
end of If Statement

REG(b) :=  100

If Statement:
Conditional: REG(cond)
end of sequence
Branch delay slot: nop

Then Statements:
REG(a) := REG(a) +  1
end of sequence

Else Statements:
end of sequence
end of If Statement

<<Restore Registers>>
```

Figure 5.6: Reducing the number of loads and stores of local variables

similar to local variables in that they can only be referenced inside one function body. However, they differ slightly in that they are first defined outside the function. The solution used in McCAT is to insert a dominating load in the function body, so that all subsequent uses can be guaranteed that the value is in a register. The algorithm is straight forward: A function is scanned in a forward manner, and at the first use or definition, a load is inserted. If the use/definition is in the outer most block of the function, the load precedes it, otherwise the use/definition was inside one or more conditional bodies. In this case, the load is inserted before the outermost conditional body. A conditional body is considered to be any control structure, including loops. If the variable is already in a register (for example, a parameter in an *in* register window), no load is needed.

A dominating load is inserted only for those parameters referenced in the function body—if they are never used, no load is generated. If the parameter is used in only one arm of a conditional body (*eg* the then part of an `if` statement), the dominating load is still generated, even though the load may be unnecessary. However, should the parameter be referenced subsequently, then the dominating load pays for itself by eliminating the need of a load for the second use.

Figure 5.7 illustrates this approach. On the left is a C function using the parameter twice, both instances are nested inside a conditional body. The LOAD instruction (marked by `(*)`) is the inserted load that dominates both uses of the parameter (marked by `(%)`). This transformation for local and parameter variables makes a significant difference in reducing cycle time, as shown in Chapter 8, and in addition is extremely cheap to perform, adding a small cost to the overall runtime of the compiler, linear in the size of the program.

Reducing Global Loads and Sure Alias Substitution

The next transformation works on references to global and aliased variables. First, a description of dealing with global references is given, and then a description of the extension to aliased variables.

In the previous section, the transformation was possible because of the limited scope of the variables being referenced *ie* limited to one particular function body. However, references to global and aliased variables can occur anywhere in the program, and thus nothing can be determined *a priori* about the sequence of variable uses and definitions.

In addition, the McCAT register allocator is intra-procedural, so global variables cannot be guaranteed to be mapped to the same register when referenced in different function bodies, so keeping globals in registers across function calls is unsafe. As a result, McCAT aspires only to keep globals in a register within a basic block—they are loaded on the first use in a basic block, and stored at the end if they were defined.

```
void foo(int cond,
         int param1,
         int param2){
int a,b;

  if(cond){
    a = 10;
  }
  else {
    a = param1;
  }

  while(cond){
    cond = param1 -1;
  }
}
```

⇒

```
<<Save Registers>>
REG(param1)(0) <-LOAD(Int)- MEM(param1)  (*)

If Statement:
Conditional:
REG(cond)(0) <-LOAD(Int)- MEM(cond)
REG(cond)
end of sequence
Branch delay slot: nop
Then Statements:

REG(a) :=  10

Jump-Over-Else to Lifend0:
nop
end of sequence

Else Statements:

REG(a) := REG(param1)                    (%)
end of sequence
end of If Statement

While Statement:
Conditional: REG(cond)
end of sequence
Branch delay slot: nop

While Body
REG(cond) := REG(param1) -  1            (%)
Jump-to-while branch delay slot:
nop
end of sequence

<<Restore Registers>>
```

Figure 5.7: Insertion of a dominating load for parameter variables

The algorithm is simply a forward traversal of each basic block, inserting a load immediately before the first use,[3] and when the end of the block is reached, all the globals that were defined are stored.

Dealing with references to aliased variables is slightly more complex, but similar. After running the points-to analysis, two types of aliased variables can be distinguished: *definitely points-to* and *possibly points-to.* A definitely points-to variable is a variable that, at a particular program point, the analysis can statically determine that it points to only one variable. A possibly points-to variable, on the other hand, is a variable that the analysis knows is a pointer, but can give only a conservative estimate of to what it points. In Figure 5.8, for example, the code on the left illustrates a variable `ptra` that the points-to analysis would catergorize as defintely pointing to `a`, at program point X. On the right, at program point X, since `ptrab` is previously dynamically set, it is considered to possibly point to either `a` or `b`.

```
{                                          int foo(int size)
int a=0,*ptra;                             {
  ptra = &a;                               int a=0,b=1,*ptrab;
  *ptra = *ptra + 10; /* X */    ⇔           if(size > 12)
}                                              ptrab = &a;
                                           else
                                              ptrab = &b;
                                           *ptrab = *ptrab + 10; /* X */
                                           }
```

Figure 5.8: Sure and possibly points-to variables

If the referenced variable is possibly points-to, then each use requires a load, and each definition a store. However, if the variable is definitely points-to then a two-phase transformation is used. In the first phase, the dereference operations are replaced by references to the points-to variable, as shown in the C fragment in Figure 5.9 (the replaced dereference operations are on the right).

Invisible variables, *ie* variables pointed to by parameters, are also handled in this fashion. A variable is created at the LAST level to represent the points-to variable while in a register, and this variable is substituted for the dereference operations. Figure 5.10 shows a C fragment on the left, and the corresponding generated pseudo assembly code on the right. The parameter `param` in function `bar` points to the variable `local_a`, which has scope only in function `foo`. Since `local_a` is not visible in function `foo`, then `param` points to an invisible variable. A register called `-parm-param`, using the naming convention `-parm-`<*ParameterName*>, is created, and represents this invisible variable when it is held

[3] Assuming no definitions of the variable preceded it in the basic block.

```
{                                        {
int a=0,*ptra;                           int a,*ptra;
  ptra = &a;                               ptra = &a;
  a = 10;                          ⇒       a = 10;
  *ptra = *ptra + 10;                      a = a + 10;
}                                        }
```

Figure 5.9: Sure alias substitution of pointer dereference

```
foo(void){
int local_a;
   bar(&local_a);
}

bar(int *param){
   *param = *param + 1;
}
```

```
;=======================================
;       Function body for "bar"
;=======================================

<<Save Registers>>
REG(param)(0) <-LOAD(Addr)- MEM(param)
REG(-parm-param)(0) <-LOAD(Int)- REG(param)
REG(temp_1) := REG(-parm-param)
REG(-parm-param) := REG(temp_1) +  1
REG(param)(0) <-STORE(Int)- REG(-parm-param)
<<Restore Registers>>
end of sequence
```

Figure 5.10: Example of an invisible variable

in a register (*ie* it is equivalent to `*param`).

Once alias substitutions have been made, then the second phase of the transformation is applied: loads and stores are inserted in a similar manner as they were done for global variables *ie* a load at the initial use, and a store at the end of the block if the variable was defined. An important observation at this point is that since this transformation occurs only in basic blocks, then the instructions occur in a strictly sequential order, and so a definitely points-to variable will remain definitely points-to throughout the whole basic block.[4]

As a future extension, global definitely-aliased variables could be held in registers across basic blocks that contain no function calls, or over function calls that are identified as free of references to these variables. Such a call must also not have any calls within it that reference these variables.

[4]Although, through the magic of pointer arithmetic, a dereferenced variable cannot always be guaranteed to definitely point to a variable. In this case however, as soon as the dereferenced variable no longer definitely points to a specific variable, substitution from this point on will no longer be performed, and loads/stores will be generated for each use/definition.

5.2 Instruction Scheduling

This section presents an overview of basic-block instruction scheduling algorithms, an enumeration and brief introduction of the scheduling algorithms used on LAST and how the scheduling framework was implemented in LAST.

5.2.1 Overview of Instruction Scheduling

The motivation behind a pipelined architecture is to have the CPU working 'all the time' *ie* an instruction issuing every cycle [HP90]. This requires that the results of one instruction be produced before being needed by a following instruction. For many integer arithmetic operations the result takes only one cycle to compute and so is not a problem, but for others it may take longer: load and multiply instructions, for example, can take several cycles before their result is ready; branches make it difficult to determine what instruction next to fetch, so it can be several cycles until the next one is issued [HP90, DS90].

These long latency operations can degrade the performance of a program in conventional scalar architectures[5] if results are not produced fast enough: instructions that use the result will simply wait until it is ready. With instruction scheduling, programs can run upwards of 25% faster [Tie89, GM86], that is, *without* scheduling, programs can run at least 20% slower.

There are several kinds of instruction scheduling; only local, or intra-basic block, scheduling is performed on LAST, as implementing a cross-basic block (global) scheduler [Fis81] is beyond the scope of this thesis. There are various algorithms for scheduling at the basic-block level, but they all take a similar approach [Kri90], and so it was possible to support several list schedulers in the McCAT compiler (currently five are supported).

Essentially, list schedulers take the instructions in a basic block, create a directed acyclic graph (DAG) of the instructions, where a node represents an instruction, and an arc a dependency between instructions. Then, using this DAG, the instructions are sorted into various levels, where usually all the nodes on the same level can be issued simultaneously, should the hardware support parallel execution on such a scale. A list of potential candidate instructions is generated, from which instructions are selected and inserted in order.

For example, Figure 5.11 shows three array accesses in a single basic block. Figure 5.12 shows the DLX[6] assembly code generated for this C fragment of code; on the left is the unscheduled code, on the right the scheduled version. Looking at the unscheduled code, it is noticeable that the load (`14: lw`) generates a `nop` instruction (`15: nop`) (here, a load

[5] For simplicity of explanation, out-of-order issue and execution architectures are ignored. Due to resource limitations, however, they too can benefit from scheduling. For a more thorough treatment, refer to Johnson's text on superscalar design [Joh91].

[6] DLX is a fictional architecture based on the MIPS R2000 [HP90]. It is one of the target architectures, and is used in this example for brevity.

is assumed to take two cycles to complete), since the store instruction (`16: sw`) uses the result. Note that the entire basic block takes 16 cycles to complete.

```
main(){
int i, A[4], B[4], temp_0;
   i = 2;
   A[i] = 10;
   temp_0 = (i + 1);
   B[i] = A[temp_0];
}
```

Figure 5.11: Three array references in a basic block

Figure 5.13 shows the dependency graph for the DLX assembly code. The instructions have been arranged in levels, where each level indicates the machine cycle (marked on the right) that the instruction starts execution; given enough resources on a parallel machine, the code could complete in seven machine cycles.[7] Looking at the unscheduled code in Figure 5.12, there is one opportunity for overlapping instructions—replacing the **nop** with another instruction, and bring the cycle count down from 16 to 15 (for a scalar, pipelined machine). There are several candidates for replacing the **nop**, and the various algorithms use different heuristics to make a selection. For simplicity, instruction 10 has been chosen, and the scheduled code on the right shows it replacing the **nop** instruction.

The next section lists the various scheduling algorithms supported in the McCAT compiler.

5.2.2 List Schedulers in LAST

As mentioned above, McCAT currently supports five list-schedulers. The contribution of this thesis includes only the framework for instruction scheduling, but the list scheduling algorithms themselves are beyond the scope of this thesis. The scheduling algorithms were therefore cloned from course projects by Erik Altman and Chandrika Mukerji (see the McCAT genealogy section). The list scheduling algorithms used are listed below.

Shieh-Papachristou: This is the default instruction scheduler used, as a previous study found it to be reasonably effective [Muk91]. It has a hierarchical list of 5 characteristics to prioritize candidate instructions and handles floating point functional units, as well as ALU forwarding [SP89].

Level Scheduling: This is the most basic algorithm, and simply calculates the critical path of instructions in each basic block.

[7] Ignoring start-up costs for the pipeline.

