A Retargetable Integrated Code Improver

Manuel E. Benitez
Jack W. Davidson

Computer Science Report No. CS-93-64
November 15, 1993

A Retargetable Integrated Code Improver

Manuel E. Benitez
meb1u@virginia.edu
(804) 982-2296

Jack W. Davidson
jwd@virginia.edu
(804) 982-2209

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

Abstract

We present a retargetable, machine-level framework that tightly integrates the three primary functions performed by an optimizer: code generation, register allocation and code improvements. This framework is highly retargetable, able to fully exploit unique architectural features and, unlike most integrated frameworks, easily extended to include a comprehensive set of local and global optimizations. A unique approach to performing transformations improves register allocation, reduces interactions between the code improvement phases and enhances retargetability by eliminating the need to determine an effective phase ordering for each target architecture.

1 Introduction

Many retargetable optimizing compilers perform code improvements on fixed, high-level intermediate representations [TSK+83] [CH84]. While these representations promote retargetability by allowing much of the optimizer to be machine-independent, they make it difficult to exploit the unique architectural features of each target machine. Alternatively, a well-designed low-level intermediate representation based on register transfer lists (RTL) [DF80] can be manipulated using machine-independent algorithms to perform machine-specific optimizations such as peephole optimization and instruction scheduling effectively. These representations have made highly-retargetable, high-quality code generation a reality, as evidenced by the fact that both *vpo* [BD88], which has been targeted to more than a dozen different architectures, and the GNU **C** compiler [Sta89], which has been targeted to more than 20, produce code whose quality meets or exceeds that of many machine-specific optimizing compilers.

Low-level intermediate representations make it feasible to operate exclusively at the machine-level, where the cost and benefits of each transformation can be accurately gauged. Unfortunately, the register allocation and phase ordering techniques used to optimize high-level intermediate code do not allow the optimizer to fully utilize accurate cost and benefit information. Thus, the full potential of low-level intermediate representations will not be realized until techniques that give the optimizer more control over register allocation and phase ordering are developed.

How does a common register allocation strategy such as priority-based coloring [CH84] undermine the ability of a machine-level optimizer to use cost and benefit information? Priority-based register coloring schemes initially allocate values to memory and promote them to registers after all code improvement transformations have been applied. Consider the machine-level code shown in Figure 1(a), which contains a redundant computation. Assume that the optimizer has determined that the most effective way to improve this code is to extend the life of the register containing the sum so that it reaches the store and remove the second add instruction as shown in Figure 1(b). Since this transformation cannot be performed directly under a priority-based coloring strategy, a new value is created and allocated to memory as shown in Figure 1(c). Although this initially reduces the quality of the code, we assume that the

```
r[3]=r[5]+1;                r[3]=r[5]+1;               r[3]=r[5]+1;
                                                       M[val_x]=r[3];
...                         ...                        ...
r[8]=r[5]+1;                                           r[8]=M[val_x];
M[r[6]]=r[8];               M[r[6]]=r[8];              M[r[6]]=r[8];
     (a)                         (b)                        (c)
```

Figure 1: Priority-based coloring at the machine-level

register allocator will promote the value to the appropriate register and produce the code in Figure 1(b). Unfortunately, there is no guarantee that this will happen, especially when the demand for registers is high. Although the cost and benefit information available to the common subexpression elimination phase allows it to determine that a transformation is worth applying only if a register can be obtained, the priority-based coloring framework does not give the optimization enough feedback to determine if it should attempt the transformation.

Because the separation between register allocation and code improvement is an integral part of priority-based register coloring, we abstained from using it in *vpo* in favor of an on-demand register allocator that provides instant feedback regarding the allocation status of each new value. While this approach allows the optimization phases to avoid performing optimizations that do not appear beneficial at compile-time, it increases importance of phase ordering because the early phases tend to monopolize the registers.

Phase interaction and phase ordering problems are not unique to on-demand register allocation schemes, but occur in all optimizing compilers. They are more pronounced in machine-level retargetable optimizers, because the same source code will often have different phase ordering requirements when translated to machine code on different machines. Consider the following code fragment:

```
extern int a[];

int foo(i)
int i;
{
     ...
     while (a[i]) {
          ...
          }
     ...
     return(i);
}
```

There are no other references to *i* or *a* within the function. Portions of the code produced for this function on a RISC and a CISC machine prior to any global optimizations are shown in Figure 2(a) and (b), respectively. In each case the loop code is contained between the `loop` label and the “`PC=loop`” instruction. On the RISC machine, the base address of the array is computed by combining the upper and lower portions of the address. The value of *i* is loaded from memory, multiplied using a left shift operation and added to the base address of the array by the instruction that loads the array element. The corresponding CISC code consists of two instructions. The first loads and multiplies the value of *i* while the second adds in the base address of the array and loads the array element.

