

MirvKit

A framework for compiler and computer architecture research

Krisztián Flautner (manowar@engin.umich.edu)

David A. Greene (greened@eecs.umich.edu)

Trevor N. Mudge (tnm@eecs.umich.edu)

Abstract

The MirvKit compiler framework provides a set of abstractions that includes a language for intermediate program representation, compilation methodology and an object model for the implementation. These aspects of the framework were developed concurrently, each influencing the others. A single design pattern emerged from this evolutionary process that forms the core of the compiler. This pattern, known as Attribute Flow, is used for such diverse tasks as dataflow analysis, program transformations, code generation and simulation. This paper explores the main issues behind the design and describes how dataflow analysis and simulation are implemented using attribute propagation.

1. Introduction

MirvKit is an object based framework for compiler and computer architecture research. The object model is closely tied to the Mirv language, which is used as the compiler's intermediate program representation (IR), and to the attribute propagation based compilation methodology. This technique is used throughout the compiler to perform code analysis and transformations.

MirvKit provides a set of components that can be used to represent and manipulate the high level program representation. Most of the components were written in Objective-C, using Apple's Rhapsody environment. At the core of these components is the *Attribute Flow* pattern that provides a mechanism for traversing and propagating attributes over the program representation.

The components of the MirvKit framework center on the Mirv language. The aim of this language is to provide 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 struc-

tures 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 concurrently, each asserting its influence on the other.

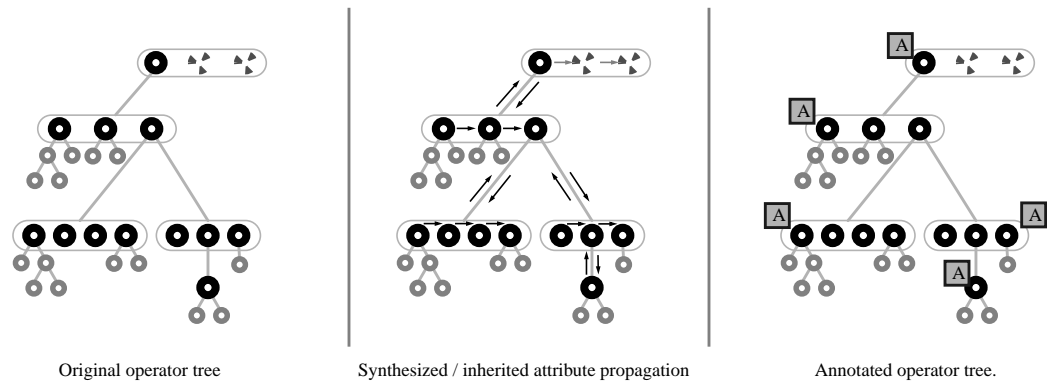
Other compiler frameworks do not tie the IR and the compilation model as closely together as MirvKit does. We did this in the hope that this will provide our design with more integrity. The trade-off is that it may be harder to extend Mirv with new operators, since the underlying assumptions of the language must always be considered. Another distinction from other compiler frameworks such as SUIF [14] and Impact [3] is that MirvKit uses only a single level IR. The aforementioned compilers use multiple levels of IRs for the different phases of compilation. While our approach uses only a single IR for the sake of simplicity, we have attempted to bridge the multiple levels of representations by the use of attributes associated with operator nodes.

Section 2 describes the compiler methodology and provides an overview of the Mirv language. Section 3 gives an introduction to attribute propagation and demonstrates how it is used to perform dataflow analysis. We provide an example of live variable analysis. Section 4 discusses the implementation of the MirvKit framework. The attribute flow pattern and class hierarchies are described. Section 5 gives an example of the Mirv Simulator, which illustrates how the attribute flow pattern can be used to solve a divergent set of problems.

2. The 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. The attributes are usually computed and used by the filters that are invoked on the tree. Successive passes of filters communicate using these node attributes. Figure 1 illustrates the pieces in the compilation process. There are two main groups of attributes: parse and node attributes. Parse attributes are the *synthesized attributes* (sa) and *inherited attributes* (ia) [7] that are passed up and down a parse tree during traversal. *Node attributes* (na) are pieces of information that are associated with nodes in the operator tree.

FIGURE 1. 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 and perform computation using parse attributes that are propagated during tree traversal. Transformation filters change the structure of the tree by adding and removing nodes. How a transformation filter changes the tree structure is usually determined by the node attributes computed by analysis filters. Snapshot filters neither set node attributes, nor do they alter the structure of the existing tree; they simply traverse the tree and invoke methods on external objects based on the operators and node attributes in the tree.

