

# Design of a High Level Intermediate Representation for Attribute-based Analysis

*Krisztián Flautner, David Greene, Matthew Postiff, David Helder, Charles Lefurgy, Peter Bird, and Trevor Mudge*  
University of Michigan Advanced Computer Architecture Lab  
{manowar, greened, postiffm, dhelder, lefurgy, tnm}@eecs.umich.edu, plbird@aol.com  
<http://www.eecs.umich.edu/mirv>

## Abstract

*This paper discusses the design and implementation of the Mirv high-level prefix intermediate representation, which is being used to create compilers for imperative languages. This representation is designed to support efficient attribute propagation based analysis and single pass code generation. The paper explains the design of Mirv and the set of reusable components that implement attribute propagation and code transformation. An example is used to illustrate how Mirv can greatly reduce the effort needed to implement analysis and optimization filters. Difficulties with the high level representation are also noted. A third party C front-end has been used to create a working prototype compiler targeted to the x86 instruction set, and the paper concludes with some initial statistics obtained from compilation experiments.*

## 1. Introduction

Compiler implementation has been recognized as a complex problem with a large design space. In this paper, we illustrate a unified compiler infrastructure based on attribute propagation based techniques. The Mirv language — which is used as the internal representation (IR) of the compiler — has been specifically designed to support these methodologies; it facilitates attribute-based analysis and optimizations as well as fast code generation.

Mirv provides a program representation that preserves much of the high level information of the source program while discarding its idiosyncrasies. The language is not intended for human consumption, it is designed to present information about a program in a computer friendly manner. While some effort has been made to keep Mirv independent of the source language, currently only a C front-end is implemented.

This paper illustrates how the choice of the intermediate language influences the compilation techniques. Since Mirv is a language that preserves the high level control-flow structures along with expression subtrees, traditional code analysis and transformation algorithms which use basic-block and quad data types must be adapted. This is accomplished through the use of algorithms that propagate attributes over the Mirv tree. The ease of this process is greatly dependent on the underlying program representation. For this reason the Mirv language and the attribute based techniques were designed to complement one another.

Most traditional compiler frameworks do not tie the IR and the compilation model as closely together as the Mirv compiler does. We did this to provide a unified environment for analysis and transformation filters. The trade-off is that it may be harder to extend Mirv with new operators, since the underlying assumptions of the language must always be considered. Another distinction from other compiler frameworks such as SUIF [11], IMPACT [2] and gcc is that Mirv performs most optimizations on a single, high-level IR. Only very machine-specific operations (such as code scheduling) are performed on a machine instruction level representation. While our approach uses only a single representation for the sake of simplicity, we have bridged the multiple levels of representation by the use of attributes associated with operator nodes.

Section 2 provides an overview of the Mirv language and Section 3 describes the compiler methodology. Section 4 gives an introduction to attribute propagation and demonstrates how it is used to perform dataflow analysis and a simple code transformation. We provide some measurements of the compiler in Section 5. Related work is discussed in Section 6 and Section 7 presents the conclusions.

## 2. The Mirv language

Mirv is a high level tree-based program representation. All operators are in prefix form, which simplifies code generation by providing the context for a set of nodes through the parent node operator. The high-level control flow information is preserved along with information about variables and datatypes. The operator tree representation can be linearized, allowing one to write simple code generators for the language using standard parsing techniques. In fact standard tools such as Lex/Yacc and PCCTS [7] can be used for this task. A syntax directed translation scheme [1] provides fast and efficient code generation using a linear-time context-free parser. We use the parser framework both for code generation and for collecting dataflow information.

While semantically Mirv most closely resembles the C language, it differs from it in certain respects:

- Assignments cannot be used as rvalues in an expression subtree. This simplifies the job of the analysis filters.
- Function calls can only occur as statements or on the right hand side of assignment statements. Not only does this simplify the code generator but it also imposes a well defined order of function evaluation, which may be source language dependent.
- Short circuit evaluation of boolean operators must be explicitly specified using control flow operators if side-effects should be avoided. Currently the code generator has the option to short-circuit (or not) any boolean operations. This keeps the control- and data-flow aspects of the language separate, simplifying both analysis and code generation.<sup>1</sup>
- The function return statement only implies the transfer of control. Return values must be specified using an assignment operation for the same reasons as in short-circuit evaluation.
- Currently, Mirv only supports structured control operations. Most uses of `goto` in C are in fact structured, though the few unstructured uses must be rewritten.