```
 1: addi r4,r0,#2     ; i = 2
 2: add r2,r30,#-84   ; r2 = &A
 3: slli r3,r4,#2     ; r3 = i * 4
 4: add r3,r3,r2      ; calculate &(A[i])
 5: addi r9,r0,#10    ; r9 = 10 (r0==0)
 6: sw 0(r3),r9       ; store r9 in A[i]
 7: addi r5,r4,#1     ; temp_0 = i + 1
 8: add r6,r30,#-100  ; r6 = &B
 9: slli r10,r4,#2    ; r10 = i * 4
10: add r10,r10,r6    ; calculate &(B[i])
11: add r7,r30,#-84   ; r7 = &A
12: slli r8,r5,#2     ; r8 = temp_0 * 4
13: add r8,r8,r7      ; calculate &(A[i])
14: lw r8,0(r8)       ; load A[temp_0] to r8
15: nop               ; wait for load
16: sw 0(r10),r8      ; store r8 at B[i]
```

```
 1: addi r4,r0,#2
 2: add r2,r30,#-84
 3: slli r3,r4,#2
 4: add r3,r3,r2
 5: addi r9,r0,#10
 6: sw 0(r3),r9
 7: addi r5,r4,#1
 8: add r6,r30,#-100
 9: slli r10,r4,#2
11: add r7,r30,#-84
12: slli r8,r5,#2
13: add r8,r8,r7
14: lw r8,0(r8)
10: add r10,r10,r6
16: sw 0(r10),r8
```

Figure 5.12: Unscheduled and scheduled pseudo code

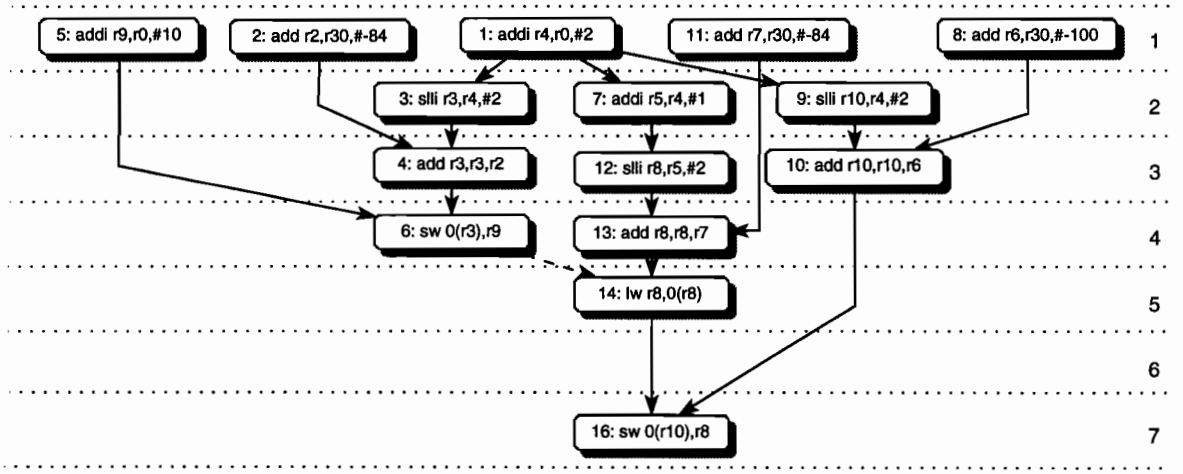


Figure 5.13: Dependency Graph used for Scheduling Array References

Gibbons & Muchnich: This algorithm schedules instructions which have the greatest number of children, inter-locks with at least one of them, and is on the longest execution path [GM86].

Bernstein fixed and variable weights: These are two algorithms based on Bernstein's work [Ber88, BG89], the second being an extension of the first. The first algorithm prioritises each instruction based on its weight, where an instruction's weight is the amount of time it and its children will take *ie* the shortest critical path. Each instruction is assumed to take the same amount of time. The second algorithm tailors the weights to the instructions.

5.2.3 Implementation of Scheduling Framework

As described in the previous sections, the preparatory work is the same for all the list schedulers, and consists of breaking the program into basic blocks, building a dependency DAG for each block, running the scheduling algorithm, and manipulating the basic block list according to the scheduler.

The McCAT scheduler considers a basic block to be a sequence of LAST subtrees delimited by nodes that represent a change of flow control, such as `if`, `while`, `do while`, `switch`, `for`, `break`, `continue`, `return` and function calls. There are several phases carried out in instruction scheduling, as illustrated in Figure 5.14. The unshaded nodes represent those phases implemented as part of this thesis.

The scheduling phases are expose loads, find basis blocks, calculate dag, schedule code and manipulate LAST, and are described below.

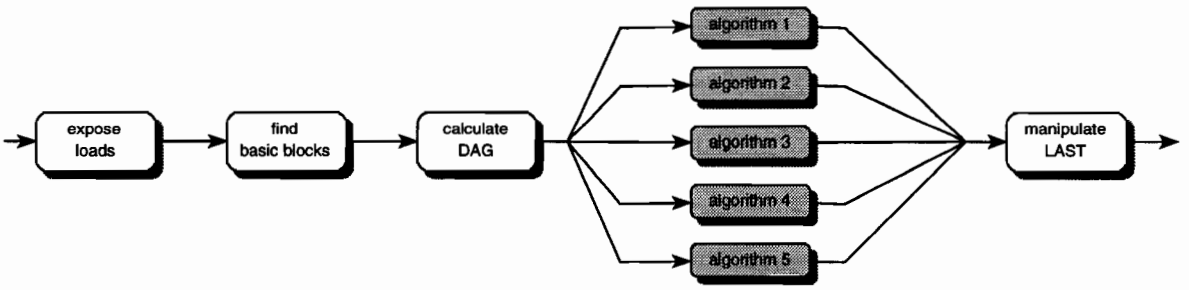


Figure 5.14: Scheduling phases in the McCAT list scheduler

Expose Loads: This phase removes any loads or stores that were in conditional bodies of control-nodes. LAST separates the nodes used for calculating the conditional expressions of these statements to ease analysis. The expose loads phase moves them into their basic block, exposing them to the instruction scheduler (the scheduler does not attempt to schedule either conditional bodies or delay slots).

For example, the statement "`if(a == 0) printf("boo")`" would have a LAST subtree for the `a == 0` expression, and a separate one for the `printf` statement. The conditional expression might require a load of a for the evaluation, and it is this load that is moved to be visible to the scheduler.

In subsequent phases it is possible for the register allocator to again spill or load a register in the conditional body, but it will be up to the allocator to insert a nop node, if appropriate.

Find Basic Block: The second phases counts and identifies individual basic blocks in each function being processed. Two arrays keep track of the beginning and end of each basic block. These two arrays are used by the list scheduler to schedule each basic block individually.

The nodes pointed to by these arrays must be *anchor nodes* or represent change of control flow *ie* must be guaranteed to remain at the beginning and end of each basic block, regardless of how the other nodes are scheduled. The BEGIN_BODY and EOSEQ, and control flow nodes serve as these anchor nodes.

Calculate DAG: The next phase calculates all the data dependencies between the variables, using the variable names to avoid false dependencies. The phase consists of a bottom-up traversal of each basic block, and creating dependency arcs between uses and preceding definitions of registers, and is similar to live variable analysis (explained in Chapter 6). The dependency graph (a directed acyclic graph, or DAG) is used to schedule LAST nodes.

Schedule Code: This phase (indicated by the five nodes labeled "algorithm n") is where one of the possible five scheduling algorithms is invoked to schedule the given basic block, using the DAG just calculated. These algorithms schedule only arithmetic operations, loads and stores—conditional branch delay slots are not handled. Scheduling delay slots is based on control dependencies, whereas long latency operations depends on data dependencies, and thus requires a different algorithm [HP90].

Manipulate LAST: After running the particular scheduling heuristic, a linked list representing the schedule of nodes is passed on. This schedule is used to reorganize the sequence of LAST nodes that represent the basic block being scheduled. The doubly linked nature of LAST greatly simplifies its manipulation at this point.

The effectiveness of instruction scheduling, and of the load/store reduction algorithms, is presented in the results chapter (Chapter 8).

Chapter 6

Analyzing LAST

This chapter provides an example analysis to demonstrate how LAST can be analysed in a structured and compositional manner. The analysis chosen is the traditional live variable analysis, which occurs just before the register allocation phase (Figure 6.1); the information generated by live variable analysis is crucial for the McCAT register allocator. First, a brief review of the analysis is given, after which the algorithm for determining live ranges on the LAST tree is presented. Finally, the third section provides an overview of the implementation details.

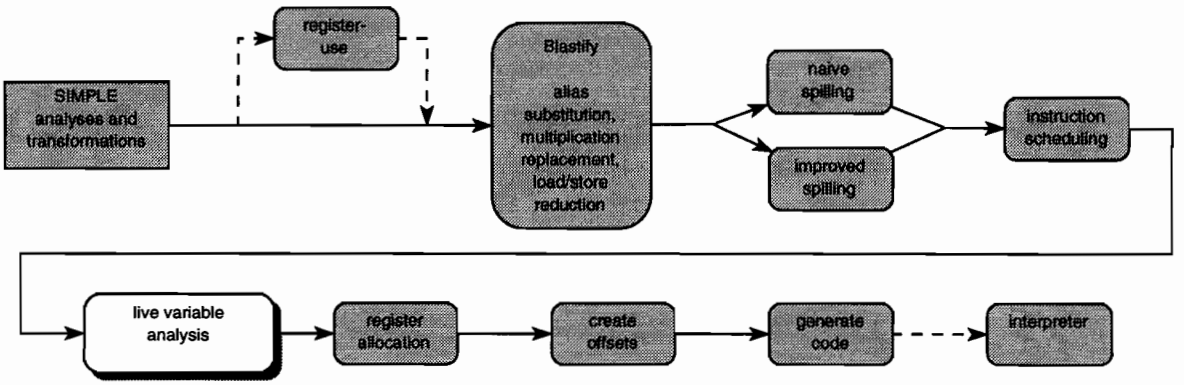


Figure 6.1: Live Variable analysis on LAST

6.1 Brief Review of Live Variable Analysis

Live variable analysis is a traditional flow analysis required by non-trivial register allocators such as those using the interference-graph coloring approach [CAC+81, Bri92, LJ92] or McCAT's cyclic interval graph approach [HGAM92]. Live variable analysis indicates when

a variable contains a value that will be used later on in the program *ie* that is alive—any such variable is an excellent candidate for being kept in a register. If, at a particular program point, a variable is never used again, or is redefined before being used, then it is considered dead.

The fragment of C code in Figure 6.2 provides a simple example. The left hand column is the C code, the middle column (inside the comments) is the line number, and the third column (again, inside the comments) are the variables that are alive at the end of the statement.

Variable a is defined[1] at line 1, used in line 3, defined again in line 6 and used finally in line 7. Notice that a is alive after the first and second statement but is dead in statements four and five, as it is redefined in the sixth statement. Since b and c are used at the very end, they are live through out the sequence of instructions. Variable a is redefined in statement six, and is used in statement seven, and so is live for the duration. The register allocator therefore knows that it is profitable to keep a in a register during statements 1, 2 and 3, and also 6 and 7, but not for 4 and 5.