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. Dependencies between filters exist as the dependence of a filter on a set of node attributes. In other words, a filter does not explicitly specify what other filters must precede its execution but rather states the names of the node attributes it is dependent upon.

2.1 The language

Mirv maintains high level control constructs (if, while, for, function call, function parameter, etc.) rather than resolving them to lower level operations. Attributes of identifiers normally maintained within a symbol table are exposed directly in the Mirv program. 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.

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 cannot appear as function call arguments. If such behaviour is desired, then the result of the function call must be first assigned to a variable which is then used as the function argument. Not only does this simplify the code generator but it also imposes a well defined order of argument evaluation, which may be source language dependent.
- Short circuit evaluation of boolean operators must be explicitly specified using control flow operators. The existing boolean operators in the language do not short circuit. This keeps the control- and data-flow aspects of the language separate, simplifying both analysis and code generation.
- 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.

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 [8] / Yacc [6] and PCCTS [10] can be used for this task. A Syntax Directed Translation Scheme (SDTS) [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.

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

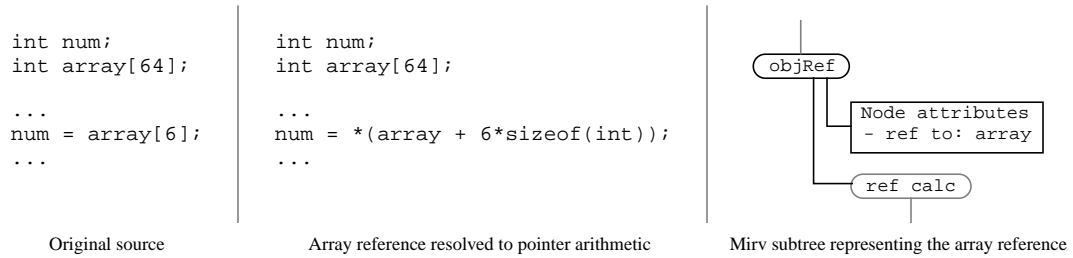
Class	Operator
Arithmetic	add, div, mod, mul, neg, pow, sub, sqrt
Bitwise	and, cpl (complement), or, rol, ror, shl, shr, xor
Boolean	cand, cor, eq, le, lt, ne, not, ge, gt
Casting	cast
Control flow	destAfter, destBefore, funcCall, gotoDest, if, ifElse, return
Looping	doWhile, while
Object reference	constObjRef - specifies a constant value assignCalculated, objRefCalculated - specify a destination or source reference through pointers assignDirect, objRefDirect - specify a destination or source by name
Object size	sizeof

In order to localize data access to a small number of operators, high level reference constructs are resolved to pointer arithmetic. However, high level information about the access is preserved through the use of node attributes. The key to this approach is deciding exactly what is important. In the case of array references, the following information is interesting:

- The location and size of the array, which bound the memory locations that are accessed as the result of the array reference. If the array reference is an indirect reference, then the type of the array is preserved. This information allows optimization filters to make better decisions about what memory locations may be referenced.
- Any access patterns (stride) that may be known based on language and profile information.

An added benefit of this approach is that any reference through pointers may have additional information associated with it. If reference patterns and aliasing information is computed about any reference, it can be encoded in the representation. The difference is in the ease of availability of this information; it is readily available for array references but may require aggressive analysis in other cases.

FIGURE 2. Array references in Mirv



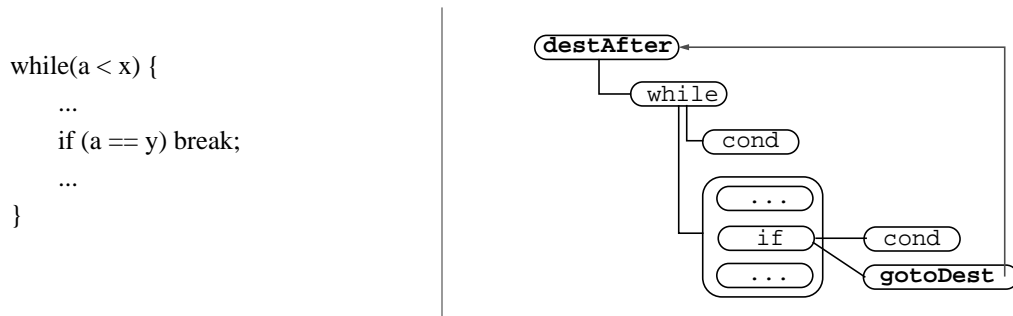
The progression of the original code (on the left) to Mirv code (on the right) is shown in this figure. The node attribute attached to the objRef operator shows how information that is apparent in the array reference is preserved, while allowing the code to be moved down to a lower level representation, enabling further optimizations.