Arbitrary `goto` instructions can complicate program analysis [6]. While these instructions could be supported in the language, in its current form Mirv only supports a structured version of the `goto` operation. This construct can be used to express many control flow constructs as well as labeled `break` and `continue` operations. The structured `goto` works as follows:

- A label operator (`destAfter` or `destBefore`) specifies a target destination that a corresponding `goto` instruction can jump to. In the case of `destAfter`, the target is the instruction after the body of the operator, `destBefore` specifies the first operator of the body as the target.
- The `goto` operator corresponding to a destination label must be in the body associated with the destination label.

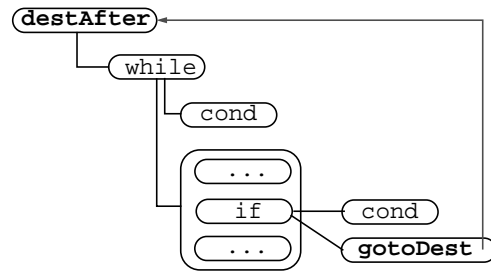
Figure 1 illustrates how a `break` statement can be represented using a structured `goto`.

---

1. Short-circuiting of expressions without side-effects does not affect dataflow analysis because no new definitions can be introduced in the expression.

**FIGURE 1. Structured goto**

```
while(a < x) {
    ...
    if (a == y) break;
    ...
}
```



This figure illustrates how a C style break construct is converted into Mirv constructs. The `destAfter` operator specifies a destination target (meaning that the target of an associated `goto` instruction is after the body of the `destAfter` operator). The `gotoDest` operator is used to jump to the `destAfter` label and achieve the same effect as a break statement in C.

The design of the structured goto was greatly influenced by the desire for a language that can be easily analyzed. For these reasons the destination label is a prefix construct (the set of operators that the label is before or after is given as an argument to the label operator) and specifies exactly which entry point (before or after the enclosed block) it delimits.

To support the C `goto` operation, the compiler provides a transformation pass that resolves labels and `goto`'s to their equivalent structured form. While unstructured uses of `goto` cannot be resolved, we have encountered very few such operations.

Since Mirv is a tree based representation, temporary values appear implicitly between expression nodes. Temporaries have a unique characteristic that differentiates them from other variables: they are written exactly once and are read exactly once. This behaviour completely defines the dataflow characteristics of the node so that these values need not be included in analysis steps.

Table 1 lists the operators used in Mirv. The language includes the standard set of expression and control flow constructs, however the object reference and structured goto operators need more explanation.

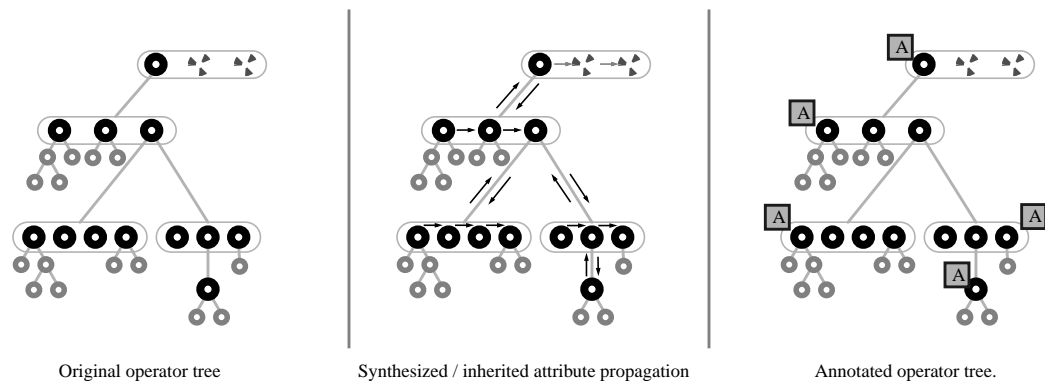
**TABLE 1. Mirv operators**

Class	Operator
<b>Arithmetic</b>	add, div, mod, mul, neg, pow, sub, sqrt
<b>Bitwise</b>	and, cpl (complement), or, rol, ror, shl, shr, xor
<b>Boolean</b>	cand, cor, eq, le, lt, ne, not, ge, gt
<b>Casting</b>	cast
<b>Control flow</b>	destAfter, destBefore, funcCall, gotoDest, if, ifElse, return
<b>Looping</b>	doWhile, while
<b>Assignment</b>	assign
<b>Object reference</b>	cref - specifies a constant value vref - specifies a variable object deref - specifies a reference through pointer aref, airef - array access through direct object or pointer vfref vfiref - field access through direct object or pointer
<b>Object size</b>	sizeof