```
main(){
                                    line #  vars alive
    a = 10;                          /* 1:  {a}     */
    b = 11;                          /* 2:  {a}     */
    c = a + b;                       /* 3:  {b}     */
    c = c + b;                       /* 4:  {b,c}   */
    d = b + 3;                       /* 5:  {b,c}   */
    a = c;                           /* 6:  {a,b,d} */
    printf("%d%d%d",a,b,d);          /* 7:  {}      */
}
```

Figure 6.2: Example of live variable analysis

The traversal of individual statements is illustrated in Figure 6.3, which shows an assignment statement a = a + b bracketed by program points: program point B preceding it, and A following. Since live variable analysis is a backward analysis, the traversal proceeds first through program point A before reaching the assignment statement, with flow information being stored at each statement. The flow information stored at the current statement (a = a + b) is the information that was generated after analysing program point A *ie* in a forward traversal of the program, the flow information associated with each statement records the state of the program just after the statement is processed.

After storing the flow information generated from program point B, the statement (a = a + b) is analysed. First, the left hand side is inspected. Variables on this side

[1] For the statement a = b + c we say a is *defined*, and b,c *used* [ASU88].

of a modify statement are defined, and so from this point backwards, the variable is considered dead (a is marked dead). Next, the right hand side of the assignment operator is processed. All variables on the right hand side will be used, so from this point these variables will be alive (a and b are marked alive). Note that the variable a is considered alive because although it is defined, the use of variable a precedes the definition. After the operands are processed, the statements previous (in a forward sense) to this are inspected (program point B) where the new, updated flow information will be stored before repeating the analysis process again.

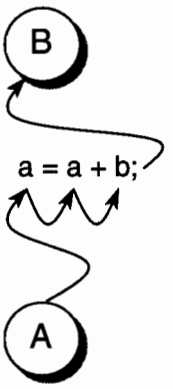


Figure 6.3: Detail of live variable analysis algorithm at the statement level

6.2 Live Variable Analysis in LAST

Since live variable analysis is relatively well-known, this section focuses on the different conceptual approach taken in LAST, compared that of traditional analyses utilizing control flow graphs. Specifically, in the analysis, the LAST traversal is aware of the structured nature of the program, rather than the traditional method of following the flow of control without recognizing the program's structure. The program is therefore treated as a series of hierarchically related conditional and unconditional bodies, as is presented in the following text.

For the purposes of elucidation, live variable analysis on some generic examples is presented, to provide an intuitive understanding of the traversal. Following the examples is an abridged version of the algorithm.

The examples utilize two control structures: the `if` and `while` statements. In addition, the three forms of 'constrained' `goto` statements (`return`, `continue` and `break`), which have an important impact on the analysis, are included in the `while` loop examples. In the diagrams, the letters "A" and "B" inside circles indicate program points, the rectangles with rounded edges sequences of instructions, the oval labeled "Merge" a merge operator,

and the solid arcs the direction of traversal of the analysis.

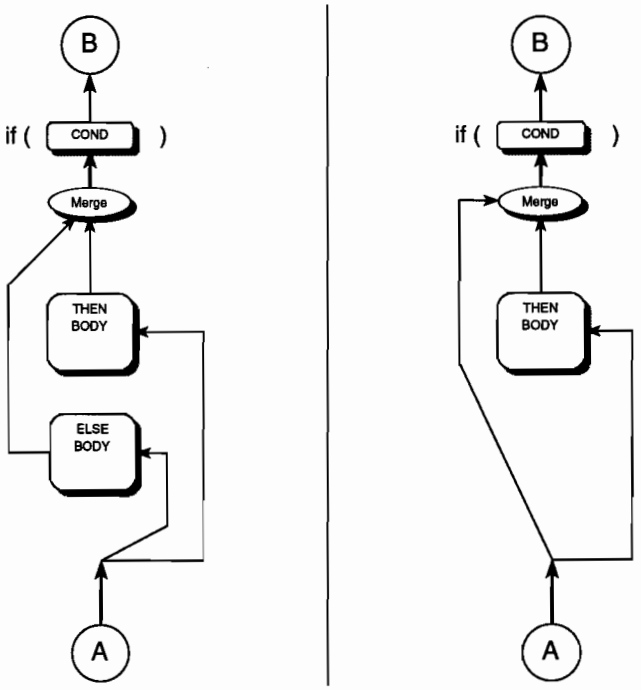


Figure 6.4: Detail of live variable analysis algorithm for an if statement

The first example is with an `if` statement, which illustrates the effect of a conditional body on the flow of the analysis through the program. Figure 6.4 depicts two `if` statements: on the left an `if` with both then and else bodies, and on the right an `if` with only a then body. Dealing with the `if` statement on the left, the diagram is equivalent to the code in Figure 6.5.

Since the analysis is backward, processing the `if` statement would start at its end (program point A), and process the then and else bodies independently. The results of these must be merged afterwards (merging using a union operator), and the result used

```
program point B
  if(cond)
    then body
  else
    else body
program point A
```

Figure 6.5: Pseudo code for live variable analysis example

when processing the conditional body of the `if` statement. The resulting information is then propagated to program point B.

For the second example, on the right, the traversal is similar, except that there is no else body to process. Therefore, the live information from program point A is merged with the information generated from processing the then body.

As is obvious, the traversal is straight forward, and essentially merges live variable analysis flow information from conditional bodies with that from previous instructions, and a single pass over the instructions is sufficient to gather the information. However, dealing with loops is slightly more complicated, requiring a fixed-point iterative solution. As a reminder, a fixed-point iterative solution means that the output data flow information of the loop is fed back into the loop, and the process repeated until the output flow information between two successive iterations is identical.[2]

The number of iterations for structured programs is at most two per loop block (intuitively, one iteration to correctly initialize the input for the next iteration, a second iteration to use the correct input). A `while` loop is used to demonstrate the processing of loops in LAST. Dashed lines indicate the flow of analysis information during the fixed-point iteration.

Figure 6.6 shows three `while` loops, each one including one of the three constrained gotos: `return`, `break` and `continue`. These are considered gotos because they change the flow of control, but are limited in the location of their destination, *ie* any `return` statements go to the end of a function body, `continue` to the beginning of the innermost surrounding loop, and `break` to the end (plus to the end of `switch` statements).

These constrained `goto` statements are included because they can alter the control flow of a program, and thus its flow information. As shown in the diagram, these statements are handled by recording the associated flow information with each statement at each hierarchy, and using this saved information to re-initialize the data flow input. The pseudo-code for the algorithm provides an explanation of the mechanics of handling `return`, `break` and `continue` statements.

The leftmost `while` loop in Figure 6.6 corresponds to a fragment of code such as in Figure 6.7. The flow information enters from point A, and is used as input into the loop body $S3$. The `return` statement, however, indicates the end of the function, and so its input is null. The null flow information is used to process the conditional body $S2$ preceding the `return` statement. At the top of the conditional, flow information from $S3$ and the conditional `return` block are merged, and used as input for the next block of code, $S1$, and the `while` condition. After the conditional body is processed, the flow information is then fed back into the bottom of the loop ($S3$) for the next iteration. When the flow information has converged, it is propagated to program point B.

The middle diagram of the `while` loop is identical to that on the left, except that a `break`

[2]Convergence is guaranteed. For the formal proof, refer to the dragon book [ASU88].

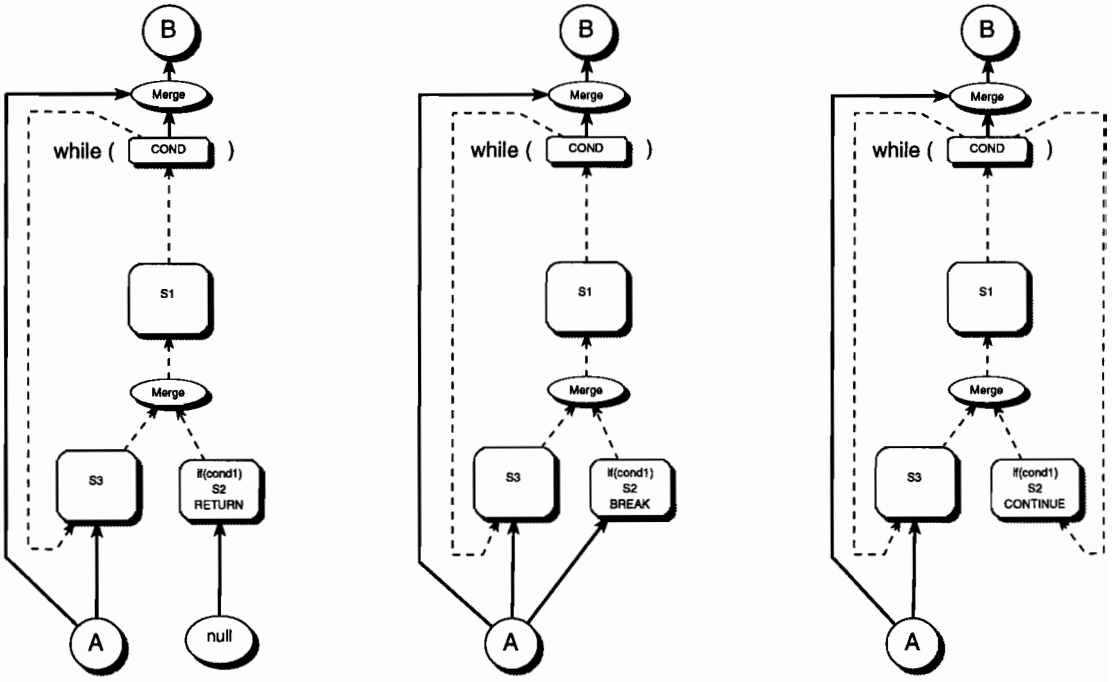


Figure 6.6: Three examples of live variable analysis algorithm for while statements

```
program point B
while(cond) {
  S1
  if(cond1){
    S2
    return;
  }
  S3
}
program point A
```

Figure 6.7: Structure of while loop containing a conditional return statement

statement replaces the `return`. Now, instead of the `return` statements' flow information being initialized from null, it's initialized from program point A, since a `break` will transfer control to just after the loop. Similarly, the third `while` loop on the right utilizes a `continue` statement, which is initialized from the flow output of each iteration.

Figure 6.8 presents an algorithm for generating and storing live variable information by traversing a LAST tree. It is complete except for function calls, which are handled identically to arithmetic operators, except that all registers holding parameters are marked Alive (this case was left out for brevity).

The algorithm determines the lifetime of variables inside of registers. The recursive function `live_analysis` takes four arguments: the node being analysed (node), the current flow information (info), the flow information associated with a break statement (BreakInfo), and the flow information for a `continue` statement (ContinueInfo).

If an assignment statement is analysed, the flow information is first stored with the node, and then variable being defined is marked as dead, and the right-hand side is analysed. This side will be a basic operator, and if they correspond to registers (*ie* are not constants), then the variables are marked as alive. Since load and store operators can take registers as operands (see Figure 4.7), they are marked as alive if being read (for the store), or dead if being redefined (for the load).

For control structures there are two groups: `if` or `switch`, and loops. For `if` and `switch` statements, their bodies must be processed in parallel, and afterwards the information merged. The condition expression is then analysed, and merged again with the flow information (see Figure 6.4). Since break statements behave differently in `switch` statements, the BreakInfo is set to the state of liveness just following the `switch` statement (*ie* assigned to info). As `live_analysis` is recursive, BreakInfo is reset to its previous value once the `switch` has been processed.

Loops are treated differently since their bodies are usually repeatedly entered, and the flow information from the previous iteration affects the lifetime of a variable (if it lives across iterations). Analysis of loop bodies is therefore performed twice: once to initialize the live variable information, and a second time to use the correct values in the analysis. Note that the ContinueInfo corresponds to the live information from the beginning of the loop, and so the ContinueInfo is set to the merged information after the first iteration. BreakInfo is set to info, as in the `switch` statement.

The three controlled `goto` statements (`continue`, `break` and `return`) are handled next. Since the analysis is intraprocedural, all variables are dead after a `return` statement. For `break` statements, flow of control is to either the end of a loop, or the end of a `switch` statement, so the BreakInfo is returned. Similarly, for `continue`, ContinueInfo is returned.

Finally, since the analysis is a backwards one, `live_analysis` moves to examine the previous LAST node.