Arbitrary goto instructions can complicate program analysis [9]. 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 3 illustrates how a break statement can be represented using a structured goto.

FIGURE 3. Structured goto



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.

3. Analysis using attribute propagation

Information is gathered about a program by propagating parse attributes over its structure. These attributes are referred to as *inherited attributes* when information is passed to a node before it is traversed, and *synthesized attributes* when information is returned after traversal [7]. Another class of attributes associate information with a node in the operator tree. These are referred to as *node attributes*.

By specifying what information is passed to the children of a node (member nodes) as inherited attributes and what information is passed up from a node as synthesized attributes, information can be collected about a program. The results of the attribute propagation are marked as node attributes on the operator tree.

3.1 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 [11][12]. Unlike in many other compilers, the high level control-flow structure does not have to 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.

In iterative dataflow analysis the structure of the program is presented as a collection of basic blocks and a control-flow graph. In order to compute dataflow information about a program, a dataflow analyzer iterates over the control graph applying equations at the basic block level until a steady state is reached.

While structural dataflow analysis also uses basic blocks as the units of computation, it assumes the existence of a hierarchical control tree. This structure can either be built up from the control-flow graph or derived from the high level program representation (as in the Mirv approach). This allows incremental updating of information as the program undergoes optimizations [9].

A variation of structural dataflow analysis is used by Mirv analysis filters because the control tree is inherent in the Mirv program representation. The control-flow characteristics of a statement are derived from its semantic meaning. Mirv also carries the data dependence tree for expressions, so a tree based analysis method is a natural fit.

While statements in the program graph define the control-flow characteristics of the program, expressions specify the data values that are accessed. All leaf expression nodes in the tree represent a variable or constant access.

A crucial difference between Mirv dataflow and classical dataflow analysis (both iterative and structural) is that in Mirv temporary variables need not be analyzed. In the structured graph temporary values appear implicitly between expression nodes. However, 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, thus requiring less work to be performed.

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 [9]. May and must problems in a given direction usually differ only in the confluence operator that is used to combine the attributes that take part in the analysis.

In Mirv, dataflow calculations are represented by the way inherited and synthesized attributes are generated during parsing. Table 2 and Table 5 describe the attribute flow behaviour corresponding to statements in Mirv.

TABLE 2. Attribute flow behaviour of backward dataflow analysis problems

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

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 backward analysis:

1. Inherited attribute is received for the ifElse node. The inherited attribute of the ifElse node is used as the inherited attribute of the elseBody 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 else part of the node, inherited attributes are set up for the ifBody member node. In this case this node also receives the same inherited attributes as the ifElse node.
3. After the ifBody member node is visited the condition member node is traversed. The inherited attributes for this node are produced from a combination of the synthesized attributes received from the previously visited member nodes. The results of these two nodes are combined using the confluence operator (represented by the ω symbol in the tables). This operator usually takes the intersection in must problems or the union of two sets in may problems.
4. The synthesized attribute of the ifElse node is simply the synthesized attribute received from the condition of the ifElse.

In certain cases (destBefore, doWhile and while nodes in backward analysis), there is a need to iterate over a subtree of operators multiple times. The reason for this is that there is a circular dependency between inherited and synthesized attributes of member nodes of a node. An example of this can be seen in the attribute flow rules of the while operator. Here, the inherited attribute of the condition is dependent on the synthesized attribute of the body, while the inherited attribute of the body is dependent on the synthesized attribute condition.

The solution to the resolution of the circular dependency is to traverse the member nodes in the prescribe order while setting the contents of the initial synthesized attributes appropriately for the given analysis. After propagating the attributes through the member nodes once, the received synthesized attributes can be used to compute a new inherited attributes for the nodes. This process can be repeated 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.

The attribute flow through the structured goto operators make use of the ability of associating attributes with individual operator nodes of the representation. The following procedure is used to compute a synthesized attribute for a destAfter node:

1. The destAfter node is visited before any of the associated gotoDest operators are reached (due to the structure of the Mirv representation). This allows the attribute propagator to store the current state of the inherited attributes as a node attribute at the destAfter node. This state corresponds to the state of the inherited attributes immediately following the body of the destination node.
2. When a gotoDest operator is reached, it can merge the node attribute of its destination into its attribute calculations to compute the correct synthesized attributes.

