Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations

Laurie J. Hendren*, Chris Donawa, Maryam Emami,
Guang R. Gao, Justiani, and Bhama Sridharan

McGill University, Montréal, Québec, Canada

Abstract
The effective exploitation of advanced technology for the development of the next-generation high-performance computers requires the integrated development of compiler techniques and architectural design. In order to provide a research tool with which we can experiment with both new architectural features and compiler support for those features, we have been developing the McGill Compiler/Architecture Testbed, McCAT.

In this paper we focus on the design of the McCAT compiler. The central theme of the paper is that the design of the family of intermediate representations should be driven by the analyses and transformations that are most important for effective compilation for architectures supporting some level of fine-grain parallelism. A primary objective of our design was to provide a natural way of supporting a framework for alias analysis that is general (handles scalars, arrays and pointers), accurate (provides accurate enough estimates for parallelizing transformations), and pervasive (alias information is available at the lowest level intermediate representation). In addition, we discuss how the design of the intermediate representations was motivated by the desire to support the use and development of analyzer-generator and code-generator generator tools that make the compiler flexible and useful for experimental development.

1 Introduction and Motivation

The efficient exploitation of fine-grain parallelism is a major challenge in the design of next-generation high-performance computers. To achieve both high-performance and cost-effectiveness, advanced VLSI architectures must take advantage of the ample resources provided by the hardware to aggressively exploit fine-grain parallelism: both within a uniprocessor and between multiple processors. The effective exploitation of this advanced technology presents new challenges to compiler developers and architecture designers alike. In fact, those who design high-performance architectures today must design machines in such a way that compiler technology can exploit the power of their architectures—often through parallelism at many different levels. Moreover, new compiler concepts and techniques are needed to cope with rapidly advancing hardware technology. In order to facilitate this interaction between the development of architectures and compilers, we have been designing and implementing McCAT, a compiler/architecture testbed which provides a unified approach to the development and performance analysis of compilation techniques and high-performance architectural features. In this paper we concentrate on presenting the overall design of the McCAT compiler, and in particular we discuss our use of a family of structured intermediate representations as the basis for the compiler.

*The work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

In the past, many techniques of automatic exploitation of scientific code parallelism can be classified into two categories: (1) mainly high-level (source to source) restructuring compilers and (2) low-level code optimizations and scheduling. These two problem classes are often treated apart, since they do not share a priori the same objectives. For example, compilers with high-level restructuring that are primarily targeted to automatic loop vectorization or coarse-grain parallelization do not usually also handle techniques for fine-grain loop scheduling such as software pipelining, or other instruction scheduling techniques. Even though the high-level transformation techniques perform parallelism detection and construct dependence graphs, the dependence analysis in not usually "pervasive", i.e. the information collected at the high-level does not pervade down to the level where lower-level optimizations for fine-grain parallelism may need it most. This is particularly a problem for alias analysis.

In order to develop a compiler that can perform both high-level and lower-level analyses and transformations, we have developed the McCAT compiler with the following major design goals.

A family of intermediate representations: The most important goal in our design was the development of a family of related intermediate representations. These intermediate representations range from a high-level abstract representation that accurately captures the original program, to a low-level representation that is suitable for register allocation, instruction scheduling, and code generation. The design of each intermediate representation was driven by the requirements of the analyses and transformations that we considered most important. In addition, we considered how each intermediate representation related to the next so that we would use the results of analysis at higher-level representations at the lower-level representations.

Accurate, general, and pervasive alias analysis: Alias analysis and dependence analysis is fundamental to the overall effectiveness of optimizing and parallelizing transformations. In response to this fact, we designed one intermediate representation specifically to allow us to perform accurate, general, and pervasive alias analysis. Our alias analysis methods must be *accurate* so that we can get the best results from instruction-scheduling and parallelizing transformations. The analysis must be *general* because we wish to handle a wide range of programs that use scalars, arrays, stack-allocated pointer data structures, and heap-allocated pointer data structures. Finally, our analysis must be *pervasive* in the sense that the important information collected by the alias analysis must pervade down to the lowest level of intermediate representation where it can be used for low-level transformations like instruction scheduling.

Application and development of advanced compiler tools: In designing the compiler structure we were also concerned with making our intermediate representations structured in such a way as to encourage the use and development of advanced compiler tools like analyzer generators and code-generator generators. In addition, we wanted to design such tools so that they could be easily parallelized.

We feel that it is important to also state explicitly what we were not concerned with. Our first decision was to concentrate on structured and structurable programs. That is, we selected a reasonable set of compositional control statements that we wish to consider in our analyses and transformations. Any program using gotos in an unrestricted manner must be converted to an equivalent structured program. In order to allow our compiler to handle a wide variety of benchmarks it will be necessary to provide a structurer and appropriate extensions to our intermediate representations so that we can accurately capture the more frequently used "structured" uses of gotos (for example, exiting from a nested loop). The second aspect of our development is that although we are concerned about the overall space and time complexity of the compiler, we were willing to sacrifice constant factors in both space and time if that let us develop a cleaner, more modular system that can be used and shared by a large group of students and researchers. Thus, one should not think of this as a production compiler, but as part of a testbed which encourages the rapid development of new and potentially complex analyses and transformations.