### 3. The Mirv compilation model

The compiler consists of a set of filters that operate on the intermediate representation of the program. The internal representation is an operator tree (representing the Mirv language) where every node may contain a set of user and compiler defined attributes. Successive passes of filters communicate using these node attributes. Figure 2 illustrates the pieces in the compilation process. There are two main groups of attributes: dataflow and node attributes. Dataflow attributes are the *synthesized attributes* (sa) and *inherited attributes* (ia) [5] that are passed around a parse tree during traversal. *Node attributes* (na) are pieces of information that are associated with nodes in the operator tree.

FIGURE 2. Attributes and the compilation process



The process of annotating an operator tree is shown above. Filters traverse the operator tree, carrying inherited and synthesized attributes over the structure of the tree. These attributes are used to compute node attributes at the nodes of the operator tree. Node attributes are marked on the tree and are represented by boxed A's in the third picture. Note that some filters may also change the structure of the tree based on the value of attributes.

Filters are categorized according to their purpose into three groups. Analysis filters traverse the tree in some order (usually forward or backward) and perform computation using dataflow attributes that are propagated during tree traversal. A reaching definition analysis is one such filter. Transformation filters change the structure of the tree by adding and removing nodes. Copy propagation is classified under this category. Snapshot filters simply traverse the tree and invoke methods on external objects based on the operators and node attributes in the tree. An example is a filter to linearize the tree to a prefix-form output file.

By coordinating the order in which these three types of filters are run, a program can be optimized and translated to a target machine. The level of optimization can be varied by adding and removing filters and changing the order in which filters are run.

### 4. Analysis and transformation using attribute propagation

The Mirv compilation model uses inherited and synthesized attributes to propagate dataflow information through the parse tree. Node attributes are used to communicate information between filter passes. Filter construction in Mirv requires the specification of these attributes and the coding of routines that use them to perform analyses or transformations. We present a def-use analysis as an example of attribute propagation based analysis. We also include copy propagation as an example transformation filter.

TABLE 2. Attribute flow behaviour of forward dataflow analysis problems

Operator node	Member node	Attribute propagation	
		IAs for member nodes	SA from operator node
<b>destAfter</b>	body	$ia_{body} = ia_{destAfter}$	$sa_{destAfter} = \omega(na_{destAfter}, sa_{body})$
<b>destBefore</b>	body	$ia_{body} = \omega(na_{destBefore}, ia_{destBefore})$	Traverse member nodes twice. $sa_{destBefore} = sa_{body}$
<b>funcCall</b>	argList	N/A	$sa_{funcCall} = sa_{argList}$
<b>gotoDest</b>	N/A	N/A	$na_{dest} = \omega(na_{dest}, ia_{gotoDest})$ $sa_{gotoDest} = ia_{gotoDest}$
<b>doWhile</b>	body	$ia_{body} = \omega(ia_{doWhile}, sa_{condition})$	Traverse member nodes twice. $sa_{doWhile} = sa_{condition}$
	condition	$ia_{condition} = sa_{body}$	
<b>if</b>	condition	$ia_{condition} = ia_{if}$	$sa_{if} = \omega(sa_{condition}, sa_{body})$
	body	$ia_{body} = sa_{condition}$	
<b>ifElse</b>	condition	$ia_{condition} = ia_{ifElse}$	$sa_{ifElse} = \omega(sa_{ifBody}, sa_{elseBody})$
	ifBody	$ia_{ifBody} = sa_{condition}$	
	elseBody	$ia_{elseBody} = sa_{condition}$	
<b>sequence</b>	initial (the first node)	$ia_{seqNode} = ia_{sequence}$	$sa_{sequence} = sa_{last\ node}$
	sequential node	$ia_{seqNode} = sa_{previous\ node}$	
<b>while</b>	condition	$ia_{condition} = \omega(ia_{while}, sa_{body})$	Traverse member nodes twice. $sa_{while} = sa_{condition}$
	body	$ia_{body} = sa_{condition}$	

## 4.1. Attribute-based dataflow analysis

Dataflow analysis of a Mirv program exploits the availability of high level information about the program. Both the control-flow structure as well as the expression subtrees are available in a high level, structured form. The availability of this information allows us to use a modified form of structural dataflow analysis [8][9]. Unlike in many other compilers, the high level control-flow structure need not be resynthesized from the basic-block level since it is inherent in the representation. Moreover, the basic units of analysis are not quads in basic blocks but rather operators in the Mirv tree. Mirv carries the data dependence tree for expressions, so a tree-based analysis method is a natural fit.