Note that the procedure is similar for the `destBefore` node with the difference that in that case the node attribute stored at the node is supplied by the synthesized attribute generated by a previous iteration through the body. Similarly to the while operator mentioned above, two passes over the body are sufficient to generate the correct synthesized attributes.

3.2 Live variable analysis

To illustrate how a real world dataflow problem can be solved using attribute propagation, we provide live variable analysis as an example. Live variable analysis seeks to determine whether there is a use of a variable on some path between a given point in the program and the exit.

Traditional live variable analysis uses the IN, OUT, DEF and USE sets to compute its result and works the following way [9]:

The first pass of the algorithm calculates the DEF and USE sets for every basic block in the graph. This is strictly a local analysis, where the DEF set corresponding to a basic block contains all variables that are defined (assigned to) in that block. The USE set contains all variables that are locally exposed uses of variables (i.e. uses of variables, whose definition comes from the outside of the basic block).

Calculation of the IN and OUT sets is performed as a succession of iterations over the control flow graph until the sets reach a steady state (The IN sets stops changing). The algorithm starts from the last basic block of the control flow graph and setting the OUT set of the last node to be the empty set. The following two equations are applied to every basic block in the graph as the blocks are traversed from back to front:

$$\begin{aligned}
 IN(i) &= (OUT(i) - DEF(i)) \cup USE(i) \\
 OUT(i) &= \bigcup_{j \in Succ} IN(j)
 \end{aligned}
 \tag{EQ 1}$$

Live variable analysis in Mirv uses the OUT set as the inherited attributes to nodes and receives the IN set as the result of the attribute propagation through member nodes. Since assignments in Mirv cannot be used as rvalues, the DEF and USE sets are internal information to a node and as such are not propagated.

While control-flow constructs propagate attributes, they do not usually generate them. In live variable analysis, attributes are only generated and killed by object references. Attributes are simply propagated up the expression subtrees (i.e. the synthesized attribute of the node is the combination - using the confluence operator - of the synthesized attributes if its member nodes, see Table 3). No inherited attributes need to be passed down to the expressions.

TABLE 3. Bottom-Top attribute flow behaviour of expressions

Operator node	Member node	Attribute propagation	
		IAs for member nodes	SA from operator node
Unary expression	arg		$sa_{expr} = sa_{arg}$
Binary expression	arg 1		$sa_{expr} = \omega(sa_{arg1}, sa_{arg2})$
	arg 2		

The use of a variable corresponds to the variable name being read by an operator. In traditional live variable analysis this would cause the accessed variable to show up in the USE set of the basic block. In Mirv analysis, the use information shows up as the synthesized attribute (IN set) of the operator that generated the use.

The equation for computing the IN set of a variable assignment makes use of the equation given in (Eq. 1). In traditional live variable analysis writes to variables would show up in the DEF set of the unit, however in Mirv based analysis the semantics of the DEF set are conveyed by taking out the defined variable from the inherited OUT set. The resulting synthesized attribute is the union of this set and the synthesized attribute received from the source operator subtree of the variable assignment operation.

Table 4 illustrates the attribute flow behaviour of direct object references. The OBJ_{dest} and OBJ_{src} nodes denote the names of the objects referenced by the operators. Describing the behaviour of calculated object references is more difficult and dependent on how the information would be used, and the assumptions made by the compiler.

The issue is that while direct object references specify exactly which objects they reference, calculated references only specify the run-time location of the objects they reference. Depending on the aggressiveness of the compiler different assumptions can be made about the set of objects that can be referenced from such an operator. One common policy is to assume that all objects of the referenced type can be aliased.

TABLE 4. Attribute flow of object reference operators for Live Variable analysis

Operator node	Member node	Attribute propagation	
		IAs for member nodes	SA from operator node
assignDirect	source		$sa_{objRef} = (ia_{objRef} - OBJ_{dest}) \cup sa_{source}$
objRefDirect			$sa_{objRef} = OBJ_{src}$

Table 5 lists the attribute flow characteristics of a forward dataflow problems. The issues of attribute propagation are similar to the ones described for backward dataflow problems.

In both cases, however it is apparent that the choice of the underlying language that is used for attribute flow based analysis can greatly impact the ease of such analysis.