The remainder of this paper is structured as follows. In the next section, section 2, we present the overview of our McCAT compiler/architecture testbed. In sections 3 and 4 we outline the two most important abstract intermediate representations used in our compiler: SIMPLE and LAST. For each intermediate representation we discuss the motivation, design criteria, and we provide an overview of the actual representation. In section 5 we outline our approach to accurate alias analysis, and we illustrate how our design of the intermediate representations provides a natural platform for such analyses. In section 6 we briefly discuss the advanced compiler tools that are used. Finally, in section 7 we conclude and outline our future plans for McCAT.[1]

2 McCAT Overview

As indicated in Figure 1, the McCAT compiler/architecture testbed is composed of two major pieces, a retargetable parallelizing compiler, and an architecture testbed that consists of a variety of performance analysis tools.

The compiler itself consists of four major translation steps that covert an input C program into a series of structured intermediate forms. The first translation, labeled *front-end processing* in Figure 1, converts a C program into a high-level abstract syntax tree representation called FIRST. The main purpose of the FIRST intermediate representation is to cleanly separate the front-end processing of parsing and type-checking from the back-end processing of analysis, transformation, and code-generation. Since this translation step originated from the GNU C compiler, there was a natural abstract syntax tree form that already existed at the expression level. We have extended this form so that it captures completely and accurately the structure of an entire module or program. An important characteristic of FIRST is that all information about declarations, types, and type casting is completely and accurately encoded.

The second translation step is the *simplify* process which converts the high-level FIRST representation into a simpler and more structured SIMPLE representation that has been designed to support alias and dependency analysis, and high-level loop and parallelization transformations. It is important to note that from both FIRST and SIMPLE it is possible to recreate a C program and thus use the McCAT as a source-to-source translator.

The third translation step is the *blastify* process which converts the SIMPLE representation into a lower-level LAST representation which is suitable for register allocation, instruction scheduling and low-level loop transformations. The LAST representation can be directly interpreted, or it can be used as the input the fourth and final translation step which is generating assembler code for a variety of high-performance architectures.

In Figure 2 we give an example of the representation of a simple `for`-loop program as it is represented at the different levels of intermediate representation. In the next two sections we discuss the design and implementation of the SIMPLE and LAST intermediate representations in more detail.

3 SIMPLE - An Intermediate Representation for Alias Analysis

High-level procedural languages, such as C and Pascal, allow the creation of complex user-defined types and structures. Programs written in such languages vary from the very regular computations found in many scientific applications to more irregular computations that have complicated pointer data structures and irregular control flow including recursion and mutual recursion. In order to fully exploit instruction-level parallelism in both scientific and non-scientific programs, accurate interprocedural analysis is needed for both pointer data structures and arrays. Fundamental to this problem is the need for an appropriate level of intermediate representation on which both alias analysis and dependency analysis can be performed accurately in a relatively straightforward manner.

[1] A longer version of this paper may be found in [10].

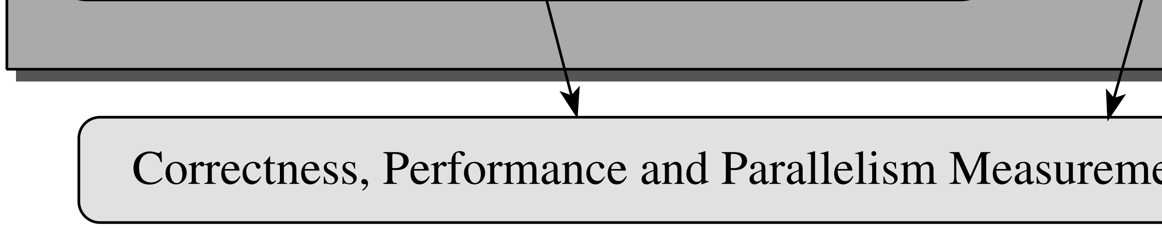
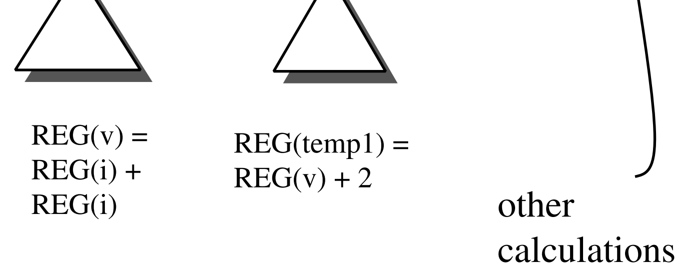