```
/* node is the current program node, info the live variable information passed from the previous
 * statement. BreakInfo contains flow information used to initialize the flow information when
 * a break is encountered, and ContinueInfo for the continue statement. */
LiveInfo live_analysis(AstNode node, LiveInfo info, LiveInfo BreakInfo, LiveInfo ContinueInfo)
{  node = GotoEndofSequence(node);      /* go to end of list */
   switch (StatementType(node)) {
   case assignment:
      node.info = info;                 /* store flow information */
      info[left_child(node)] = Dead;    /* kill the assigned variable */
      return live_analysis(right_child(node),info, BreakInfo, ContinueInfo);
   case load:
      info[destination(node)] = Dead;
      if (source(node) == REGISTER) info[source(node)] = Alive; /* ignore constants */
      return info;
   case store:
      info[source(node)] = Alive;
      if (destination(node) == REGISTER) info[destination(node)] = Alive;
      return info;
   case basic_operator:   /* includes arithmetic, logical and conversion operators */
      if (left_child(node) == REGISTER info[left_child(node)] = Alive;
      if (right_child(node) == REGISTER info[right_child(node)] = Alive;
      return info;
   case control_structure:
      if (structure == IF or SWITCH) {
         if (structure == SWITCH) BreakInfo = info; /* break reset for switch statement */
         foreach subbody in node
            subbody.info = live_analysis(node.subbody, info, BreakInfo, ContinueInfo);
         merged_body_info = merge_all_subbodies(node);
         condition_info = live_analysis(conditional(node), info, BreakInfo, ContinueInfo);
         merged_info = merge_info(merged_body_info, condition_info);
   }
   else {
      merged_info = info; /* initialize to info */
      do twice                      /* is a loop, so process twice */
      {  body_info = live_analysis(body(node), merged_info, info, merged_info);
         condition_info = live_analysis(conditional(node), merged_info, info, merged_info);
         merged_info = merge_info(body_info,condition_info);
      }
   }
   /* the above calculated flow information through a control structure. This information
    * must now be merged with information that did not flow through the control structure */
   return merge_info(info, merged_info);

   case RETURN: return EMPTY;   /* analysis is intraprocedural, so everything is dead */
   case BREAK: return BreakInfo;
   case CONTINUE: return ContinueInfo;
   }
   return live_analysis(previous(node),info, BreakInfo, ContinueInfo); /* go backwards */
}
```

Figure 6.8: High-level algorithm for performing live variable analysis on LAST

Chapter 7

Retargeting McCAT

As mentioned previously, there is a considerable software investment in an optimizing compiler, and retargeting is a natural way to amortize this investment. The objective, of course, is to make retargeting the compiler cheaper than actually rewriting the analyses and transformations that motivate the retargeting in the first place. This can be a challenging objective, as there is a trade-off between generating efficient code and simplifying retargeting, and there is little point in retargeting an inefficient compiler.

The McCAT retargeting strategy is to restrict the types of machines targeted, and within this set, abstract the architectural details that are common across all the machines. These features are then explicitly represented in LAST where the various algorithms can examine and manipulate them. Since an abstract machine is modeled, the algorithms can be reused for all the new targets—they do not need to be rewritten. In addition, since all the complexity is contained in LAST, the code generation phase is relatively easy to write. The code generation phase is where LAST is mapped onto assembly instructions and printed to a file, and consists mostly of traversing LAST, recognizing subtrees of nodes as a particular operation, and printing the corresponding assembly instruction(s).

7.1 Architectural Classes

Even though LAST is restricted to RISC architectures, there are still sufficiently different approaches to designing them which can result in sometimes radically different features. Recognizing these features can often mean a significant difference in performance, and so warrants different configurations of LAST. LAST therefore supports different classes of architectures. They are described in Section 3.2.4, but as a brief reminder are: register windows (or not), explicit (implicit) condition codes, multiplication/division in floating point registers (or not), and support for high-level operations such as structure copies, exclusive or and negation operators.

When compiling for a specific machine, the architectural classes are selected, and the options toggled on or off (see Figure 7.1). Then, during the blastify phases where LAST is generated from SIMPLE, the appropriate set of LAST nodes are created.

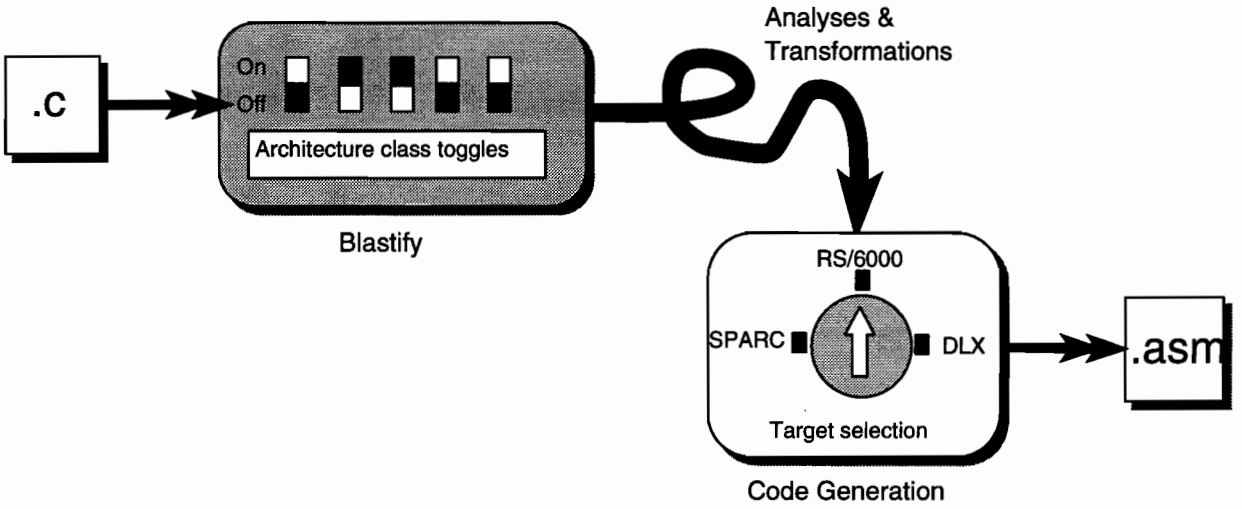


Figure 7.1: Selection of Architectural Classes

The analyses and transformations in LAST operate on all the different configurations—they must, since the classes are orthogonal and so any combination is possible. For instance, whether or not a machine supports exclusive or operations is orthogonal to the type of condition codes supported. Therefore, the algorithms can be used on any architecture without being rewritten. There is, however, the issue of specific architectural features that are found on only certain RISC machines. These features can require sophisticated analyses to identify. The RS/6000 is an excellent example: it has a special multiply-accumulate instruction that consumes only three machine cycles [OMMN90] that can significantly improve floating-point performance of some programs. Since this hardware feature is currently found only on RS/6000 architectures, an algorithm that exploits it is of limited usefulness, and should be implemented as a separate phase. One has the option of placing the phase in either the code generation module, or intermixed with other compiler phases—either are feasible approaches, but it is conceptionally simpler to implement it as another transformation phase, rather than being part of the code generation module.

7.2 Code-Generator Generator

The code generation phase is where the abstract meets the concrete: LAST subtrees are mapped to appropriate assembly instructions. This module is meant to be completely

rewritten when McCAT is retargeted, and so is designed to be as simple as possible. The code generation phase actually consists of two phases: calculation of offsets, and generation of assembly code (see Figure 7.2). The offsets module calculates the offsets for local and parameter variables, and allocates/deallocates temporary storage for structures passed as parameters. In addition, it detects variables marked as existing only in registers, and does not allocate space for them (see Section 3.4.4 for a brief description of this optimization).

The code generation phase handles the mapping of LAST subtrees to assembly instruction(s). The actual instruction set used to specify the program varies from machine to machine, and it is the job of the code generation phase to map LAST to the appropriate assembly format. For example, there is only one representation in LAST for an addition instruction, but the machine could require the addition be specified as "`add` *result*, *op*1, *op*2", "`add` *op*1, *op*2, *result*", "*result* `add` *op*1, *op*2", or some other variation. This encoding, and other 'nitty-gritty' details are hidden in the code-generation module.

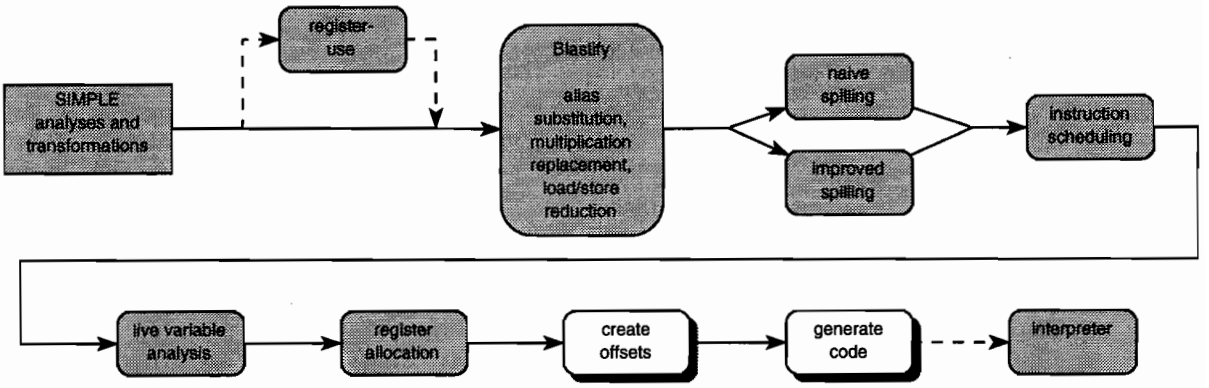


Figure 7.2: Code Generation Phase

Isolating the actual mapping of instructions simplifies retargeting by limiting the files requiring modification, and by keeping the complexity in LAST, simplifies changing the module.

To simplify the code generation phase, a code-generator generator is used. A code-generator generator, as the name implies, takes the description of a target machine and creates a code generator. Code-generator generators are useful and powerful tools used to simplify the porting of compilers [AGT89, FHP92b, ESL89]. They provide a way of specifying a pattern to match, and associated 'actions' to perform (usually being the printing of assembly code instructions to a file). The generator then creates a program that automatically will traverse the data structure provided as input, and perform the matching of patterns specified.

McCAT utilizes the code-generator generator BURG[FHP92b] (and its more powerful relative IBURG[FHP92a]) to automate the monotonous task of creating code generators for

the various target architectures.

As high-level input, BURG takes IR tree patterns, associated 'actions' (code templates), and a cost. This cost is calculated by the programmer depending on the relative 'expense' of one instruction to another. BURG traverses the IR, looking for the largest pattern it can match with the cheapest cost (called a cover), using dynamic programming [ASU88, CLR92]. When a pattern is matched, the associated action is performed *ie* the code template is emitted. For example, BURG might be handed the pattern `ASSIGN(REG,PLUS(REG,REG))` (corresponding to the LAST subtree generated for the C code `x = y + z`) and action `add REG1,REG2,REG3`. When the pattern is matched, the string "`add r1,r2,r3`" is printed (assuming the register allocator has allocated r1, r2 and r3 to x, y and z respectively).

The patterns matched are limited to individual subtrees (with SEQ nodes as the root), but within these subtrees BURG guarantees an optimal cover. This property is crucial for CISC architectures, but under-utilized for RISC, and indeed a great portion of BURG's potential goes untapped. However, BURG allows the target machine to be specified in concise, high-level templates, and provides the traversal mechanism for LAST, and it is these features that make BURG so useful.

A 'user-friendly' interface was written for BURG, called MCBURG[Don92], and is currently used for three target machines (DLX, SPARC and RS/6000) plus the pseudo assembly. Figure 7.3 illustrates a fragment of the MCBURG specification used for the add instruction in DLX. There are three patterns specified: "`REG`", "`PLUS(reg,reg)`" and "`MODIFY(reg,reg)`". The first pattern is for a leaf register (REG) node, and it simply prints out the register. The function `emit` knows the correct format for the target architecture, and takes a pointer `t` to the REG node, where the register mapping has been stored.

`MODIFY(reg,reg)` expects two `reg` subtrees (which reduce in this fragment to `REG` and `PLUS(reg,reg)`) the left child being the assignment destination, and the right child being the plus operator. The action saves a pointer to the destination, and then descends into the operand. This rather inelegant arrangement is used since LAST uses an infix notation, rather than a pre- or postfix notation *ie* in an assignment operation, the plus node is the right child of an assignment node, and so the assignment destination is unavailable when processing the plus node (there are no pointers back to parent nodes). Assembly instructions usually use prefix notation, and so the use of `modifykids` is a way of converting the infix form of LAST to the prefix form.

The next pattern, `PLUS(reg,reg)`, expects a plus node with two register children. First, it emits the string `add`, and then descends subtrees pointed to by three variables: `modifykids, kids[0]` and `kids[1]`. The variable `modifykids` is a global variable and is set by the MODIFY pattern,

The entire MCBURG specification for the various target machines consist of similar patterns for all the possible subtree combinations in LAST(which number under 100 for