TABLE 5. Attribute flow behaviour of forward dataflow analysis problems

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

4. Implementation

MirvKit includes the components to implement the techniques described in the previous sections. It provides a set of objects for representing Mirv programs, attributes and the filters. The components were written in Objective-C using the Foundation and AppKits under Apple's Rhapsody. The existing code generator and the C front-end are implemented in C++. The current MirvKit implementation includes the following tools for the compiler writer:

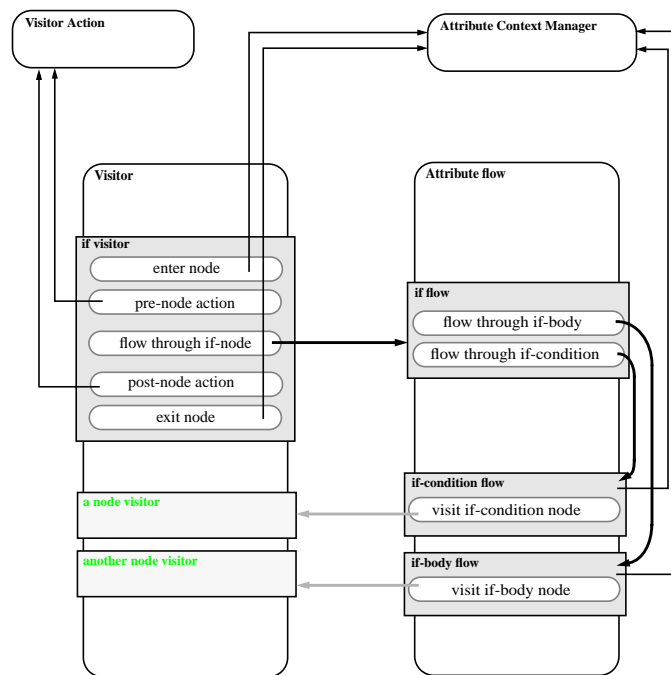
- Persistence of the compilation state; the operator along with marked up attribute information can be saved and reloaded.
- Expression and statement evaluation along with the Mirv simulator.
- Filter component hierarchies based on the Attribute Flow pattern.
- Visualization of information related to the program representation.

These components provide the necessary infrastructure for building a variety of code transformation filters as well as the ability to simulate the resulting program as described in Section 5.

4.1 The Attribute Flow pattern

The previous sections describe how various analyses can be performed using attribute propagation. These filters all assume an underlying mechanism that can be used to manage the flow of attributes. Standard parsing techniques and parsing tools such as Lex/Yacc could be used as foundations for some of the filters. While Lex and Yacc can be used to efficiently drive certain filters such as our code generator, it can prove to be cumbersome if flexibility is desired in the way attributes are propagated. To provide a foundation for building filters, the Attribute Flow pattern was designed.

FIGURE 4. Interaction of the visitor and the attribute flow objects



The image illustrates the visitor and attribute flow actions that occur when an if-node is traversed. Bold arrows represent the flow of control between methods, while control flows top-down inside methods. Depending on the problem, both the attribute flow object and visitor action objects may also access and/or modify state in the Attribute Context Manager. The particular attribute flow object in the picture visits the operator tree nodes backwards.

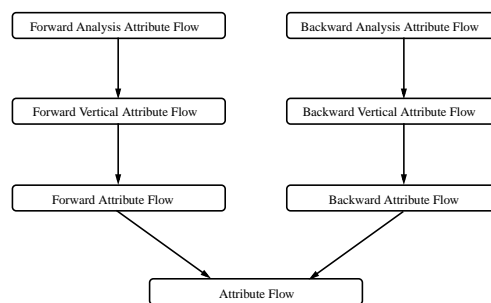
The Attribute Flow pattern is an extension of the visitor pattern and consists of the following pieces:

- The Visitor object implements the Visitor pattern [4] and is used to invoke a function based on the type of an object in the operator tree. The Visitor class has methods corresponding to all the operators in Mirv and they invoke a predetermined set of methods on external objects. The Visitor object usually does not change from one filter to another.
- The Attribute Flow object encapsulates the attribute propagation behaviour of a filter. It has two main roles: it determines the order in which the tree is traversed and it specifies

how attributes are propagated between nodes in the tree. These two tasks are related, since attribute propagation also implies a traversal order. The Attribute Flow class has methods corresponding to all the operators in Mirv as well as to all member nodes that these operators have. The methods corresponding to member nodes of operators are used to provide a context to nodes based on the type of their parent.