Figure 1: Overview of McCAT




```
for(i=0;i<10;i++)
{
   v = i+i;
   a[v+2] = a[2*i];
}
```

```
for(i=0;i<10;i=i+1)
{
   v = i+i;
   temp1 = v+2;
   temp2 = i*2;
   a[temp1] = a[temp2];
}
```

```
For statement:
Branch delay slot: nop
Start statement:
REG(i) :=  0
Stop condition:
REG(aValue0) := REG(i) <=  9
For Body:
REG(v) := REG(i) + REG(i)
REG(temp1) := REG(v) +  2
REG(aValue1) := 2
REG(temp2) := REG(i) * REG(aValue1)
REG(aAddr0) <-LOAD(Addr)- ADDR(a)
REG(aValue2) := 4
REG(aAddr1) := REG(temp1)*REG(aValue2)
REG(aAddr1) := REG(aAddr1)+REG(aAddr0)
REG(aAddr2) <-LOAD(Addr)- ADDR(a)
REG(aValue3) := 4
REG(aAddr3) := REG(temp2)*REG(aValue3)
REG(aAddr3) := REG(aAddr3)+REG(aAddr2)
REG(aValue4) <-LOAD(Int)- REG(aAddr3)
REG(aAddr1) <-STORE(Int)-< REG(aValue4)
;Iteration statement(s):
REG(i) := REG(i) +  1
```

Figure 2: A Simple Example of the FIRST, SIMPLE, and LAST Intermediate Forms

3.1 Design Criteria

In response to the challenge of accurate alias analysis and dependence analysis, we designed the SIMPLE intermediate representation. In this design process we considered the following design criteria:

Compositional Representation: The intermediate representation should be a compositional representation of the program, where the control flow is regular and explicit. For example, we would like to be able to analyze a while loop by analyzing only its components: the conditional expression and the body. This kind of compositional representation has three advantages: (i) the flow of control is structured and is explicit in the program representation, (ii) structured analyses techniques and tools supporting such techniques can be used to analyze all the control-flow constructs, and (iii) it is simple to find and transform groups of loop nests. It should be noted that in addition to ordinary compositional constructs such as conditionals and loops, our compositional approach should directly support the commonly used `break` and `continue` statements for loops, and the `return` statement for procedures and functions. However, unrestricted use of `goto` is not compositional and cannot be supported directly. Any program with unstructured control flow must be converted into an equivalent program with structured control flow [2, 3, 23].

Explicit Array and Structure References: The identity of array and structure references should be retained i.e., the array and structure references should not be broken down into a series of lower level statements that perform address calculations. This is required so that we can make full use of high-level information such as array dimension, array size, pointer types, recursive structure types and so on.

Types and Typecasting: The exact type information and type casting should be retained. Often the alias analysis methods can take advantage of type information to provide more accurate results. A simple example is that one can take advantage of type information, and the absence of type casts to infer that a variable of one type cannot be aliased to a variable of another type. A more advanced example is the use of recursive types for dynamically-allocated pointer structures [5, 13].

Pervasive Dataflow Information: It should be possible to transmit important dataflow information collected at higher-level intermediate representations to the lower-level representations, and thus improve the effectiveness of the low-level transformations. For example, alias analysis information collected at a high-level can be used to perform better dependence analysis and therefore better instruction scheduling at a lower-level.

Simple to Analyze: The intermediate representation should be simple enough so that it could be analyzed in a straightforward manner. A proficient programmer will use complex structures and all the tricks allowed by the language (especially C!) to develop his or her software. To analyze such a program as accurately as possible we need to break down complex structures and statements into simpler ones.

Clear Semantics: The intermediate representation should have clear and obvious semantics. One part of this process is clarifying some of the implicit meanings in C programs. For example, consider the statement `b = a`. If `a` is a scalar, then it denotes a value, while if `a` is an array it denotes the address of `a`. These implicit semantic rules in C must be made explicit in the intermediate representation.

Interprocedural Analysis: The intermediate representation should retain all the information about the complete program or module, so that interprocedural data-flow and alias analysis can be performed. This is particularly important when we have non-scientific code that is composed of many small and possibly recursive procedures.

Standard Representation: Similar to the spirit of DIANA[20], we are aiming at a standard intermediate representation for programs, so that a large number of researchers and students can have a common ground to independently develop analysis and optimization techniques. This also allows a number of users to share a class of software tools.

Automating the Analysis and Optimization Phases: The intermediate representation should be able to support the automatic generation of the analysis and optimization phases of the compiler.

SIMPLE to C: After performing certain high-level transformations and program restructuring, there should be a straight-forward translation from the intermediate representation back to C. This will help in two ways: (i) we can verify the syntactic and semantic correctness of our transformations by testing the output C program with a standard C compiler, and (ii) we can check the effectiveness of high-level transformations without requiring a complete back-end for every architecture.

3.2 Overview of SIMPLE

In this section, we give an overview of the salient features of SIMPLE. As illustrated in Figure 1, the *simplify* translation takes a high-level FIRST representation of a programs and produces an output representation at the SIMPLE level. As its name suggests, the SIMPLE intermediate form is a simplified form of FIRST; control flow is structured, complex statements are broken down into a series of simpler statements, complicated variable names are split whenever possible and all loops, switches and conditionals are modified to adhere to the restricted SIMPLE format.