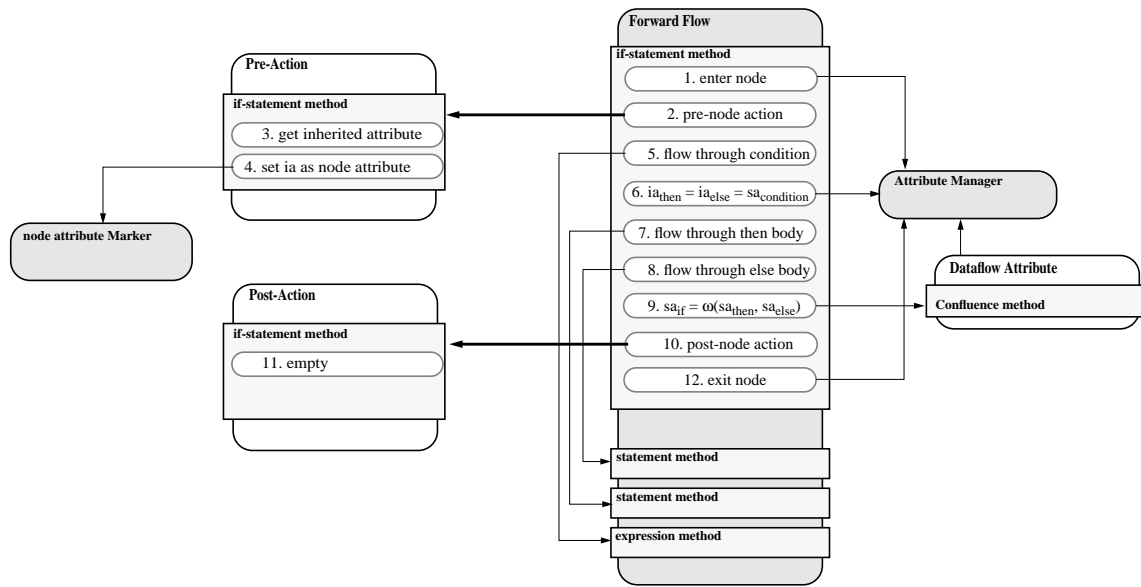
Dataflow analysis problems can be characterized as forward or backward by the direction of traversal over the program representation. They can be further divided into two groups by classifying them as *may* or *must* problems [6]. May and must problems in a given direction usually differ only in the confluence operator that is used to combine the attributes at control join points.

In Mirv, dataflow calculations are represented by the method used to generate inherited and synthesized attributes during parsing. Table 2 describes the forward attribute flow behaviour corresponding to statements in Mirv. The attribute flow tables denote how attributes are propagated between operators in the language. As an example, let us examine how attributes are propagated through an `ifElse` node during forward analysis:

1. The inherited attribute is received for the `ifElse` node. The inherited attribute of the `ifElse` node is used as the inherited attribute of the condition part of the node. After setting up the inherited attribute for the member node, the filter visits it and a synthesized attribute is received from it.

2. After traversing the condition, inherited attributes are set up for the `ifBody` and `elseBody` member nodes. In this case this the nodes receive the attribute synthesized by the condition.
3. The synthesized attributes from the `ifBody` and `elseBody` nodes are combined using the confluence operator (represented by the  $\omega$  symbol in the tables). This operator usually takes the intersection in must problems or the union of two sets in may problems. The synthesized attribute of the `ifElse` node is simply the output of the confluence operation.

**FIGURE 3. Def-Use operation for an if-else node**



The significant objects that participate in dataflow computation are shown above. Objects are represented by rounded rectangles while object methods are shown as shaded rectangles over their corresponding objects. Shaded objects are provided by the compiler infrastructure while unshaded objects are written by the filter designer.