- The prefix and postfix Visitor Action objects are called from the Visitor and are used to perform certain tasks before and after a node is visited. Among other things, these actions can be used to print out textual representation of the tree or to mark parse attributes at the nodes of the tree.

FIGURE 5. The Attribute Flow hierarchy



The figure illustrates the Attribute Flow hierarchy in Mirv. While other user defined flows (especially for code generation) are possible, these flows can be used to perform all the different kinds of dataflow analyses.

- The Attribute Context object keeps track of the inherited and synthesized attributes of both parent and member nodes.

Figure 4 provides a high level view of the interaction between the parts of the Attribute Flow pattern. The attribute flow object can be configured with various actions and context managers to create new filters. However, common attribute propagation behaviour of filters can also be factored out through inheritance. This is particularly useful for dataflow analysis, where most of the problems can be classified into a small set of categories.

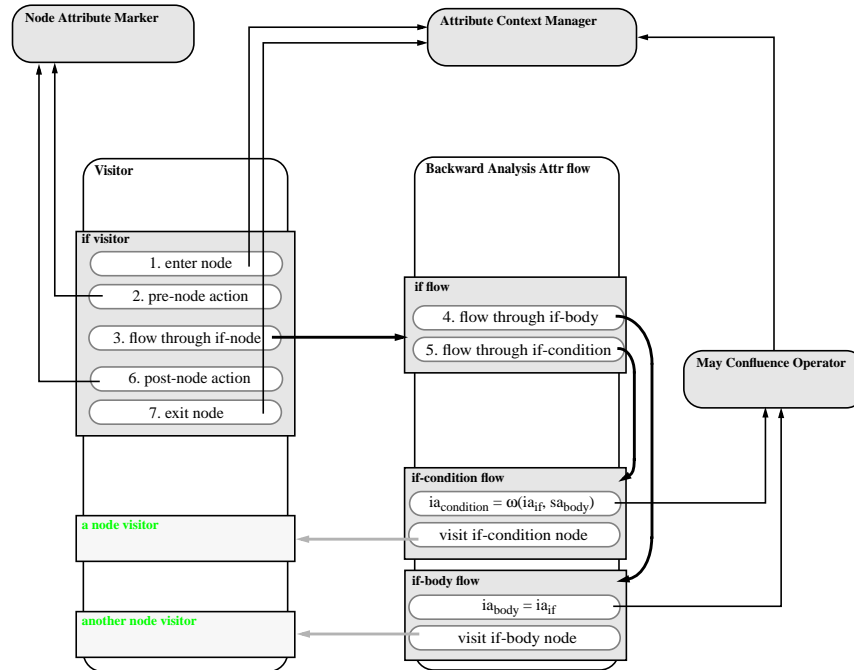
The current class hierarchy is illustrated in Figure 5. The Forward and Backward Analysis classes can be used to perform any forward and backward dataflow analysis problem. Both of these analyses have two variants corresponding to may or must dataflow problems. However, these are not represented as derived classes in the hierarchy but rather as a confluence operator parameter to the particular analysis attribute flow object. This approach contrasts with the solution in [13], where the different confluence operators are expressed through inheritance.

The Vertical Attribute Flow classes propagate attributes top-down and bottom-up through expression subtrees. This means that expression member nodes all receive the same inherited attributes and that synthesized attributes are simply combined in some way and propagated up the parse tree. Similarly to the analysis classes, the manner in which the attributes are combined is specified by a parameter, such as the confluence operator. Meth-

ods in these classes are used by the analysis classes, but they are very useful on their own for generating formatted visual representations of trees or to drive the Mirv simulator.

The Backward and Forward attribute flow classes only specify the order in which member nodes of a given node are traversed, however they leave the implementation of flows associated with members up to the derived classes.