Assume that other transformations have been applied and there is only one register available within the loop. If the local variable promotion phase, which allocates local and argument variables to registers, is invoked next, it promotes *i*. The resulting code is shown in Figure 2(c) and (d). On the RISC machine, the transformation removed one instruction from the loop and replaced the load at the end of the function with a register-to-register transfer. On the

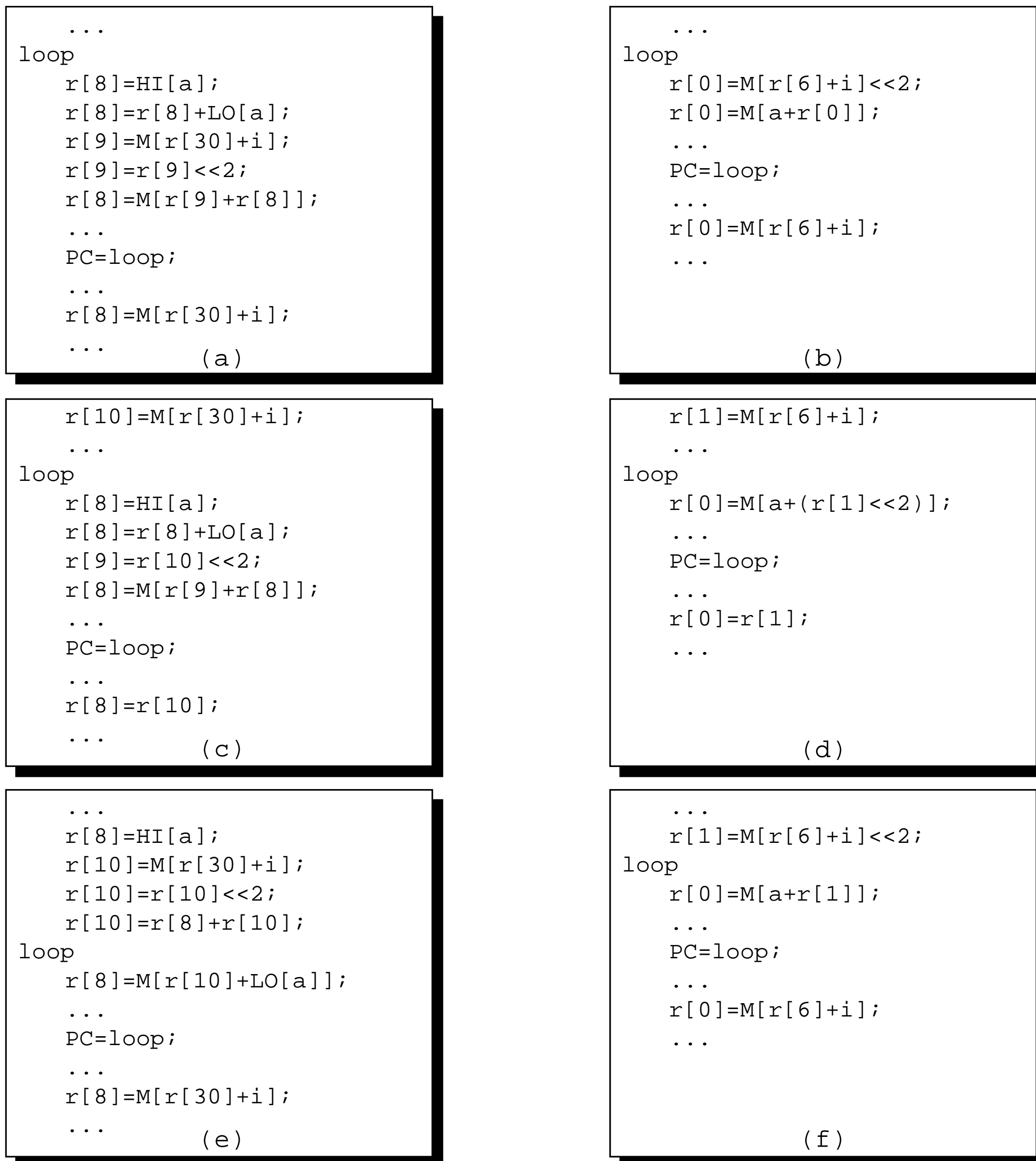

Figure 2: Machine-level code on two different architectures

CISC machine, the promotion had similar effects. Alternatively, if the loop-invariant code motion phase had been next in the invocation order, the code shown in Figure 2(e) and (f) would have been produced. In the case of the RISC machine, code motion removed three instructions from the loop and, despite the fact that it did not eliminate the load at the end of the function, made better use of the single remaining register than the local variable promotion phase did. Does this mean that loop-invariant code motion should always be performed before local variable promotion? The CISC results do not agree because although code motion moved the extra instruction out of the loop, it could not eliminate the load at the end of the function.

This simple example suggests that it is not possible to choose a single static phase ordering that produces the best results under all circumstances. In practice, phase interactions get more complex and phase ordering more difficult as the number of optimization phases and the range of architectures that the compiler is expected to handle increases. In

the *vpo* compiler, we reduced the impact of phase ordering problems through phase iteration [BD88]. While this technique was partially successful, it cannot guarantee that the registers will be used to perform the transformations that yield the maximum benefits. In general, no static phase ordering scheme allows an on-demand register allocator to allocate the registers to the transformations that most benefit the code, which is unfortunate on a machine-level framework that can provide accurate cost and benefit information.

The remainder of this paper is devoted to describing the *new technology retargetable optimizer* (*ntro*), which overcomes the register allocation and phase interaction problems presented here. Section 2 presents an overview of *ntro* and lists some of its advantages. A more detailed explanation of the system is given in Section 3. Section 4 discusses the current state of the implementation and the areas that require further work. Section 5 concludes with a brief summary.

2 Overview