```
/*************************************************************************/
/* Since operators such as PLUS are children of MODIFY, I must remember
   the destination for when I bump into the operator                     */
/*************************************************************************/
stm: MODIFY(reg,reg) TOPDOWN;{
        /*remember what I was looking at */
        modifykids= kids[0];
        modifynts= nts[0];
        /* go down right child */
        reduce(kids[1], nts[1]);
/*************************************************************************/
/* just print out the name of the register                               */
/*************************************************************************/
reg:    REG ;{
            emit("%R",t);
            }
/*************************************************************************/
/* print out code for add first, then the children (left then right) */
/*************************************************************************/
reg:    PLUS(reg,reg) TOPDOWN = (2);{
        emit("add ");
        reduce(modifykids,modifynts); /* descend assignment dest. */
        emit(",");
        reduce(kids[0], nts[0]);      /* descend left child */
        emit(",");
        reduce(kids[1], nts[1]);      /* descend right child */
    }
```

Figure 7.3: Sample McBURG specification

DLX), a relatively easy task. However the proof, as they say, is in the pudding, and the success of this retargetability strategy will be tested as McCAT is retargeted to more and more RISC architectures, although the initial retargeting to the SPARC and RS/6000 is very promising. A basic working version of McCAT was targeted to each of these machines, in both cases, in under one month by one person working part time, while learning both the architecture and LAST. It is expected that once experienced in re-targeting LAST, subsequent new RISC machines will be more easily accommodated.

Chapter 8

Results

8.1 Description of Benchmarks & Test Strategy

To illustrate the utility and effectiveness of the example transformations performed on LAST, the results from a small collection of benchmarks is presented. These benchmarks are described in Table 8.1, with the second column, # SIMPLE statements, indicating the number of SIMPLE C statements in each program. Each benchmark has been stripped of comments, and all blank lines removed.

The benchmarks were compiled for the DLX architecture [HP90], and their execution simulated using the DLXSim simulator [HM90]. DLXSim can report individual instruction counts. However, for the purposes of these experiments we concentrate on the number of loads, stores and total cycles consumed.

The DLX architecture, as mentioned before, is a RISC architecture based partly on the MIPS R2000. The simulator configuration used is the default, with Table 8.2 listing the various latencies for the floating-point and load/store functional units. All other operations are single cycle. DLXSim does not simulate a cache.

Each benchmark was compiled six different ways, twice with DLXCC, and four times with McCAT. DLXCC is a 1.37.1 version of GCC ported to DLX. The DLXCC-compiled benchmarks were compiled with and without the `-O` option. The latter is used as a base for comparison—the results of all the other benchmarks are normalized to its results. While the McCAT results are reasonable, in general they do not perform as well as the optimized DLXCC versions. The reason is that DLXCC implements many other optimizations such as constant sub-expression elimination, strength reduction, jump optimization (which is needed, especially after a `goto`-filled program is restructured), dead-code elimination, and scheduling of conditional branch delay slots. The only optimizations currently performed by McCAT are the load/store reduction, register allocation and instruction scheduling.

The four variations for the McCAT compiled benchmarks use two of the load/store available algorithms: improved (keep all local variables in registers) and extended (keep

Benchmark	# SIMPLE statements	Description
Dhrystone	440	The well known synthetic benchmark Dhrystone, inspired by Whetstone. Attempts to characterize CPU and compiler performance for a typical program. Outermost loop performs only one iteration due to the slow simulation speed of the simulator.
Hanoi	25	A recursive solution to the Towers of Hanoi problem.
Intmm	105	A 40×40 integer matrix multiply.
Knight	93	A recursive solution to the Knight's Tour problem: given an 8 by 8 chessboard, determine a series of moves so that a knight starting at position 1,1 visits each square without repetition.
Mersenne	248	Computes the digits of $M = 2^{p-1}$, where p is set to 89.
Sorts	226	Generates a random sequence of 100 integers and applies both bubble- and quicksort algorithms.
Tomcatv	567	C version of the FORTRAN program Tomcatv. This program is a highly vectorizable double precision floating point mesh generating benchmark. The FORTRAN version is part of the SPEC benchmark suite.
Whetstone	1715	A synthetic benchmark, based on the frequency of Algol statements in programs submitted to a university batch operating system in the early 1970s.

Table 8.1: Description of benchmarks

Unit	Latency (cycles)
Add/Subtract	2
Divide	19
Multiply	5
Load	2
Store	1

Table 8.2: Latencies for DLX floating-point and load/store functional units

aliased and global variables in registers within basic blocks). The naive approach generates inferior code, and for brevity is left out. For each load/store reduction transformation, an unscheduled and scheduled (using the Shieh-Papachristou algorithm) is performed. Table 8.3 lists the meanings of the abbreviations used in the following graphs.

Abbrev.	Meaning
OptDLX %	Percentage improvement for optimized DLXCC (dlxcc -O)
Impr. %	%age improvement for McCAT using improved load/store reduction
ImprSch. %	as above but with scheduling
Ext. %	%age improvement for McCAT using extended load/store reduction
ExtSch. %	as above but with scheduling

Table 8.3: Explanation of abbreviations for results

Note that in the result figures, the vertical scale is typically from 0 to 1.0, with 1.0 representing 100%.

8.2 Benchmark Results

8.2.1 Dhrystone

Figure 8.1 shows the results for the dhrystone benchmark. The results are quite good, and McCAT actually performs better than the optimized version of DLXCC when using the 'improved' load/store algorithm. Obviously, loads and stores constitute a large proportion of the execution time of the benchmark. Interestingly, the 'extended' algorithm performs worse than either the DLXCC or 'improved' versions. The transformation was, in a sense, too successful, since it found many global and aliased variables to keep in registers. However, the register pressure was too great, causing the the register allocator to spill and reload variables enough times to slow the overall execution of dhrystone. When the current register allocator spills a variable, loads and stores are inserted for all uses of the variable within the function, not just at the site of high register pressure. DLXCC has a more effective strategy.

8.2.2 Hanoi

Hanoi (Figure 8.2) is another benchmark where McCAT produced better code than DLXCC (*ie* another program with a significant number of dynamic loads). There are very few global or pointer dereferences, so the results are the same for the improved and extended versions.

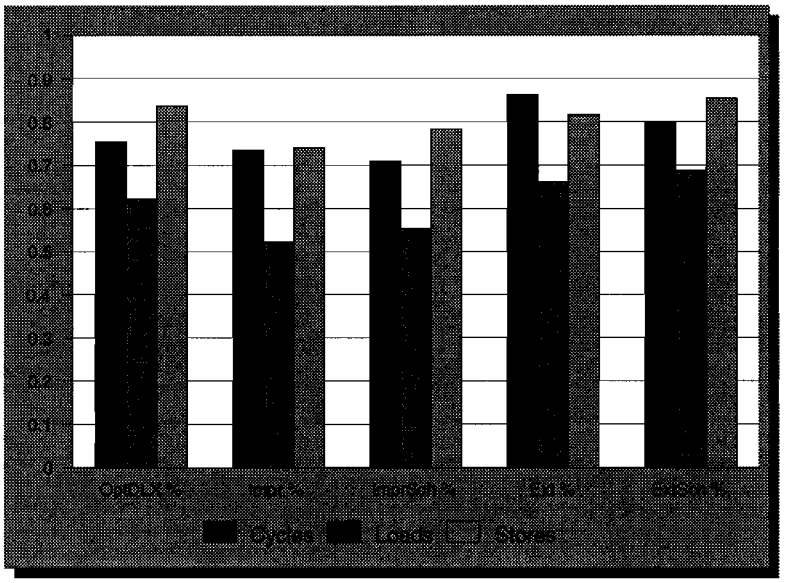


Figure 8.1: Dhrystone results

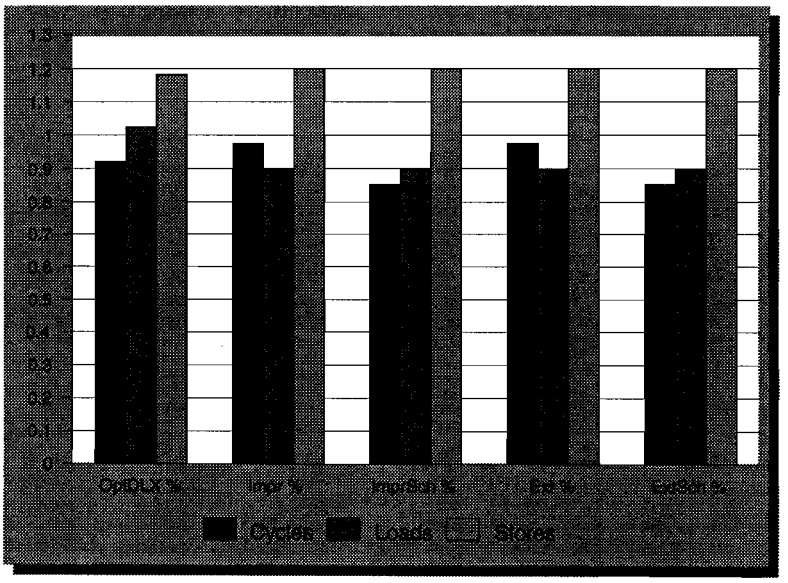


Figure 8.2: Hanoi results

8.2.3 Intmm

While McCAT does not perform as well as DLXCC on the intmm benchmark (Figure 8.3), the load/store transformations by themselves enable a speed-up of 30%. Scheduling can reduce it by a further 5%, although McCAT will definitely benefit from the addition of other traditional optimizations.

As in the dhrystone benchmark, high register pressure causes the extended version of McCAT to perform worse than the unscheduled improved version.

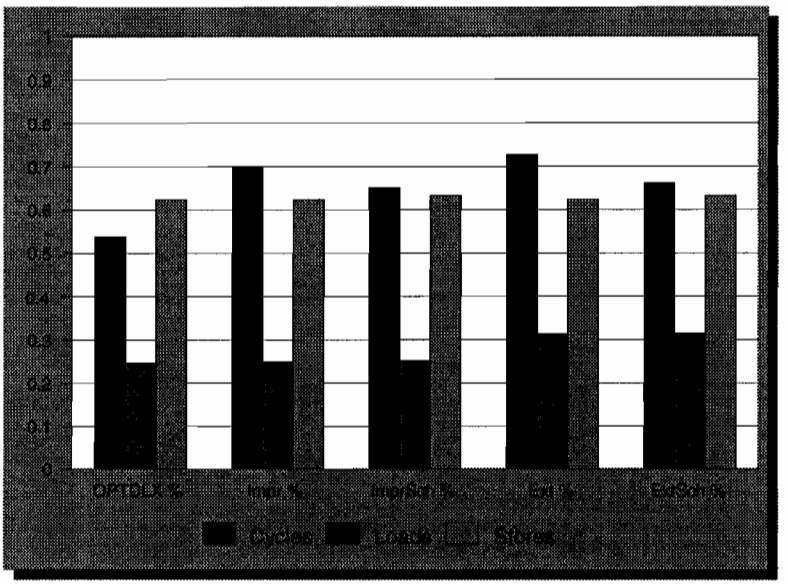


Figure 8.3: Intmm results

8.2.4 Knight

In the knight benchmark (Figure 8.4), the improved load/store reduction algorithm outperforms that of DLXCC, although the lack of other optimizations keeps the McCAT version about 15% slower than the optimized DLXCC version. There are neither pointers nor globals, so there is no difference between the improved and extended versions. Additionally, since knight's basic blocks are small, instruction scheduling makes only a minor improvement to the program's running time.

8.2.5 Mersenne

Figure 8.5 presents the results for the mersenne benchmark. Like hanoi, there are few globals and pointer dereferences, and like intmm, the McCAT versions are up to 30% faster than

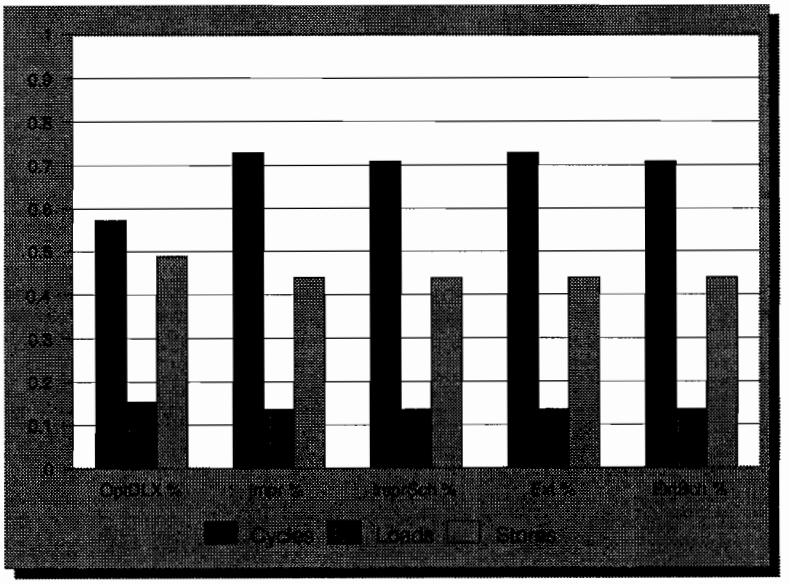


Figure 8.4: Knight results

the unoptimized version. Again, there is room for further significant improvements with other optimizations, when compared with the optimized DLXCC results.

Additionally, there is a spectacular reduction in the number of loads for both DLXCC and McCAT. The majority of the savings come from moving the loading of array base addresses to the outside of loops (see Section 3.4.2).

8.2.6 Sorts

The sorts benchmark (Figure 8.6) provides the first example of a significant gain from the extended load/store reduction algorithm. The improved version allows only a 10% improvement in performance, but the extended version is close to 30%. There is, again, an illustration of the need for other optimizations when compared to the optimized DLXCC result. The number of loads in the optimized DLXCC version is quite low, not only because of its own load/store reduction algorithm, but also the effect of other optimizations that reduce the total number of instructions generated for the benchmark.

8.2.7 Tomcatv

The results for the tomcatv benchmark highlight one area of needed improvement: register allocation. This is the only benchmark where a scheduled version performs worse than an unscheduled version (Figure 8.7). Tomcatv has very large basic blocks that, even in unscheduled versions, place pressure on the register allocator. The Shieh-Papachristou

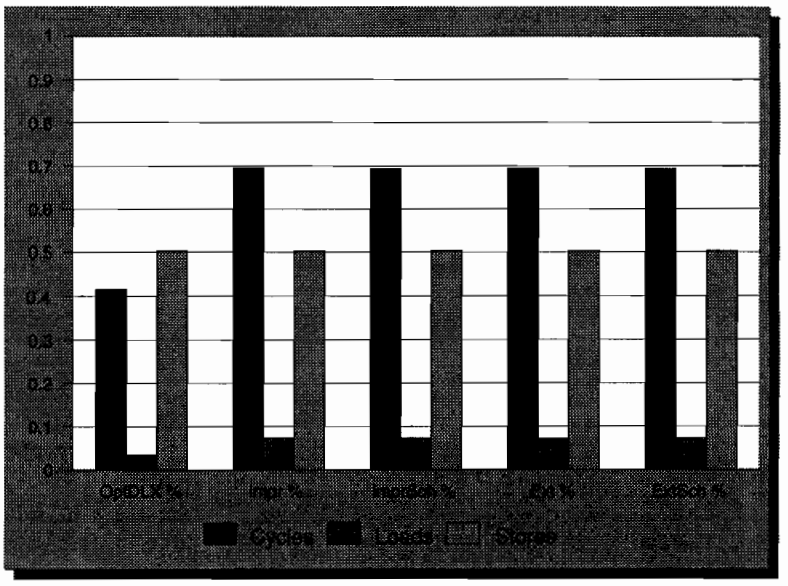


Figure 8.5: Mersenne results

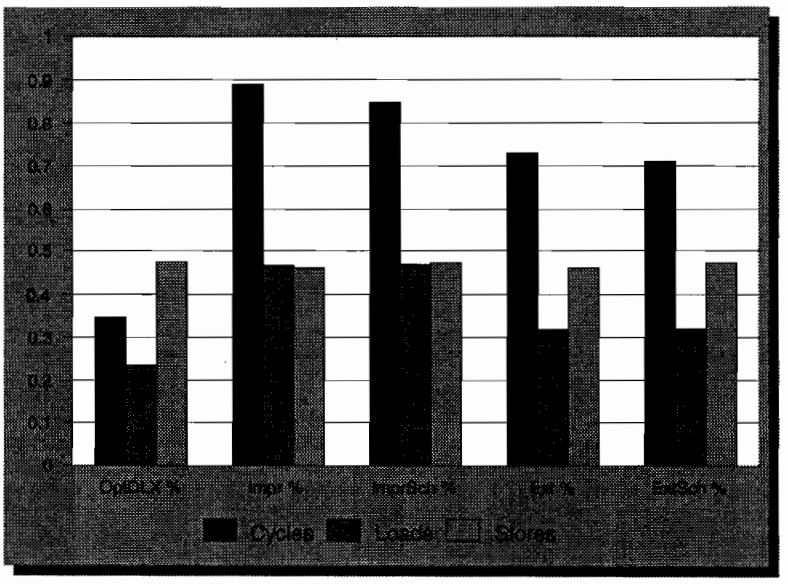


Figure 8.6: Sorts results

algorithm used in McCAT stretches the lifetimes of variables enormously in this benchmark, resulting in an avalanche of register spills and reloads. The McCAT extended version reduces the number of loads and stores, and thus reduces the restrictions on the scheduler, producing a highly parallel, but register intensive program.

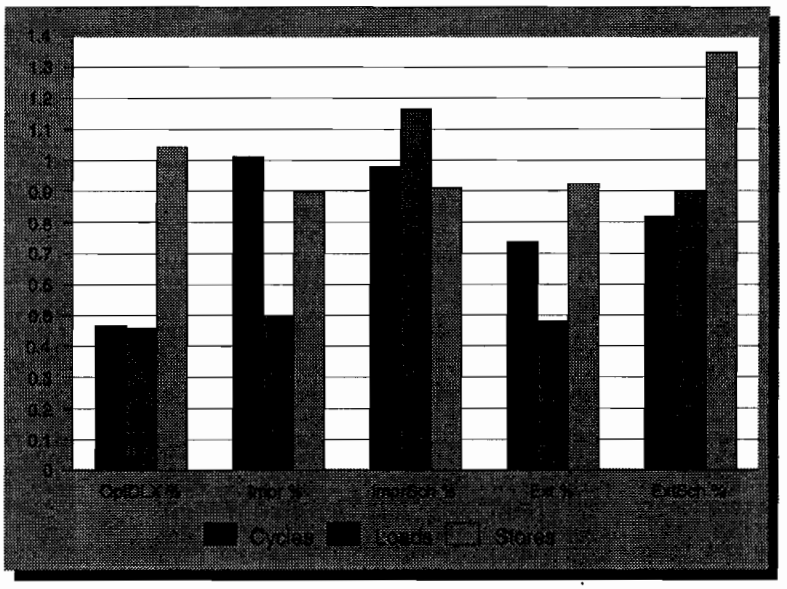


Figure 8.7: Tomcatv results

8.2.8 Whetstone