The following subsections describe the special features found in SIMPLE, while a complete description of SIMPLE is given in [21]. While transforming FIRST to SIMPLE, simplifications are performed in these three major areas: (i) in the representation of variables, (ii) in the format of basic statements — only a restricted number of operands are allowed, and (iii) in the representation of control flow constructs — these are simplified.

3.2.1 Variables

In high-level programming languages such as C which allow user defined type structures, arbitrarily complex types can be defined by the programmer. We could have nested array and structure references, with pointers in them to further complicate matters. To perform accurate analyses of the variables of these types, we need to break them up in a systematic manner so that appropriate alias analysis techniques can be applied. We define a `varname` to be a simple variable name (e.g., '`myname`'), a pure structure reference(e.g., '`a.b.c`'), a pure array reference (e.g., '`a[5][7]`'), or a pointer to a structure reference (e.g., '`(*a).b.c`'). Any other reference is broken down into these simple references. By standardizing the variable references in this way we can reduce the number and complexity of advanced alias analysis rules that we must define. Thus, structure and array names are kept at the right level for high level analyses and transformations.

3.2.2 Basic Simple Statements

In the design of SIMPLE, we have identified fifteen basic statements. Any other complicated statement in C can be broken down into a sequence of these statements. In Figure 3, we list the set of basic simple statements. Note that variables '`x`' and '`y`' denote `varnames`, whereas the variables '`a`', '`b`', '`p`' and '`q`' denote `vals` (`vals` are either constants or identifiers).

Certain special kinds of expressions allowed by C, namely compound expressions and conditional expressions, are also transformed into equivalent SIMPLE statements. For example, conditional expressions are transformed into if-then-else statements, and compound expressions transformed to a sequence of simple statements of the above form.

3.2.3 Compositional Control Statements

The compositional control statement forms supported directly by SIMPLE are restricted (simplified) versions of : statement sequences, `for`-loops, `while`-loops, `do`-loops, `switch/case` state-

`x = a` *binop* `b` `*p = a` *binop* `b`	`x = a` `*p = a`	`x =` *cast* `b` `*p =` *cast* `b`	`f(`*args*`)`
`x =` *unop* `a` `*p =` *unop* `a`	`x = *q` `*p = *q`	`x = &y` `*p = &y`	`x = f(`*args*`)` `*p = f(`*args*`)`

Figure 3: Basic Statements

ments, and `if/else` statements. In addition, `return` is supported for exiting a procedure or function, and `break` and `continue` are supported for exiting a loop.

In each of the simple control constructs the complexity of the conditional expressions is reduced. For example, the conditional expression in a while loop must be simplified so that all side-effects are moved out of the conditional expression. The conditional parts in `do`-loops, and `if/else`-statements are handled in a similar fashion. `For`-loops are given special treatment because compound expressions are possible in the `for`-loop header.

In simplifying `switch/case`-statements, special care must be taken. In C, the `case` labels are implicit `gotos` and therefore a `switch/case` statement may have a very unstructured control flow. In fact, the `switch/case` statement may not even have structured block nesting because a block may begin in one branch of the `switch` and end in another branch of the `switch`. The body of a simplified `switch/case` statement consists of a sequence of `case` statements, finally ending with a `default` statement. Each `case` statement is made up of one or more case expressions, followed by a statement list, and ending with a stop statement, which could be either a `break` or a `continue` or a `return` statement. Various transformations are performed on FIRST to bring it to the above format. If a `default` statement is missing, one is created and appended to the end of the switch statement. If a case does not end with a stop statement (a `return`, `break` or a `continue`), then code is replicated so that the above format is achieved. If there are unstructured block nestings, these are taken care of by a process an unnesting transformation where block nestings are removed by yanking variables to outer blocks. Here, renaming of variables must be performed when necessary.

4 LAST - A Structured Low-Level Intermediate Representation

The final and lowest level of abstract program representation, LAST (Lower-level Abstract Syntax Tree), maintains some of the characteristics of FIRST and SIMPLE, but does have some basic structural differences. There are two phases requiring LAST: analyzing/transforming the program at the machine level, and actually generating assembly language code. The design of LAST was heavily influenced by our experiences with the first two intermediate representations, by code generation tools like TWIG [1] and BURG [8], and by the primary target architectures: load/store machines that support some level of fine-grain parallelism.

Just as accurate alias analysis was the motivating factor in the design of SIMPLE, register allocation, instruction scheduling, and code generation were the motivating factors for the design of LAST. We consider register allocation one of the primary factors because we have oriented McCAT towards load/store architectures that are becoming increasingly popular for high performance computers [15]. Hand in hand with register allocation is instruction scheduling as a method of improving performance. Since all of our target architectures have some sort of fine-grain parallelism, instruction scheduling is one of the most crucial transformations required for effective utilization of the hardware resources. Finally, as mentioned in the introduction, McCAT is a research tool, and as such, flexibility is an important characteristic. We want to experiment with a wide range of architectures, and so the code generators should be highly re-targetable within certain classes of architectures.