In certain cases (`destBefore`, `doWhile` and `while` nodes in forward analysis), there is a need to iterate over a subtree of operators because a circular dependency exists between inherited and synthesized attributes of member nodes. For example, the inherited attribute of a `while` condition is dependent on the synthesized attribute of the body, while the inherited attribute of the body is dependent on the synthesized attribute of the condition.

To resolve the circular dependency, the member nodes are traversed iteratively until the synthesized attributes reach a steady state, after which the synthesized attribute for the entire subtree can be computed. In practice, the synthesized attributes reach a steady state after at most two iterations (per node initiating the iteration) in Mirv programs, since the control-flow graph is always reducible.

Analysis of the structured goto operators will be detailed in the final version of this report.

## 4.2. Implementation and def-use analysis

To illustrate how a dataflow problem may be solved using attribute propagation in Mirv, we present the implementation of the def-use analysis filter. The purpose of def-use analysis is to associate a definition of a variable with its uses. The information provided by the filter is used extensively in the current version of the compiler.

There are two pieces of information provided by the filter. The first is a set of def-use (D-U) and use-def (U-D) chains. For a particular definition of an object, the D-U chain contains a list of object uses that the definition may

reach. A U-D chain provides a list of definitions that reach a particular use. In addition to the D-U and U-D chains, the filter annotates each statement with a list of all definitions that reach that program point (i.e. the filter performs a *reaching definition analysis*).

Traditional iterative analysis calculates two sets of information for each basic block: a GEN set and a KILL set. The GEN set contains those definitions that reach the end of the basic block. The KILL set contains those definitions killed by definitions appearing in the basic block. A definition is killed if the object it defines is the target of a later definition in this basic block. These sets are then propagated through the control tree to perform the analysis.

In the Mirv compiler, these sets are never explicitly constructed. Instead, an object known as an *attribute flow* traverses the program control graph. During traversal it invokes *action* objects which perform the computation. Both pre- and post-actions may be used. A pre-action is invoked before any children of the node are visited. A post-action is invoked after all children have been visited. An action object may request inherited attributes from an *attribute manager*. These attributes contain the dataflow information relevant to the current node being visited. In addition, the action objects sets its synthesized attribute containing the updated version of the dataflow information produced after analysis of the current node. To perform the dataflow computation, action objects request an inherited attribute which is the dataflow information flowing into the node. The information is modified by the action and the result is propagated as a synthesized attribute. A control-flow node such as an if-else or looping construct must have the dataflow information from its child nodes combined in a conservative manner to represent the possibilities of the various control flow paths. This is done by the flow automatically, provided that the filter designer provides a correct confluence operator. Operation of the def-use filter on an if-else node is diagrammed in Figure 3.

To construct most dataflow analysis filters, the programmer must provide the following:

- A dataflow attribute class with associated confluence operator
- Pre-action and/or post-action objects

The compiler infrastructure provides:

- A flow object to traverse the program
- An attribute manager to track attribute state

The dataflow attribute class for the def-use filter is simply a set of definitions that are currently live. The confluence operator is set union because the reaching definition analysis provides a set of definitions that *may* reach a given program point. The set and confluence operation can be represented in many ways. The usual method is to use a bit vector where each bit represents a particular definition. The confluence operation is performed with a bitwise-OR.

As the flow visits each node in the program tree, it invokes the action objects. The action objects need only implement methods for the types of nodes relevant to the filter. The def-use filter uses both pre- and post-actions. The pre-action provides methods for function entry and statements. The function entry action is used to initialize the analysis. Definitions are created for each parameter and global object. The statement action merely copies the current reaching definition dataflow information to a node attribute which is then attached to the statement.

The post-action manages all of the dataflow computation. Methods are provided for each type of object reference, for function call expressions and for return statements. An object reference may appear in a definition or a use context. Each method queries the flow to determine in which context the reference appears.

A definition-context object reference causes the creation of a definition object for the data object. Each definition of the same data object currently in the dataflow attribute is removed. The new definition is then added to the dataflow attribute. Finally, the definition object is attached as a node attribute.

An object reference appearing in a use context causes the creation of a use object. The current reaching definitions are extracted from the dataflow attribute. Each definition of the object currently used is added to the use object, creating a U-D chain. In addition, the use object is added to each definition of the object, adding to the D-U chains. The use and definition objects are attached to their corresponding nodes with the updated chain information.