FIGURE 6. Objects involved in a backward may attribute flow through an if node



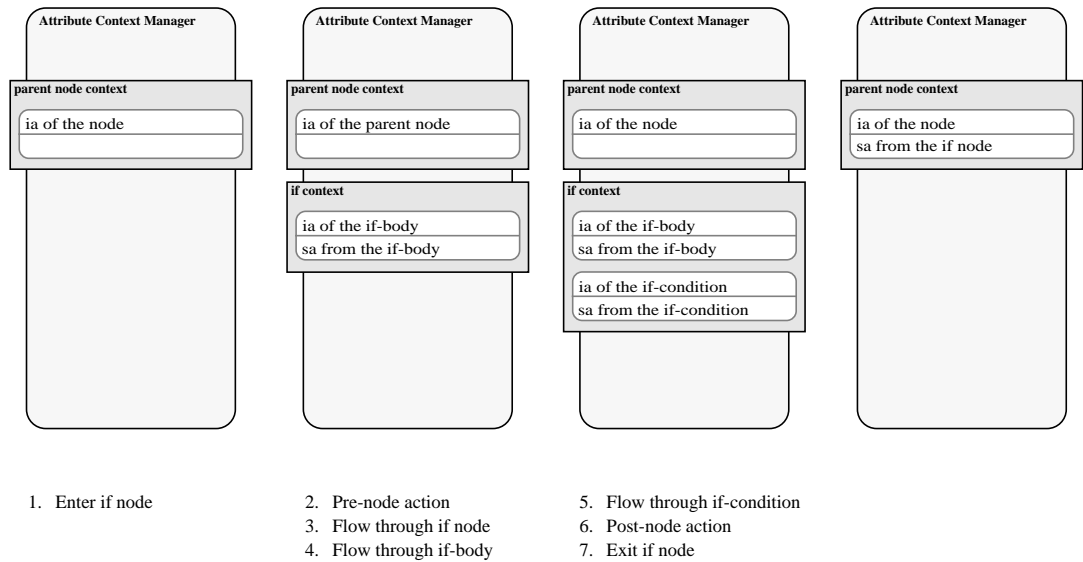
This figure is an expanded version of Figure . The significant objects that take place in the computations are shown above. A significant addition is the May Attribute Creator object. This object is responsible for performing the correct computations for a May dataflow problem. In this example, the ω operation in the if-condition flow method is implemented as a union operation by this object. For a must-problem, this operation would be different.

To illustrate the behaviour of the use of the Attribute Flow pattern for dataflow analysis, the steps taken when computing the live variable attributes of an if node are described below and illustrated in Figure 6. The state of the attribute stack during the analysis is presented in Figure 7.

1. The if node is visited and is provided a set of inherited attributes. These attributes are copied into a node attribute context that is set up when the “enterNode” method is invoked on the Attribute Context Manager. The appropriate visitor action method is invoked on the pre-node visitor action, which in this case would take the node’s inherited attributes and save them as a the OUT node attribute.
2. The if node attribute flow method is invoked, which is used to sequence the order of traversal of the member nodes. In backward analysis the body is visited before the condition.

- The attribute flow of the if-body member node is invoked. The method creates a member node context and sets up the inherited attributes for the node. In this case, the inherited attributes of the if node are simply propagated to the body. The attribute flow method invokes the appropriate visitor method on the Visitor and the attribute flow process continues with the visitation of the body.

FIGURE 7. Computing attributes for an if statement



The state of the Attribute Context Manager is shown as an if node is processed. The list of actions under the image of the current state of the Context stack outline what steps are invoked before the state of the context stack changes. Note that the steps are not identical for all operators. The numbers and actions at the bottom of the figure correspond to the actions shown in Figure 6.

- The attribute flow of the if-condition member node is invoked. A member node context is created in the Attribute Context Manager and the appropriate inherited attributes are passed to the node. In this case, the inherited attribute consists of the combination of the synthesized attribute of the if-body and the inherited attribute of the if node. The appropriate visitor method is invoked for the if-condition on the Visitor object.
- After the attribute flow methods are completed, execution continues in the if-visitor method. The post-node action is invoked, which saves the last member node's synthesized attribute as the IN attribute of the if-node. Having marked the resulting node attribute, the Attribute Context Manager frees up the space associated with the node's member contexts and propagates the resulting synthesized attribute to the enclosing member context's synthesized attribute slot.

5. The Mirv simulator

While the attribute propagation ideas can be used to easily describe dataflow problems, the same methodology can also be used to solve many other problems. One example of this is

a simulator that executes the high level Mirv program representation. This model allows the compiler to easily obtain information about the execution profile of the program given a sample input set. Information can be communicated between the simulator and analysis filters using node attributes.

The simulator is built upon the same Attribute Flow pattern as used in the dataflow analysis filters. The simulator can be configured with visitor action objects that collect information about the program's behaviour. This information can include information about the program's profile, the range of values produced by the expressions or even information about the frequency that certain control-flow paths are taken.

The Simulator attribute flow is built upon the forward attribute flow class, where all the control flow nodes are overridden to perform the appropriate control-flow behaviour. The behaviours associated with the operator nodes are outlined in Table 6.