4.1 Design Criteria for LAST

Based on our desire for an intermediate representation that not only supported register allocation, instruction scheduling and favorable code-transformations, but was also very flexible, we defined several design criteria, which are described below.

Support for structured analysis: The intermediate representation should be a structured compositional representation whose constructs directly correspond to the constructs in SIMPLE. This requirement for a compositional representation carries with it the same advantages as outlined in the design of SIMPLE. In addition, a compositional form allows us to develop new structured techniques such as our register allocation algorithm that is based on hierarchical cyclic interval graphs [14]. Structured analysis techniques requires that a high level construct such as a `for`-loop be identified as such, and not just as a conditional branch[2] with associated basic blocks.

Support for pervasive analysis: The data flow analysis information, generated and stored in SIMPLE, should be accessible by LAST. Unfortunately, SIMPLE represents programs at too abstract a level to perform low-level code-improving transformations such as instruction scheduling. At the same time, information such as alias analysis that is best generated at the SIMPLE level is crucial for some of these lower-level transformations, so LAST must be able to access SIMPLE along with its associated analysis information. Since SIMPLE stores this information at the statement level, LAST should have constructs corresponding to statements.

Support for load/store machines: LAST should be oriented towards load/store architectures, meaning that the representation should differentiate between memory locations and registers holding a value for a variable, so that there are explicit loads and stores to memory.

Support simple code generators: Since we wish to experiment with a wide variety of real and experimental architectures, McCAT should require very little 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 intermediate representation should be generic enough for load/store architectures that the compiler is easily ported between machines.

Support for high-level tools: McCAT should support high-level code-generator tools, such as a code-generator generator, in order to ease the task of porting McCAT between architectures.

Expose all opportunities for transformations: LAST should expose all opportunities for code and performance-improving transformations, such as filling branch delay slots, variable induction with 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.

4.2 Overview of LAST

Given the design criteria in the previous section, we have designed the LAST intermediate form. In this section we provide an overview of the important features of LAST.

As illustrated in Figure 1, LAST is produced after the SIMPLE representation has been created and all flow analysis information has been generated on the SIMPLE representation. The *blastify* (binary lastify) process is used to produce from the SIMPLE representation an equivalent, lower-level, LAST representation.[3] The most striking difference between SIMPLE and LAST is that LAST exposes the underlying architecture, differentiating between memory and registers and specifying branch statements with their associated labels (see Figure 2). However, LAST still maintains the compositional structures represented in SIMPLE such as `for`-loops and `switch` statements.

In essence, all high-level structures are maintained, but LAST provides a different, more detailed perspective; `for`-loops, for example, not only have initialize, condition and body compo-

[2] And of course an unconditional branch at the end of the loop's body.

[3] A binary version was chosen simply because our code-generator generator, BURG, supports only trees of maximum arity two.

nents, but also branch delay slots and labels, corresponding to the conditional and unconditional branches that will be produced at code generation time (see Figure 2).

There are several important differences between SIMPLE and LAST, required mostly to expose the architecture to code-transformation phases. These differences, listed below, are:

Explicit Address Calculations: Array and structure references, which are both represented by sequences of address calculations followed by the appropriate load or store. This is a significant transformation, in that we have now lost the high-level notion of variable, and instead have the lower-level notion of memory address. This means that we must perform all analyses that requires the notion of variable and type at the SIMPLE level, and leave the analyses and transformations that require detailed information about memory locations to be done at the LAST level.

Relation between SIMPLE and LAST trees: Individual LAST statements which, while structured similarly to SIMPLE statements, have a parent/child relationship with their corresponding SIMPLE statement; each LAST statement keeps a pointer back to its associated SIMPLE statement in order to access the flow analysis information that is generated and stored there. This allows us to associate a particular memory location at the LAST level with its corresponding variable reference (and associated alias information) at the SIMPLE level.

Exposed function prologue/epilogue: We have introduced at the LAST level the *save-registers* and *restore-registers* instructions, which expose a function's prologue and epilogue to the instruction scheduler. After LAST is created with these instructions, subsequent passes over it, one of which is the register allocator, will insert machine dependent information such as which registers must be saved and restored to/from the stack. The *save* and *restore* instructions will be replaced by individual statements that perform the actual load and store operations, which in turn may be scheduled with the body of the function.

Exposed memory hierarchy: In order to clearly distinguish between values stored in memory and values stored in registers, we introduced the *REG* and *MEM* specifiers, which indicate references to variables. They identify whether the reference is to a register, a memory location or a memory location whose address is stored in a register (REG handles the first and last of these functions). Each of these specifiers is associated with a variable, which distinguishes them from each other. For example, REG(a) would refer to a register containing the value of variable 'a', and MEM(b) the memory location holding the value of variable 'b'. See Figure 2.