Both the improved and extended load/store reduction algorithm substantially reduces running time in whetstone (Figure 8.8), by around 40%. DLXCC, again, produces a faster program (over 20% faster), indicating the many further improvements possible in McCAT. Scheduling slightly improves running time, but in conjunction with the extended load/store algorithm, increases the register pressure to a point where register spills and reloads increase running time past that of the improved scheduled version.

8.3 Observations & Impressions

As much background reading as one may perform, research inherently requires a certain leap of faith and of risk taking. Hindsight enables 20/20 vision, and in retrospect while some design decisions may have been excellent choices, others might have been better. This section lists some of the author's impressions on the development of LAST.

- As in many software projects, the amount of testing needed was underestimated. The author initially tried to split his time about equally between development and testing,

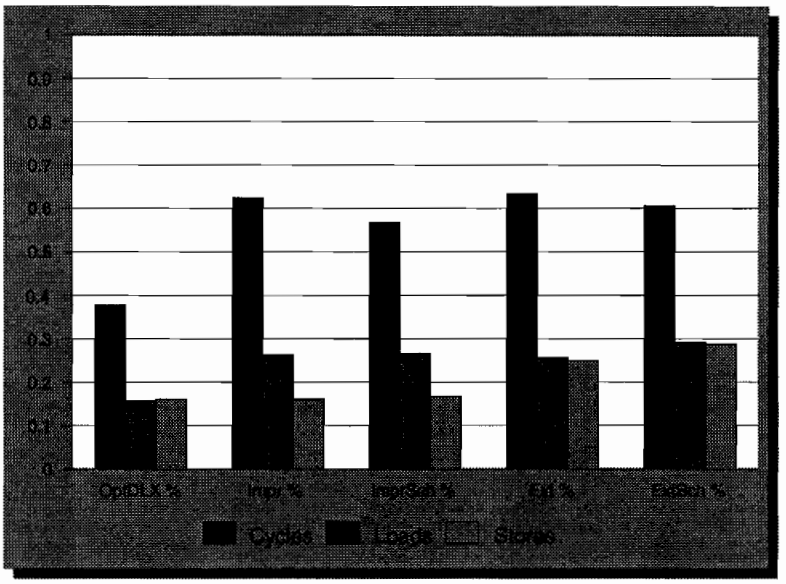


Figure 8.8: Whetstone results

but in the final stages, much more time was required for testing. Above everything else, compilers must be correct, and due their complexity, benchmarks are the only way to efficiently ensure correctness.

In addition, it became apparent that the worst person to create test cases for a compiler is a compiler-writer. Fortunately, amazing permutations of C code are not uncommon in publicly available C source code, and were invaluable for testing LAST.

- Limiting the abstract machine to RISC was an effective decision, and allowed LAST to be quite low level while maintaining retargetability. Low-level optimizations, as well as bug-fixes, were instantly available for all supported targets.

- The interpreter was extremely helpful in debugging LAST. It allows a direct, machine-independent way of manipulating LAST, along with some debugging capability. The current version of the McCAT backend does not support debugging information, and so complicated benchmarks that failed were very challenging to debug (before the interpreter). In one case, the author spent six days on a benchmark, while the interpreter was being developed. Within three hours of it being available, the problem was located and solved.

- The current method of retargeting, while simple, requires a fair knowledge of LAST, and represents a reasonable investment in time. An interface to the backend, and of configuring LAST(perhaps similar to the machine description files used in GCC), would be very useful.

Chapter 9

Conclusions & Future Work

9.1 Conclusions

This thesis has presented LAST, a low-level structured intermediate representation used in the McCAT C compiler. LAST is designed to expose low-level architectural details to various code-improving algorithms, while retaining a high degree of retargetability.

The structured approach simplifies the development of analysis and transformation phases, as evidenced by the simple but powerful algorithms used for live variable analysis, and the reduction of the number of loads and stores. The reduction algorithm alone improves the performance of several benchmarks by up to 30%. In addition, the structured nature of LAST easily supports pervasive flow information, enabling the points-to information gathered and stored in the SIMPLE IR to be used in LAST to reduce loads and stores associated with global and aliased variables. The pervasive flow information has enabled an additional performance improvement in the tomcatv benchmark of 20%.

There is, however, an 'over-optimization' effect in some benchmarks, where too many variables are held in registers, causing the register allocator to spill and reload the variables on every use. These register reloads retard overall performance considerably in some benchmarks, and point to the need for additional information to locate and isolate regions of high register pressure, so the allocator spills variables in these regions only, rather than at every use and definition.

LAST also enables McCAT to be retargeted to other RISC machines without unduly compromising the quality of code produced. This was a non-trivial goal—there is an inherent trade-off between high-performance and ease of retargeting. Abstraction minimizes the changes required when retargeting the compiler, but gives the code-improving transformations less information to work with, often resulting in less efficient code. On the other hand, the more specific the compiler, the more work is required to rewrite the analysis and transformation phases when retargeting to a different machines. LAST attains both goals by limiting McCAT's targets to RISC architectures, and abstracting features common to

them.

Since LAST is focused towards RISC machines and is configurable, it can represent low-level architectural details such as branch delay slots and register windows, allowing powerful low-level transformations (like register allocation and instruction scheduling) to be highly effective. At the same time, since only the configuration of LAST changes from machine to machine, all the analyses and transformations on LAST can be reused, and only the MCBURG specification file need be rewritten. Currently MCCAT is being retargeted to the SPARC, Alpha and RS/6000 architectures, and in all cases a basic, working version was available within one month, including the time taken to learn SIMPLE, LAST and MCBURG.

9.2 Future Work

The work done in this thesis has laid the foundation for the McCAT backend by providing both an intermediate representation and a code generator. Additionally, it provides important analyses and code-improving transformations. Based on this foundation, there remain many important issues to address, the most important of which are discussed below.

- Accurate information needs to be generated for the register allocator. Initially, it was assumed that the register allocator was sophisticated enough to handle the additional pressure placed by the 'extended' algorithm, but there are serious limitations, especially when other life-extending transformations (*eg* instruction scheduling) are used.

- Jump optimizations such as branch chain elimination need to be implemented. McCAT's program structurer can sometimes create structured programs that are not as fast as their unstructured counterparts. While the penalty generally appears to be in the region of 5% for programs requiring structuring, some benchmarks are penalized even further [EH94]. Structuring will typically produce a cascade of test conditions, using sequences of `if`, `do while` and `break` statements to replicate the flow control of a `goto`. Branch chain elimination would bypass all the intermediate condition checks by specifying the ultimate target label for a branch (such as for a `break`). The LAST nodes used to represent a program would not change: the only apparent difference would be at code generation time, when printing out the label for a `break` statement. At first glance, this optimization seems to create an unstructured program out of a structured one, but in fact does not. The change in control flow is not arbitrary, but essentially follows the normal flow without performing the tests. In following the flow of control of the cascading tests, there are no side effects *ie* values are only read, not written, and so the succession of jumps can be safely shortened. If there were intervening instructions between the conditions, then the branch chain elimination would not be performed.

- Larger, longer running benchmarks should be used to test McCAT, in order to further investigate the optimizations, and also to ensure the correctness of LAST generated code.
- The scheduling algorithm should be extended, to include
  1. Scheduling instructions inside conditional branch delay slots. This optimization can improve a program's performance by an additional 10% [Tie89].
  2. Minimizing register pressure by scheduling with knowledge of register usage. The balanced scheduling algorithm, by Kerns and Eggers, [KE93] can be a candidate algorithm for implementation.
  3. Utilizing points-to information to reduce various data dependencies between aliased variables while scheduling. Currently, a load or store of any aliased variable will have a dependency edge to all other loads and stores of aliased variables. The may points-to information (a list of variables that may be aliased to one another) gives highly precise information [EGH94], so the number of dependencies between loads and stores can be reduced significantly.
- Implementation of traditional optimizations, either at the LAST level, or at SIMPLE. Of immediate benefit would be common sub-expression elimination and strength reduction, as evidenced by the generally superior performance of the optimized DLXCC generated code.

Glossary

ACAPS: Advanced Compilers, Architectures and Parallel Systems, a research group at McGill university.

AST: Abstract syntax tree.

CFG: Control flow graph.

CISC: Complex Instruction Set Computer.

DAG: Directed acyclic graph.

DLX: Fictious instruction set based on MIPS R2000 architecture. Created as a pedagogical tool in Hennessy and Patterson [HP90].

FIRST: An abstract syntax tree generated by GNU's GCC front-end, and used to create SIMPLE.

GCC: The GNU C compiler, a highly portable, publicly available C compiler.

GNU: *G*nu's *N*ot *U*NIX. An organization associated with the Free Software Foundation in Massachusetts, Boston, devoted to providing free software to the programming community.

IR: Intermediate representation. Also known as intermediate code (IC) and intermediate language (IL).

LAST: Low level Abstract Syntax Tree, a low-level intermediate representation used in the McCAT compiler.

MIPS: Company designing RISC architectures and manufacturing UNIX-based workstations. The MIPS series of architectures includes the R2000 and R3000.

McCAT: McGill Compiler/Architecture Testbed. A system used for researching advanced compiler and architecture concepts, developed by the ACAPS group. Included in the the testbed is the McCAT C compiler.

RISC: Reduced Instruction Set Computer.

RTL: Register Transfer Language (GNU). A common acronym else, also used by Davidson to mean Register Transfer Lists.

SEQ: Sequence node, a LAST structural node (Section 4.1).

SIMPLE: Simple intermediate representation. A simplified abstract syntax tree used as the high-level IR in the McCAT C compiler.

SPARC: *S*calable *P*rocessor *ARC*hitecture. The architectural specification of a RISC chip used in modern Sun workstations. SPARC architectures utilize register windows.

Sun: A manufacturer of UNIX-based workstations.

Appendix A

SIMPLE Grammar

The grammar for SIMPLE is presented below, as implemented in the McCAT C compiler. For a more complete coverage of SIMPLE, the reader is directed to Bhama Sridharan's thesis [Sri92].