TABLE 6. Attribute flow behaviour of the Mirv simulator

Operator node	Behaviour
destAfter	<ul style="list-style-type: none"> •Enter a nested exception handler context. •Flow through the body. •If an associated exception is caught, then return from the node context.
destBefore	<ul style="list-style-type: none"> •Enter a nester exception handler context. •Flow through the body. •If an associated exception is caught, then flow through the body again.
funcCall	<ul style="list-style-type: none"> •Flow through the function call arguments. •Set up the called function's evaluation context. •Visit the called function.
gotoDest	<ul style="list-style-type: none"> •Throw an exception (include the destination name in it).
doWhile	<ul style="list-style-type: none"> •Flow through the function body. •Flow through the condition. •If the condition is true, repeat the first two step, else return.
if	<ul style="list-style-type: none"> •Flow through the condition. •If the condition is true, then flow through the body.
ifElse	<ul style="list-style-type: none"> •Flow through the condition. •If the condition is true, then flow through the ifBody, else flow through the elseBody.
sequence	<ul style="list-style-type: none"> •Flow through every element in the sequence in forward order.
while	<ul style="list-style-type: none"> •Flow through the condition. •If the condition is true, then flow through the body, else return. •Repeat the first two steps.

Expressions are evaluated using value objects that implement all the operations specified by Mirv. The result of an expression subtree is a synthesized attribute containing the value generated by the subtree.

Since Mirv supports the addressing of objects through pointer arithmetic, one must find an efficient mapping between the objects in the system and the assumed memory model. Furthermore, pointer casting (which is not disallowed in Mirv) must be supported.

In order to deal with the aforementioned issues, one must run an allocator filter over the program before the simulator is run. The allocator examines all the objects in the system and assigns them locations. Memory locations are assigned in one of the following memory areas:

- Global area - for values that are visible from all of the functions.
- A local area for local variables.
- An argument area, where the function's arguments are stored.
- A return area, where the function's return values are stored.

Object locations in the local, argument and return areas exist for each function and may contain unique values on a per function invocation basis. A single global area is responsible for managing global values.

Once the mappings of objects to memory locations have been established, mappings from locations to objects are stored in interval trees and are accessed by object reference operators. To keep track of the appropriate memory areas, an evaluation context is passed to the object reference operators containing the interval trees and associated objects for the individual memory areas.

6. Conclusions

We have presented a unified framework for analyzing and manipulating the Mirv high level program representation. The methodology centers around attribute propagation over the structure of the Mirv language using the Attribute Flow design pattern. The design of the framework 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:

- Graphical visualization of information about the programs undergoing compilation. This includes information about the program's operator tree, variables, types and node attributes. Currently this information is only presented as text.
- Adding support for source languages other than C, such as Java.
- The previous point brings up the need to consider what information needs to be preserved in Mirv to support object-oriented languages. What information from the class hierarchy is useful for preservation in Mirv?

We are planning on investigating optimization techniques that take advantage of the Mirv simulator. These include profile based optimizations and communication of compile time and profile information to the target architecture. Currently we have a code generator for x86 based machines and are working targeting the compiler to SimpleScalar [2] to facilitate computer architecture research.

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] Doug Burger and Todd M. Austin “The SimpleScalar Tool Set, Version 2.0,” Technical Report #1342, Computer Science Department, University of Wisconsin, (1997).
- [3] 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).
- [4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley (1995).
- [5] Mirv technical report to be published May, 1998.
- [6] S. C. Johnson, “Yacc - Yet Another Compiler Compiler,” Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J. (1975).
- [7] Donald E. Knuth, “Semantics of Context-Free Languages,” *Mathematical Systems Theory*, Vol. 2, Springer-Verlag, pp. 127-145 (1968).
- [8] M. E. Lesk, “Lex - A Lexical Analyzer Generator,” Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J. (1975).
- [9] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco (1997).
- [10] Terence John Parr, *Language Translation Using & C++*, Automata, San Jose, CA (1997).
- [11] Barry K. Rosen, “High-Level Data Flow Analysis,” *CACM*, Vol. 20, No. 10, pp. 712-724 (1977).
- [12] M. Sharir, “Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers,” *Computer Languages* Vol. 5. pp. 141-153, (1980).
- [13] 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).
- [14] Robert P. Wilson et al., “The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler,” Stanford University Technical Report CSL-TR-94-620, (1994).