Explicit *load* and *store* instructions: Related to the previous point, we also want to make explicit loads and stores between registers and a memory locations.

Exposed labels and delay-slots: Even though we wish to keep a compositional structure, we also want to expose *delay-slots* and destination labels, indicating delay slots after branches and possible branch targets. The explicit targets allow branch chaining to be detected and eliminated.

4.3 Related Intermediate Representations

There are two intermediate representations that are interesting comparisons with LAST: SUIF [22], and RTL [16]. SUIF is similar to LAST in its ability to represent high-level constructs such as `for` loops; SUIF can maintain program structure, while exposing low-level details such as array reference address calculations. However, SUIF achieves this in a manner opposite to LAST. Where LAST is essentially a superset of SIMPLE (i.e. similar structures with extra instructions such as *save-registers*), SUIF requires a superset (high-SUIF) to represent its high-level constructs. SUIF is derived from a bottom-up perspective, and LAST from a top-down. In addition, SUIF is one intermediate representation and is used for all optimization phases, whereas McCAT applies each analysis and transformation at the appropriate representation level. For example, alias analysis is best done at the SIMPLE level, while instruction scheduling is best done at the LAST level.

The other intermediate representation, RTL, does not represent high level structures. RTL represents a control-flow graph and not an abstract syntax tree, like LAST; RTL consists of jumps and target labels. RTL does have classes of nodes, and it's explicit representation of branch targets is similar to LAST.[4]

5 General, Accurate and Pervasive Alias Analysis

One of the fundamental motivating factors in the design of the McCAT compiler was the desire to support accurate, general, and pervasive alias and dependency analysis. As described in the previous section, we have supported the concept of pervasive analysis by designing the SIMPLE and LAST representations such that analysis information collected at the SIMPLE level is accessible at the LAST level. In this section we outline how we can support accurate and general analysis at the SIMPLE level. To provide accurate analysis we must support interprocedural analysis even in the presence of mutual recursion, and to provide a general analysis we must support a wide variety of alias analysis methods, where each method is designed to produce good results for a certain class of data structures. More precisely, we have separated the alias analysis problem into three problems: (1) analysis for stack-allocated, non-recursive data structures, (2) analysis for heap-allocated, recursive data structures, and (3) analysis for array references based on traditional dependence tests. In the following sections we briefly outline each of these analysis problems and we discuss how SIMPLE provides a natural platform for performing such analyses.

5.1 Analysis of Non-recursive Stack-allocated Structures

In this section we give an overview of our approach to interprocedural aliasing for stack-allocated data structures in C. A good summary of previous work in this area, and the presentation of a new analysis algorithm is given in [17, 18]. Our approach differs from other approaches in that we do not compute the alias pairs explicitly, but we have chosen an approximation that computes the relationships between abstract stack locations. The central idea is that since we are dealing with data-structures that are stack-allocated, we can model the real stack with an abstraction of the stack. This is a very direct and simple approach that gives accurate results with which we can perform a straight-forward dependence analysis based on the *reads* and *writes* to the abstract stack locations.

Let us begin by illustrating the *stack-alias* problem with a very simple example. Consider the program in Figure 4(a). Note that at the program point just before the statement `**pp = 2`, the location denoted by `**p` and the location denoted by `x` are aliased. This aliasing imposes an output dependence between the statement `x=3` and the statement `**pp = 2`. As in all of our analyses, we do not solve the problem directly on the input program, but we first simplify the problem by applying the *simplify* transformation to produce an equivalent SIMPLE program as shown in Figure 4(b). As discussed in section 3, this resulting SIMPLE representation is compositional and only the restricted simple statements given in Figure 3. For our small example program we note that the statement `**p = 2` has been converted into two statements `temp1 = *p; *temp1 = 2`.

Our stack-alias method is based on a structured analysis that approximates the relationships between abstract stack locations. For each program point we collect information about relationships between each pair of abstract stack locations `x` and `y`. In particular, we say that `x` is *definitely points to* `y` if the contents of stack location `x` *definitely* points to stack location `y`, and we say that `x` *possibly points to* `y` if the contents of stack location `x` *might* point to stack location `y`. Figure 4(c) gives a pictorial view of the relationships between the abstract locations at program point 4 for the simplified SIMPLE program given in 4(b). Figure 4(d) gives the concrete representation actually used in the analysis. In general, each entry in a stack matrix is one of: (1) empty indicating there is guaranteed to be no relationship between the abstract stack locations, (2) D indicating there is a *definite* relationship, or (3) P indicating that there is a *possible* relationship. As there is no approximation due to control flow in our example, we find that the stack matrix in Figure 4(d) contains only empty and D entries.