Our approach to overcoming the lack of control and feedback provided by common optimization frameworks is to integrate code generation, register allocation and code improvements. This is done in the *vpcc/ntro* compiler using the framework illustrated in Figure 3. The machine-independent *vpcc* front-end translates traditional **C** source code [KR78] into CVM, a high-level intermediate stack-based representation. A code expander performs macro expansion on each CVM opcode to produce target machine code in RTL from. Pseudo-registers are used to hold temporary expression values while all of the local user variables declared in the source code initially reside in memory.

This machine code passes to *ntro*, the back-end of the compiler, whose initial phase builds a control flow graph (CFG) for each source function and improves it by applying optimizations like dead code elimination, jump minimization and branch chain elimination. Loops are also located at this time. The subsequent instruction selection phase eliminates the naive code sequences caused by performing code generation via macro expansion of a low-level intermediate language without the benefit of case analysis. The third phase performs global dead variable elimination. Evaluation order determination, also performed at the machine level [Dav86], reduces the number of hardware register needed to assign the pseudo-registers used by the code expander.

Instead of transforming the code, the common subexpression elimination, local variable promotion, loop-invariant code motion and induction variable elimination phases describe potential transformations in a transformation directory. After all of the potential transformations are described, the transformation dispatcher:

- determines which of the transformations described in the directory would most improve code,
- attempts to find sufficient hardware registers to perform the transformation,
- applies the transformation,
- removes the transformation description from the directory and
- updates the descriptions of the remaining transformations as required.

The dispatcher continues to perform these steps repeatedly until:

- the transformation directory is empty,
- none of the transformations remaining in the directory would improve the code or
- sufficient registers cannot be found to perform any of the transformations in the directory.

The final optimization phase performs instruction scheduling and fills branch delay slots on target machines that benefit from these transformations. Finally, an appropriate function prologue and epilogue is inserted and the code is translated to assembly language. An assembler and linker handle the remainder of the compilation process.