The information computed during the def-use analysis is used by later filters to perform further analyses or optimizations.

### 4.3. Copy propagation

The copy propagation filter is a transformation filter that performs traditional copy propagation using D-U and U-D chains. This filter first runs the Def-Use Analysis filter to annotate the tree with use, def, and reaching definition node attributes.

The filter examines each assignment in the program. For each use of the assignment target, the filter checks the use's def set, contained in its U-D chain, to see if the above left-hand-side variable is its only possible def. If there is more than one reaching definition, the copy cannot be propagated. The filter then checks the reaching definition node attributes of the assignment and of the statement the use appears in to see if the the reaching definitions of the right-hand-side are the same. If they are not, the right-hand-side has been redefined in an intervening statement and propagation cannot occur. If both conditions are met, the use is replaced by a copy of the right-hand-side using the use's Replacement node attribute. The replacement node attribute keeps track of a node's parent and the method of the parent that must be called to replace the node.

The code specific to copy propagation was written in 150 lines of code (not including comments).

## 5. Measurements

In this section we present some results from the prototype Mirv compiler and compare the generated code to that produced by lcc and gcc. We use lcc version 4.1 and the egcs 1.1 version of gcc. The prototype compiler accepts C as input and produces IA32 assembly code. All experiments were run on a 300 MHz Pentium II with 512K of L2 cache.

**TABLE 3.** Performance of Mirv compiler compared to GCC and LCC

Benchmark	Size of code segment (bytes)			Execution time (seconds)		
	Mirv	gcc	lcc	Mirv	gcc	lcc
compress 95	11731	7776	6806	343	258	331
eqntott 92	32702	29906	22998	8.76	6.57	8.83
go 95	271197	336997	-	28.07	21.18	-
adpcm-decode	6208	6519	2339	2.33	2.27	2.18
g721-encode	15250	14794	9637	7.25	4.57	5.66

Table 3 provides preliminary statistics about our compiler. Currently, code quality is comparable to lcc and to gcc. Note that lcc could not successfully compile the go benchmark.

We are currently running regression tests on several optimization filters and we plan to give detailed analysis of their efficacy in the final version of this report. These filters are shown on the right side of Table 4.

**TABLE 4.** Attribute flow behaviour of forward dataflow analysis problems

Infrastructure		Filter code	
Description	Lines of Code	Description	Lines of Code
Forward Traversal	2300	Constant folding	900
Attribute management	450	Arithmetic simplification	790
Def-Use analysis	2600	Dead code elimination	1467
Alias analysis	2200	Constant/copy propagation	475
Replacement decoration	692	Loop invariant code motion	680
Code cleanup	214	Strength reduction	450
Call-graph construction	620	Loop induction variable opts.	713
Output routines	1575	Function inlining	860
IR representation	17000	Loop unrolling	1150

By comparison, the optimization filters are markedly smaller than in the support framework, even though in this prototype little effort has been made to reduce code size.

Another filter, the MirvSim simulator, has been described in [13]. This filter uses the same attribute propagation technique as the other filters but instead of analyzing or optimizing the Mirv tree, it simulates it. The simulator can be configured with observer objects which track “interesting” events during simulation (such as function side effects, call frequency, etc.).

## 6. Related Work

To represent a program in memory, many optimizing compilers choose a fairly low-level representation to expose as much machine-specific computation as possible to allow optimization of elements abstracted by the high-level language, such as array index calculations. Typical examples are the IMPACT compiler [2] and the Hewlett-Packard low-level optimizer [14]. Low-level representation allows machine-specific program transformations. Code scheduling in particular is possible because computations are not hidden behind high-level constructs.

The Stanford University Intermediate Form [11] provides a higher-level representation of the program. The goal of the SUIF project is to provide source-to-source transformations of C programs for parallel machines. Because the output of the compiler is C source code, the internal representation of the program must preserve enough information to allow production of a valid C program.

Structural dataflow analysis [9] attempts to identify well-known structural elements in the control graph, such as if-else structures, loops, sequences and so forth. Flow functions are defined that describe the effects of the control structure. These functions describe how to combine the GEN and KILL sets of child nodes to create new GEN and KILL sets for the parent. Looping constructs result in equations utilizing the Kleene closure operation, analogous to the iteration in iterative dataflow analysis.