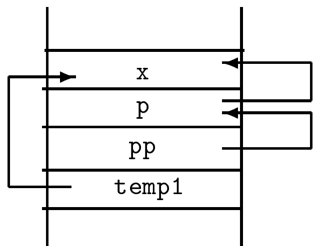
[4] This part of LAST was actually patterned after RTL.

```
{ int x, *p, **pp;

  x = 3;
  p = &x;
  pp = &p;
  **pp = 2;
}
```

(a) Example program

```
{ int x, *p, **pp;
  int *temp1;

  x = 3;           /* 1 */
  p = &x;          /* 2 */
  pp = &p;         /* 3 */
  temp1 = *pp;     /* 4 */
  *temp1 = 2;      /* 5 */
}
```

(b) SIMPLE representation



(c) Abstract Stack

	x	p	pp	temp1
x				
p	*D*			
pp		*D*		
temp1	*D*			

(d) Stack Matrix

Figure 4: A simple example of stack-aliasing

Although a complete description of our stack-alias method is outside of the scope of this paper, we summarize why SIMPLE provides a good basis for such an analysis. A more complete description of the analysis method and its implementation on SIMPLE is available in [7]. In terms of the stack-alias analysis, the two most important features of SIMPLE are: (1) the basic statements are very simple and contain all of the required type information, and (2) the SIMPLE intermediate form is compositional.

The simplicity of the basic statements is important for this type of analysis because we may define very straight-forward rules that only need to consider at most one level of dereference, and in addition each dereference must be associated with a name that can be used as the abstract stack location. This feature is illustrated in the program given in Figure 4(b) where we have introduced the name `temp1` to give us a name for the first dereferencing of `**pp`.

The compositionality of the representation means that we can define one analysis rule for each of the compound statements such as conditionals and loops. This makes it very easy to: (1) reason about the fixed-point computations for each kind of loop construct and (2) to experiment with a variety of optimizations of the analysis.

5.2 Analysis of Recursive Heap-allocated Structures

In addition to the stack-alias problem discussed in the previous section, we also wish to support an advanced alias analysis mechanism for dynamically-allocated data structures such as lists, trees, and related structures. In developing this alias analysis we have built upon our previous experience with path matrix analysis, a technique for analyzing programs with tree-like dynamically-allocated pointer data structures [13], and our recent experience with extending these techniques to more complex multi-dimensional pointer data structures [11].

Although these analysis techniques are more complex than those required for stack-allocated structures, they follow the same paradigm of collecting an interprocedural approximation based on structured flow analysis. Even though the approximation itself is very different, the SIMPLE representation is also the right level of representation for this class of alias analysis problems. Of particular importance for these analyses is that SIMPLE contains all of the type information for each defined type and each variable. This information is crucial for the safe and accurate functioning of this class of analysis. Furthermore, the compositional nature of the analysis based on SIMPLE allows one to develop correctness proofs based on structural induction [12].

5.3 Analyzing Array References

SIMPLE also provides a good base for implementing the traditional alias and dependence analysis needed for the sort of array references found in scientific programs. In order to demonstrate the benefits of performing array dependence analysis on the SIMPLE representation, we implemented a standard dependence analysis for array references in a single loop nest [4, 9, 19, 24, 25].

We can summarize the process as follows. We are currently focusing on *perfect nested loops*, i.e. a single loop nest in which the innermost loop body conforms to certain restrictions. After identifying the loop nests, the next step is to analyze the loop body to determine which pairs of array references *might* be aliased. For example, we need to consider all pairs of array references that might refer to the same array. Two arrays may be the same if they have the same array name, or if their array names are possibly aliased (both array names point to the same actual array). This aliasing information is available as a result of the stack matrix alias analysis. After the references of interest have been located, the actual index expressions are collected using use-def information, and then transformed to a set of expressions in canonical form so that standard dependence tests may be applied.

By applying all of these steps, the SIMPLE intermediate form is decorated with both a concise description of the canonical form for each array reference, as well as the results of the dependence tests. This information is then available for further analysis and transformation at the SIMPLE level as well as for dependence information at the LAST level (i.e. the information is pervasive).

6 Using and Developing Compiler Tools

One of our initial motivations in the design of SIMPLE and LAST was the application and development of advanced compiler tools. In our design we considered the development of: (i) a SIMPLE analyzer-generator to automate the production of data-flow analyzers, (ii) a SIMPLE optimizer-generator which produces different optimizers given a set of input specifications, (iii) a LAST interpreter to interpret the LAST tree, (iv) a LAST profiler which can measure parallelism under a variety of architectural models, and (v) a code-generator generator which simplifies the task of generating code from LAST.

In this section we discuss our SIMPLE analyzer-generator which takes full advantage of the compositional structure of SIMPLE, and our use of BURG, a code-generator generator that is based on minimal-cost tree-matching.

6.1 The McTAG Analyzer-Generator

Research in automating the analysis and optimization phases of a compiler is still in its early stages. Within the McCAT project we have developed the McTAG analyzer-generator which takes advantage of the compositional nature of the SIMPLE intermediate form to provide a straightforward way of specifying new analyses. This tool allows us to quickly implement the traditional flow analyses, as well as providing the backbone for the development of relatively advanced analysis techniques like the alias analysis presented in section 5.