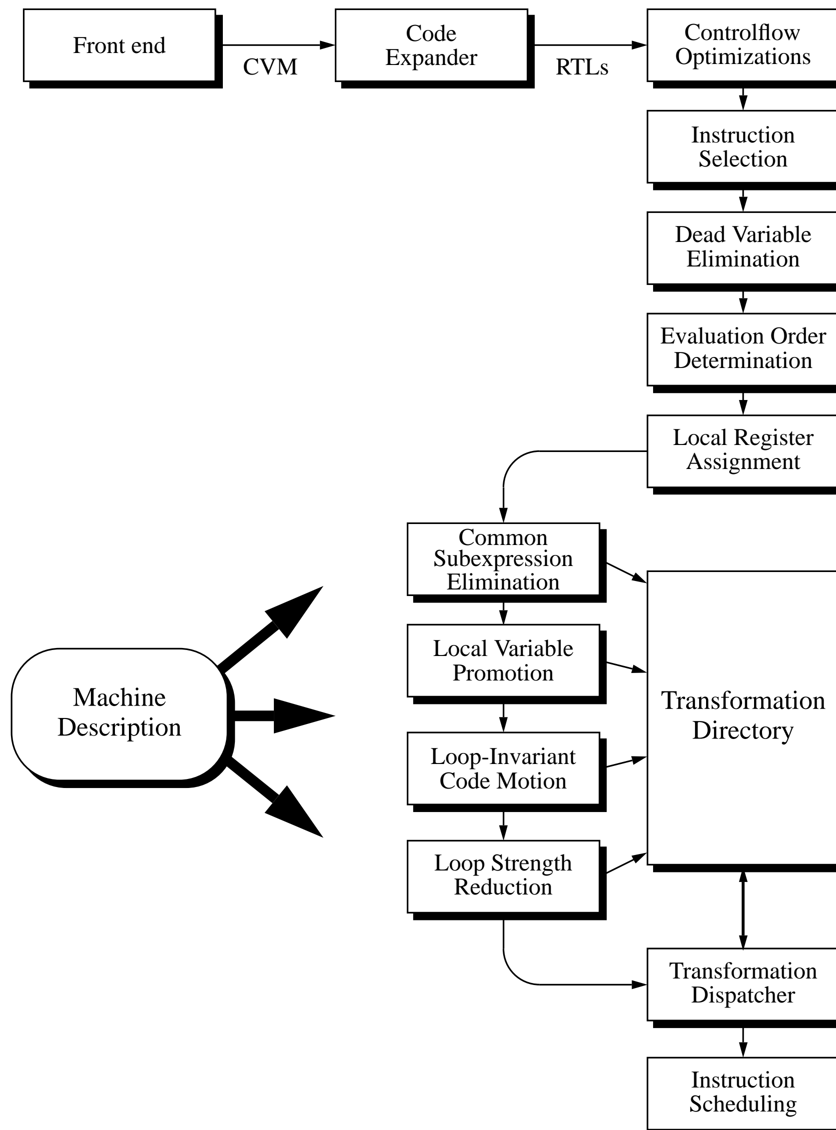

Figure 3: Structure of *ntro*

This framework is highly retargetable and generates high-quality code. A number of factors contribute to its retargetability. First, code generation is performed via simple macro expansion of 50 simple CVM opcodes. No machine-specific case analysis, which requires a great deal of time, effort and experience to implement, is used. The standardized RTL notation used to represent machine instructions can be manipulated by algorithms that are largely machine-independent. These algorithms, which make few assumptions about the target machine, are partitioned into machine-independent and machine-specific parts. Whenever possible, machine information is provided by constants and variable declarations instead of hand-written code. Tools have been developed to generate some of the machine-dependent code from a concise description of the target machine. This is especially true of code that handles the instruction set and the register file.

The transformation dispatcher enhances retargetability because it apportions registers well regardless of the size of the allocable register set on the target machine. The dispatcher also reduces phase ordering problems by using its knowledge of the target machine and the code being processed to determine the best ordering of not just phases, but

of individual transformations, to best suit the needs of each function. This completely eliminates the time and effort required to determine an appropriate phase ordering for each target machine.

Performing optimizations at the machine-level allows *ntro* to exploit unique architectural features. The *ntro* framework does not require the allocable register set to be partitioned between the code generator and the optimizer. Describing all of the transformations before applying them allows the dispatcher to invest the register resources on the transformations that are most likely to improve the code. This strategy is useful in a retargetable optimizer, because it allows a large set of optimizations to be performed even on a machine that has few registers without the danger of overcommitting the register set. Since the code is manipulated at the machine level, the costs and benefits of each transformation can be determined more accurately than a high-level representation would allow, and increases the ability of the dispatcher to perform the most beneficial transformations.