```
all_stmts : stmtlist stop_stmt
          | stmtlist

stmtlist : stmtlist stmt
         | stmt

stmt : compstmt
     | expr ';'
     | IF '(' condexpr ')' stmt
     | IF '(' condexpr ')' stmt ELSE stmt
     | WHILE '(' condexpr ')' stmt
     | DO stmt WHILE '(' condexpr ')'
     | FOR '('exprseq ';' condexprseq ';'exprseq ')' stmt
     | SWITCH '(' val ')' casestmts
     | ';'

compsmt : '{' all_stmts '}'
        | '{' '}'
        | '{' decls all_stmts '}'
        | '{' decls '}'

/** decls denotes all possible C declarations. The only difference is that**/
/** the declarations are not allowed to have initializations in them.     **/

condexprseq : exprseq ',' condexpr
                      | condexpr

exprseq : exprseq ',' expr
        | expr
        | condexpr '?'exprseq':'exprseq
```

```
stop_stmt : BREAK ';'
          | CONTINUE ';'
          | RETURN ';'
          | RETURN val ';'
          | RETURN '(' val ')' ';'

casestmts : '{' cases default'}'
          | ';'
          | '{' '}'

cases : cases  case
      | case

case  : CASE INT_CONST':' stmtlist stop_stmt

default : DEFAULT ':' stmtlist stop_stmt

expr : rhs
     | modify_expr

call_expr : ID '(' arglist ')'

arglist : arglist ',' val
        | val
        |

modify_expr : varname '=' rhs
            | '*' ID '=' rhs

rhs : binary_expr
    | unary_expr

unary_expr : simp_expr
           | '*' ID
           | '&' varname
           |call_expr
           | unop val
           | '(' cast ')' varname

/** cast here stands for all valid C typecasts      **/

binary_expr : val binop val

unop : '+'
     | '-'
     | '!'
     | '~'
```

```
binop :  relop
      | '-' | '+' | '/' | '*' | '%'
      | '&' | '|' | '<<' | '>>' | '^'

relop : '<' | '<=' | '>' | '>=' | '==' | '!='

condexpr : val
         | val relop val

simp_expr : varname
          | INT_CONST
          | FLOAT_CONST
          | STRING_CONST

val : ID
    | CONST

varname : arrayref
        | compref
        | ID

arrayref : ID reflist

reflist : '[' val ']'
        | reflist '[' val ']'

idlist : idlist '.' ID
       | ID

compref : '(' '*' ID ')' '.' idlist
        | idlist
```

Appendix B

LAST Grammar

This appendix gives the grammar for LAST.

```
function : save sequence restore EOSEQ

save : SEQ SAVE_REGISTERS
restore : SEQ RESTORE_REGISTERS

sequence : SEQ seq_body sequence
 | /* empty */

termsequence : SEQ BEGIN-BODY sequence EOSEQ

seq_body : modify
         | load
         | store
         | fn_call
         | conversions
         | reg_moves
         | comparisons
         | control
         | structured_goto
         | REG_WIN_OUT REG
         | REG_WIN_IN REG
         | ADJUST_SP
         | POP_PARMS
         | PASS_PARM REG


modify : MODIFY REG mod_src

mod_src  : REG
         | CONSTANT
         | fn_call
```

```
        | comparisons
        | arithmetic

arithmetic : MULTIPLY REG REG
           | DIVIDE REG REG
           | PLUS REG regconst
           | PLUS CONSTANT REG
           | MINUS REG regconst
           | BIT_AND REG regconst
           | BIT_IOR REG regconst
           | BIT_XOR REG regconst
           | BIT_NOT REG
           | NEGATE REG
           | TRUTH_NOT REG
           | LSHIFT REG regconst
           | RSHIFT REG regconst

control: if
       | switch
       | for
       | do
       | while

if : IF cond if_hack

if_hack : IF_ELSE_HACK thenbody JUMP_OVER_ELSE delay_slot EOSEQ elsebody
        | IF_ELSE HACK thenbody

thenbody : termsequence
elsebody : termsequence

switch : cond delay_slot cases

cases : SEQ CASE case_labels casebody cases
      | SEQ DEFAULT casebody EOSEQ

case_labels : SEQ CONSTANT case_labels
            | EOSEQ

casebody : sequence structured_goto EOSEQ

do : DO termsequence cond delay_slot

while : WHILE cond delay_slot whilebody

whilebody : sequence JUMP_TO_WHILE delay_slot EOSEQ

for : FOR initNstop delay_slot incrementNjump
```

```
initNstop : FOR_STUFF initialize stopNbody

stopNbody : FOR_C_N_B stop forbody

incrementNjump : FOR_INC_N_JUMP increment jumpback

initialize : termsequence

stop : cond

increment : termsequence

jumpback : LOOP_TO_FOR delay_slot

forbody : termsequence

delay_slot : sequence EOSEQ

cond : seqloads comparisons seqstores
     | seqloads modify seqstores

seqloads : SEQ load seqloads
         | /* empty */

seqstores : SEQ store seqloads
          | /* empty */

load : LOAD  REG load_src

load_src : REG
         | ADDRESS
         | CONSTANT
         | MEM

store : STORE store_dest REG

store_dest : REG
           | MEM

fn_call  : FN_CALL args LABEL
         | FN_CALL args REG

args : ARG REG args
     | ARG CONSTANT args
     | NOARG

reg_moves : MVI2F REG REG
```

```
          | MVI2F REG CONSTANT=0
          | MVF2I REG REG

conversions : FD2S REG REG
            | FD2I REG REG
            | FI2S REG REG
            | FI2D REG REG
            | FS2I REG REG
            | FS2D REG REG
            | FIXUD2S REG REG

comparisons : GE REG regconst
            | GT REG regconst
            | EQ REG regconst
            | LT REG regconst
            | LE REG regconst
            | NE REG regconst

structured_goto : BREAK
                | CONTINUE
                | RETURN sequence EOSEQ