The generator takes as input a file giving a set of specifications for the abstraction being computed (sets, stack matrices, path matrices, etc.) and a set of pattern-based rules, one rule for each expression and statement type supported by SIMPLE. The output is a data-flow analyzer program which traverses the SIMPLE tree applying the appropriate rules and collecting the specified data-flow information at each program point. A complete description can be found in [21].

6.2 Code-Generator Generator

As mentioned in our design criteria for LAST, we use a code-generator generator, called BURG [8], to produce assembly code for RISC-like machines. BURG assumes an intermediate representation in the form of a binary tree, and provides a minimal-cost cover evaluator and a general traversal mechanism. The BURG user provides a high-level interface specification for the intermediate representation, and a set of tree patterns with associated costs and actions. We have created an interface called McBURG, that takes this specification and produces a BURG specification [6]. Given this input, BURG generates a code-generator that matches these patterns and performs the actions and associated tree traversals. We use BURG mainly for its traversal routines and we use McBURG as an interface that allows us to express the target architecture in an abstract, concise, easy-to-comprehend form.

7 Conclusions and Further Work

The design of an experimental compiler/architecture testbed is a complex task with a wide variety of design choices. Our decision to attack such a problem is directly related with the demand arising from the quantitative approach for computer architecture design and development. In the past decade, we have witnessed the impressive performance gain due to the RISC architecture evolution. Much of this success has been due to an integrated approach of design of optimizing compilers and the architectures. Many design tradeoffs have been made through quantitative analysis and measurement, combining the effect of both compiler and architecture features. We anticipate that this trend towards a performance-guided design approach to architectures and compilers will continue, and there is little doubt that parallel processing is the key to the performance to computer systems ranging from single uniprocessor workstation applications to massive parallel scientific computation. Thus, the compiler design for testbeds are facing new challenges so that they can meet the requirements for parallelism analysis and exploitation, at all levels of user programs and for a variety of different architectures.

In order to experiment with both high-level and low-level optimization techniques and their effects on the underlying architecture, we needed to design suitable intermediate representations of the program. The three structured intermediate representations so designed, namely FIRST, SIMPLE and LAST, are extremely well suited to perform various analyses and optimizations at different program levels. Sophisticated alias analysis and dependence analysis techniques have been implemented on SIMPLE, and the important information gathered at that level is made available for optimizations on LAST. Further, these intermediate representations are structured in such a way that advanced compiler tools like the analyzer generator, and the code-generator generator could be used and developed.

Currently the design and implementation of FIRST, SIMPLE and LAST have been completed. At the SIMPLE level we have developed sophisticated alias analyses as well as the traditional analyses such as constant propagation. Many of these are implemented using the SIMPLE analyzer generator. The McCAT testbed now forms the backbone for compiler and architecture development in both our research and in graduate teaching.

Work is underway to perform case studies using McCAT. Currently, we are concentrating on (1) fine-grain loop scheduling: code generation for loop unrolling and software pipelining with realistic resource constraints; (2) an integrated instruction scheduling and register allocation code generation scheme; (3) various alias analysis methods to support the above implementation.

References

1. Alfred A. 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, 1989.
2. Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE transactions on software engineering*, 18(3), 1992.

3. B. Baker. An algorithm for structuring flowgraphs. *JACM*, 24(1), 1977.
4. U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Mass., 1988.
5. Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *1992 International Conference on Computer Languages*, pages 2–13, April 1992.
6. Christopher Donawa. McBURG: A kinder, gentler interface for producing BURG specifications. ACAPS Technical Note 39, McGill University, 1992.
7. Maryam Emami. An alias analysis for stack-allocated data structures. Master's thesis, McGill University, expected February 1993.
8. Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.
9. Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.
10. L. J. Hendren, C. Donawa, M. Emami, G. Gao, J. Justiani, and B Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. ACAPS Technical Memo 46, McGill University, August 1992.
11. L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proc. of the SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 249–260, San Francisco, Calif., Jun. 17–19, 1992. ACM SIGPLAN. Also in *SIGPLAN Notices*, 27(7), Jul. 1992.
12. Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell U., Jan. 1990.
13. Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), Jan. 1990.
14. L.J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Lecture Notes in Computer Science*, 641:176–191, 1992.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.
16. Ralph E. Johnson and Carl McConnell. The RTL System: A Framework for Code Optimization. Technical report, University of Illinois at Urbana-Champaign, 1991.
17. William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation*, pages 235–248, 1992.
18. William A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, 1992.
19. D. E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.
20. David S. Rosenblum. A methodology for the design of Ada transformation tools in a DIANA envi ronment. *IEEE Software*, 1985.
21. Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, September 1992.
22. Steven W.K. Tjiang, Micheal E. Wolf, Monica S. Lam, Karen L. Pieper, and John L. Hennessy. Integrating Scalar Optimization and Parallelization. Technical report, Stanford University, 1991.
23. M. H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Comput. J.*, 21(2), 1975.
24. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.
25. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.