The Mirv dataflow model lies somewhere between iterative and structural approaches. One advantage of the Mirv intermediate form is the preservation of control flow. No time is spent reconstructing basic blocks or control

graphs because they are implied by the structure of the Mirv tree unlike structural analysis, which must build basic blocks and a control graph.

One of the major benefits of the Mirv compiler is the pre-built dataflow infrastructure. Describing and implementing a dataflow analyzer often requires only the definition of a few C++ classes. The idea that dataflow analyzer construction can be automated has been explored by several researchers. Tjiang and Hennessy developed Sharlit, which is to dataflow analysis what Yacc is to parsing [12]. A control graph is described and actions are provided to drive the analyzer. Sharlit computes a set of paths through the control graph and flow functions (also given by the designer) are used to propagate the information.

Sharlit differs from the Mirv approach in two important ways. Sharlit allows the designer to vary the granularity of the analysis. In contrast, the Mirv framework always examines every node in the program tree. To the programmer, it appears as though the granularity can be varied (by only providing actions for relevant nodes), but the dataflow engine will propagate information through the entire tree, even if it is not updated at some nodes. Sharlit uses a control graph description file analogous to a Yacc grammar. In contrast, a programmer working with Mirv must define a few C++ classes. Usually these definitions are straightforward, if sometimes tedious.

The Carnegie-Mellon C Compiler (CMUCC) dataflow analyzer is probably most similar to the framework in the Mirv compiler [10]. Like Mirv, dataflow analysis is describe through the definition of C++ classes. The major difference between the systems is while CMUCC works on a low-level quad representation, Mirv uses a high-level tree based IR.

## 7. Conclusions

We have presented a compiler framework that is centered around the Mirv high level program representation and attribute propagation based techniques. The design of the environment was greatly aided by our ability to concurrently develop both the language used for the program representation as well as the techniques used by the compiler. While we have a core system that is being used for various projects, we are still pursuing additions and improvements to the framework. Among others, these include the following:

- The current set of operators have been greatly influenced by our source language: C. Given different source languages, other operators may become useful.
- What information should be preserved in Mirv to support object-oriented languages?
- Development and refinement of optimization filters.

While an attempt is made in Mirv to preserve the information apparent in a high level representation, this goal must be balanced with the need to perform optimizations as Mirv-to-Mirv transformations. The computation that is implicit in some high level operators (such as array or structure member reference) must be exposed to optimizers. Node attributes can be used to retain some of the high level information.

## 8. Acknowledgments

We wish to thank the Edison Design Group for the generous donation of their C++ front-end. This work was supported by DARPA contract DABT63-97-C-0047. Some of the hardware was provided through an Intel Technology for Education 2000 grant.

## References

- [1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, pp. 279-342 (1986).
- [2] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, 19(3), pp. 266-275, ACM Press, (1991).
- [3] Chris Fraser and David Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, (1995).
- [4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley (1995).
- [5] Donald E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, Vol. 2, Springer-Verlag, pp. 127-145 (1968).
- [6] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco (1997).
- [7] Terence John Parr, *Language Translation Using & C++*, Automata, San Jose, CA (1997).
- [8] Barry K. Rosen, "High-Level Data Flow Analysis," *CACM*, Vol. 20, No. 10, pp. 712-724 (1977).
- [9] M. Sharir, "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers," *Computer Languages* Vol. 5. pp. 141-153, (1980).
- [10] Ali-Reza Adl Tabatabai, Thomas Gross and Guei-Yuan Lueh, "Code Reuse in an Optimizing Compiler," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 51-68 (1996).
- [11] Robert P. Wilson et al., "The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler," Stanford University Technical Report CSL-TR-94-620, (1994).
- [12] Steven W. K. Tjiang and John L. Hennessy. Sharlit---A Tool for Building Optimizers. *ACM SIGPLAN '92 Conf. Programming Language Design and Implementation*, pp. 213-226, July, (1992).
- [13] Krisztián Flautner, Gary S. Tyson, and Trevor Mudge. MirvSim: A High-Level Simulator Integrated with the Mirv Compiler. *Proc. 3rd Workshop on Interaction Between Compilers and Computer Architectures (INTERACT-3)* held at ASPLOS-VIII, (1998).
- [14] Andrew Ayers, Stuart de Jong, John Peyton and Richard Schooler. Scalable Cross-Module Optimization. *Proc. 1998 ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, vol. 33, no. 5, May 1998. pp. 301-312.