regconst : REG
         | CONSTANT
```

Appendix C

Detailed Results

($\times 10^6$)	DLX	Opt. DLX	%	Impr.	%	Impr. Sched	%	Ext.	% Sched	Ext.	%
Cycles	571.49	430.49	75.33	419.00	73.32	405.00	70.87	493.00	86.27	455.00	79.62
Loads	131.00	81.50	62.21	68.50	52.29	72.50	55.34	86.50	66.03	90.00	68.70
Stores	92.00	77.00	83.70	68.00	73.91	72.00	78.26	75.00	81.52	78.50	85.33

Table C.1: Dhrystone results

	DLX	Opt. DLX	%	Impr.	%	Impr. Sched.	%	Ext.	%	Ext. Sched.	%
Cycles	760	700	92.11	740	97.37	647	85.13	740	97.37	647	85.13
Loads	164	168	102.44	147	89.63	147	89.63	147	89.63	147	89.63
Stores	160	189	118.13	192	120.00	192	120.00	192	120.00	192	120.00

Table C.2: Hanoi results

($\times 10^5$)	DLX	Opt. DLX	%	Impr.	%	Impr. Sched.	%	Ext.	%	Ext. Sched.	%
Cycles	44.54	23.94	53.75	31.19	70.03	28.95	65.00	32.38	72.69	29.45	66.11
Loads	9.77	2.40	24.58	2.43	24.91	2.44	25.07	3.05	31.30	3.07	31.46
Stores	1.88	1.17	62.35	1.17	62.35	1.18	63.20	1.17	62.35	1.19	63.20

Table C.3: Intmm results

($\times 10^6$)	DLX	Opt. DLX	%	Impr.	%	Impr. Sched	%	Ext.	% Sched	Ext.	%
Cycles	88.57	50.60	57.13	64.57	72.90	62.75	70.85	64.57	72.90	62.75	70.85
Loads	23.42	3.60	15.37	3.18	13.57	3.18	13.57	3.18	13.57	3.18	13.57
Stores	5.76	2.81	48.79	2.53	43.91	2.53	43.91	2.53	43.91	2.53	43.91

Table C.4: Knight results

	DLX	Opt. DLX	%	Impr.	%	Impr. Sched.	%	Ext.	%	Ext. Sched.	%
Cycles	181378	75342	41.54	126121	69.53	125810	69.36	126061	69.50	125751	69.33
Loads	41262	1450	3.51	3033	7.35	3034	7.35	3018	7.31	3019	7.32
Stores	19726	9904	50.21	9907	50.22	9908	50.23	9907	50.22	9908	50.23

Table C.5: Mersenne results

	DLX	Opt. DLX	%	Impr.	%	Impr. Sched.	%	Ext.	%	Ext. Sched.	%
Cycles	307940	106696	34.65	273543	88.83	260694	84.66	224798	73.00	218591	70.98
Loads	83167	19517	23.47	38796	46.65	38991	46.88	26534	31.90	26726	32.14
Stores	16295	7729	47.43	7520	46.15	7715	47.35	7523	46.17	7715	47.35

Table C.6: Sorts results

	DLX	Opt. DLX	%	Impr.	%	Impr. Sched.	%	Ext.	%	Ext. Sched.	%
Cycles	65680	30677	46.71	66448	101.17	64209	97.76	48357	73.63	53633	81.66
Loads	10216	4692	45.93	5088	49.80	11897	116.45	4899	47.95	9174	89.80
Stores	2189	2281	104.20	1968	89.90	1991	90.95	2020	92.28	2951	134.81

Table C.7: Tomcatv results

($\times 10^5$)	DLX	Opt. DLX	%	Impr.	%	Impr. Sched	%	Ext.	% Sched	Ext.	%
Cycles	15.16	5.72	37.77	9.45	62.37	8.61	56.78	9.61	63.40	9.18	60.58
Loads	4.24	0.67	15.72	1.17	26.34	1.13	26.61	1.08	25.55	1.23	29.10
Stores	2.09	0.34	16.10	0.34	16.13	0.35	16.67	0.52	24.86	0.60	28.74

Table C.8: Whetstone results

Bibliography

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AH91] Andrew W. Appel and David R. Hanson. Intermediate representation trees. Unpublished report, October 1991.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–250, March 1992.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, corrected edition, 1988.

[Bak77] B. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, 1977.

[BD88] Manuel E. Benitez and Jack Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), July 1988.

[Ber85] Robert L. Bernstein. Producing good code for the case statement. *Software — Practice and Experience*, 15(10):1021–1024, October 1985.

[Ber86] Robert Bernstein. Multiplication by integer constants. *Software — Practice and Experience*, 16(7):641–652, July 1986.

[Ber88] D. Bernstein. An improved approximation algorithm for scheduling pipelined machines. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume I, pages 430–433, St. Charles, Illinois, August 15–19, 1988.

[Bet94] Patrick Betremieux. An interpreter for LAST. McCAT Technical Memo 16, McGill University, 1994.

[BG89] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[BGM79] G. B. Bonkowski, W. M. Gentleman, and M. A. Malcolm. Porting the Zed compiler. In *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction*, pages 92–97. ACM SIGPLAN, 1979.

[BH93] Preston Briggs and Tim Harvey. Multiplication by integer constants. Available via ftp from cs.rice.edu:public/preston, October 1993.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Texas, April 1992.

[CAC+81] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[CCDM93] Philippe Canalda, Lucile Cognard, Annie Despland, and Monique Mazaud. The Pagaode system user's guide and reference manual. Technical Report 1.1, INRIA, Rocquencourt, France, April 1993.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. *SIGPLAN Notices,* 25(6), June 1990.

[CHKW86] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *Digest of Papers, 31st IEEE Computer Society International Conference, COMPCON Spring '86*, pages 132–137, San Francisco, California, March 3–6, 1986. IEEE Computer Society Press.

[Cho83] Frederick C. Chow. *A Portable Machine-Independent Global Optimizer — Design and Measurements.* PhD thesis, Stanford University, Stanford, California, December 1983. Also published as Technical Note 83-254.

[Cho88] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), July 1988.

[CKDK91] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An analysis of MIPS and SPARC instruction set utilization on the SPEC

benchmarks. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[CLR92] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, McGraw-Hill Book Co., Cambridge, MA, 1992.

[Con93] The COMPARE Consortium. Automatic generation of schedulers in the framework of the Pagode system. Technical Report D3.3.2/1, INRIA, Rocquencourt, France, May 1993.

[DF84] Jack Davidson and C. Fraser. Automatic generation of peephole optimization. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 111–116, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. *SIGPLAN Notices,* 19(6), June 1984.

[DF86] Jack Davidson and C. Fraser. A retargetable instruction reorganizer. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 234–241, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. *SIGPLAN Notices,* 21(7), July 1986.

[Dig92] Digital Equipment Corporation. *Alpha Architecture Manual.* Burlington, Vermont, 1992.

[Don92] Christopher M. Donawa. McBURG: a kinder, gentler interface for producing BURG specifications. ACAPS Technical Note 39, School of Computer Science, McGill University, Montréal, Québec, November 1992.

[DS90] Robert B. K. Dewar and Matthew Smosna. *Microprocessors: A Programmer's View.* McGraw-Hill Publishing Co., New York, 1990.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Symposium on Programming Language Design and Implementation*, pages 242–257, June 1994.

[EH94] Ana Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of IEEE 1994 International Conference on Computer Languages*, May 1994.

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montréal, Québec, September 1993.

[ESL89] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG: a generator for efficient back ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, Portland, Oregon, June 21–23, 1989. ACM SIGPLAN. *SIGPLAN Notices*, 24(7), July 1989.

[FH91] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. Technical Report CS-TR-303-91, Department of Computer Science, Princeton University, Princeton, NJ, February 1991.

[FHP92a] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Transactions on Programming Languages and Systems*, 1(3):213–226, September 1992.

[FHP92b] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG -- fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 7(30):478–490, July 1981.

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting A Compiler*. Benjamin/Cummings Publishing Co., Menlo Park, California, 1988.

[GF84] Mahadevan Ganapathi and Charles N. Fischer. Attributed linear intermediate representations for retargetable code generators. *Software — Practice and Experience*, 14(4):347–364, April 1984.

[GH86] James R. Goodman and Wei-Chung Hsu. On the use of registers vs. cache to minimize memory traffic. In *Proceedings of the 13th Annual International*

Symposium on Computer Architecture, pages 375–383, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 14(2), June 1986.

[Ghi92] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62, School of Computer Science, McGill University, Montréal, Québec, March 1992.

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. *SIGPLAN Notices,* 21(7), July 1986.

[Gui94] Ronald F. Guilmette, March 1994. Personal communication.

[HDE+92] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Conference Record of the 5th Workshop on Languages and Compilers for Parallel Computing*, pages 261–275, New Haven, Connecticut, August 3–5, 1992. Department of Computer Science, Yale University. Also available as ACAPS Technical Memo 46, School of Computer Science, McGill University, Montréal, Québec.

[HGAM92] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Proceedings of the International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 176–191. Springer-Verlag, October 1992.

[HM82] John L. Hennessy and Noah Mendelsohn. Compilation of the pascal case statement. *Software — Practice and Experience*, 12(9):879–882, September 1982.

[HM90] Larry B. Hostetler and Brian Mirtich. *DLXsim—a Simulator for DLX.* University of California at Berkeley, December 1990.

[Hor91] Nigel Horspool. CS471. Course notes, February 1991.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[JH94] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In *Proceedings of the 1994 International Conference on Compiler Construction*, volume 749 of *Lecture Notes in Computer Science.* Springer-Verlag, April 1994.

[JML91] Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors, *Code Generation — Concepts, Tools, Techniques, Proceedings of the Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, May 1991. Springer-Verlag.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[Jus94] Justiani. An array dependence framework for the McCAT C compiler. Master's thesis, McGill University, Montréal, Québec, 1994. Expected December 1994.

[JW89] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, Boston, Massachusetts, April 3–6, 1989. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 17(2), April 1989; *Operating Systems Review*, 23, April 1989; *SIGPLAN Notices,* 24, May 1989.

[KE93] Daniel R. Kerns and Susan J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, Albuquerque, New Mexico, June 1993. ACM SIGPLAN. *SIGPLAN Notices,* 28(6), June 1993.

[Kes84] Robert R. Kessler. Peep — an architectural description driven peephole optimizer. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 106–110, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. *SIGPLAN Notices,* 19(6), June 1984.

[KKM80] P. Kornerup, B. B. Kristensen, and O. L. Madsen. Interpretation and code generation based on intermediate languages. *Software — Practice and Experience*, 10(8):635–658, 1980.

[Kri90] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, 1990.

[LJ92] Luis Lozano and Alberto Jimenez. A register allocator for McCAT. McCAT Design Note 12, School of Computer Science, McGill University, Montréal, Québec, December 1992.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. pages 62–73, October 12–15, 1992. *Computer Architecture News,* 20, October 1992; *Operating Systems Review,* 26, October 1992; *SIGPLAN Notices,* 27(9), September 1992.

[Muk91] Chandrika Mukerji. Instruction scheduling at the RTL level. ACAPS Technical Note 28, School of Computer Science, McGill University, Montréal, Québec, 1991.

[NAJ+81] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and Ch. Jacobi. Pascal P implementation notes. In D. W. Barron, editor, *Pascal — The Language and its Implementation*, pages 125–170. Wiley, Chichester, 1981.

[Nel79] Philip A. Nelson. A comparison of PASCAL intermediate languages. In *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction*, pages 208–213. ACM SIGPLAN, 1979.

[OHM+90] Kevin O'Brien, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, and Mathew Zaleski. Advanced compiler technology for the RISC System/6000 architecture. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 154–161. International Business Machines Corporation, first edition, 1990. Order No. SA23-2619.

[OMMN90] Brett Olsson, Robert Montoye, Peter Markstein, and MyHong NguyunPhu. RISC System/6000 floating-point unit. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 34–42. International Business Machines Corporation, first edition, 1990. Order No. SA23-2619.

[Pat85] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[PS79] Daniel R. Perkins and Richard L. Sites. Machine-independent PASCAL code optimization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 201–207, August 1979.

[Sal81] Arthur Sale. The implementation of case statements in Pascal. *Software — Practice and Experience*, 11(9):929–942, September 1981.

[SC91] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *Computer*, 24(9):30–38, September 1991.

[SLH90] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 18(2), June 1990.

[SP89] Jong-Jiann Shieh and Christos A. Papachristou. On reordering instruction streams for pipelined computers. In *Proceedings of the 22th Annual International Workshop on Microprogramming and Microarchitecture*, pages 199–206, Dublin, Ireland, August 14–16, 1989. ACM SIGMICRO and IEEE-CS TC-MICRO. *SIGMICRO Newsletter*, 20(3), September 1989.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[Sta92] Richard M. Stallman. *Using and Porting GNU CC.* Cambridge, Massachusetts, June 1992. Available via anonymous ftp from `prep.ai.mit.edu`.

[Tie89] M. D. Tiemann. The GNU instruction scheduler—cs343 course report. Technical report, Stanford University, 1989.

[TvSS82] A. S. Tannenbaum, H. van Staveren, and J. W. Stevenson. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, 1982.

[TWL+91] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 137–151, Santa Clara, California, August 7–9, 1991. The Intel Corporation, Springer-Verlag. Published in 1992.

[Wal88] David W. Wall. Register windows vs. register allocation. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 67–78, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices*, 23(7), July 1988.

[Wir71] N. Wirth. The design of a PASCAL compiler. *Software — Practice and Experience*, 1(4):309–334, October 1971.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, June 26–28, 1991. ACM SIGPLAN. *SIGPLAN Notices,* 26(6), June 1991.

[WO75] M. H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Comput. J.*, 21(2):161–167, 1975.

[Wol82] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982. Rpt. No. UIUCDCS-R-82-1105.