3 Implementation

Our implementation of *ntro* internally represents RTLs as trees, where each node may have zero, one or two children and contain information that is specific to its kind. A global define-use web connects related register and memory reference nodes to provide the define-use information needed to perform code improvements. Figure 4 shows the define-use web for a portion of the code in Figure 2(a). Each register and dereference node whose aliases are completely known is connected to the node containing previous set or use of the item and to the node containing the next set or use of the item. Web links replace *combiner links* in the instruction selection process [DF84] and are used to substitute the expression assigned to a register item in place of its first use. If the combination represents an actual instruction on the target machine, it replaces the instructions that formed it. Web links are also used to determine where an item is last referenced, a function that is essential to the instruction selection process.

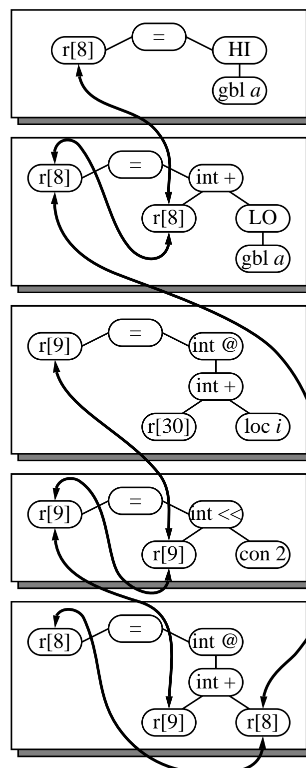

Figure 4: Define-use web links

To connect definitions and uses across basic blocks, ϕ functions, which join merging values, and μ functions, which represent diverging values, are placed at the entry and exit points of some basic blocks. ϕ functions are the confluence operators used in the static single assignment (SSA) form [CFR+91]. μ functions, which we developed specifically to allow the web to quickly find all of the possible uses of an assignment, are essentially ϕ functions for the inverted CFG, where the function's exit block becomes its entry block and all control flow paths are reversed. Unlike combiner links, the global web links can be used to combine instructions across basic blocks. An instruction pair is combined only if it contains an assignment that is the only possible source of a value and a use that is the only possible next use. These pairs are easily found because the RTL nodes representing the assignment and the use are directly connected without any intervening nodes, or functions.

The web is updated as transformations are applied to allow instruction selection to be re-invoked at any point in the optimization process. This allows instruction selection to be used to determine if a potential transformation will enable new instructions to be selected. Updating the web also makes it possible to eliminate the dead variables that are created as transformations are applied. The initial dead variable elimination pass uses the web to count the number of times that the value of each assignment is used and eliminates those that are never used. These use counts are then carefully updated as instructions are modified, inserted and removed so that they can automatically trigger the deletion of any expression orphaned by the code improvement process.

The values produced by each instruction and the relationship between these values are kept in a value graph. This graph is used to perform common subexpression elimination, code motion and loop strength reduction. Even when the exact value of an item is not known, the relationships between values can enable useful transformations. Value graph nodes are indexed for quick access and to ensure that each node represents a unique type and value combination. Every RTL node points to the value graph node that represents its value.

The transformation directory is a collection of transformation descriptions, each describing a specific transformation that can be applied to the code. Figure 5 illustrates the descriptions of the RISC machine transformations mentioned in the introductory phase ordering example. Although the directory would contain other transformations in addition to these two, space considerations prevent us from including them here. The complexity of the illustration warrants a bit of orientation to precede the explanation of the elements that make up the directory. Along the left side of the figure is the RTL tree representation of the code in Figure 2(a). The RTL items are enclosed by squares representing the basic blocks that make up the CFG. The transformation description items are the tall, narrow boxes to the right of the code. Within these boxes are the local transformation items, represented by the squares with the rounded corners and the instruction-level items, which are the smaller rectangles inside the local transformation items. The RTL trees at the bottom-right corner are RTL expression and value items.

A transformation description contains a list of local transformation items that indicate the area over which the registers required to perform the transformation would be live. The region covered by each local item is represented in the illustration by the pair of arrows originating from the local item and pointing to the basic blocks. Local items serve two purposes in the dispatch process. First, they tell the dispatcher which parts of the CGF must be examined to determine which hardware registers are available to perform the transformation. Second, each local item is connected to all of the other local items that it shares the region with. These connections form an interference graph that allows the dispatcher to determine which transformations cannot share the same registers. The dispatcher uses this informa-

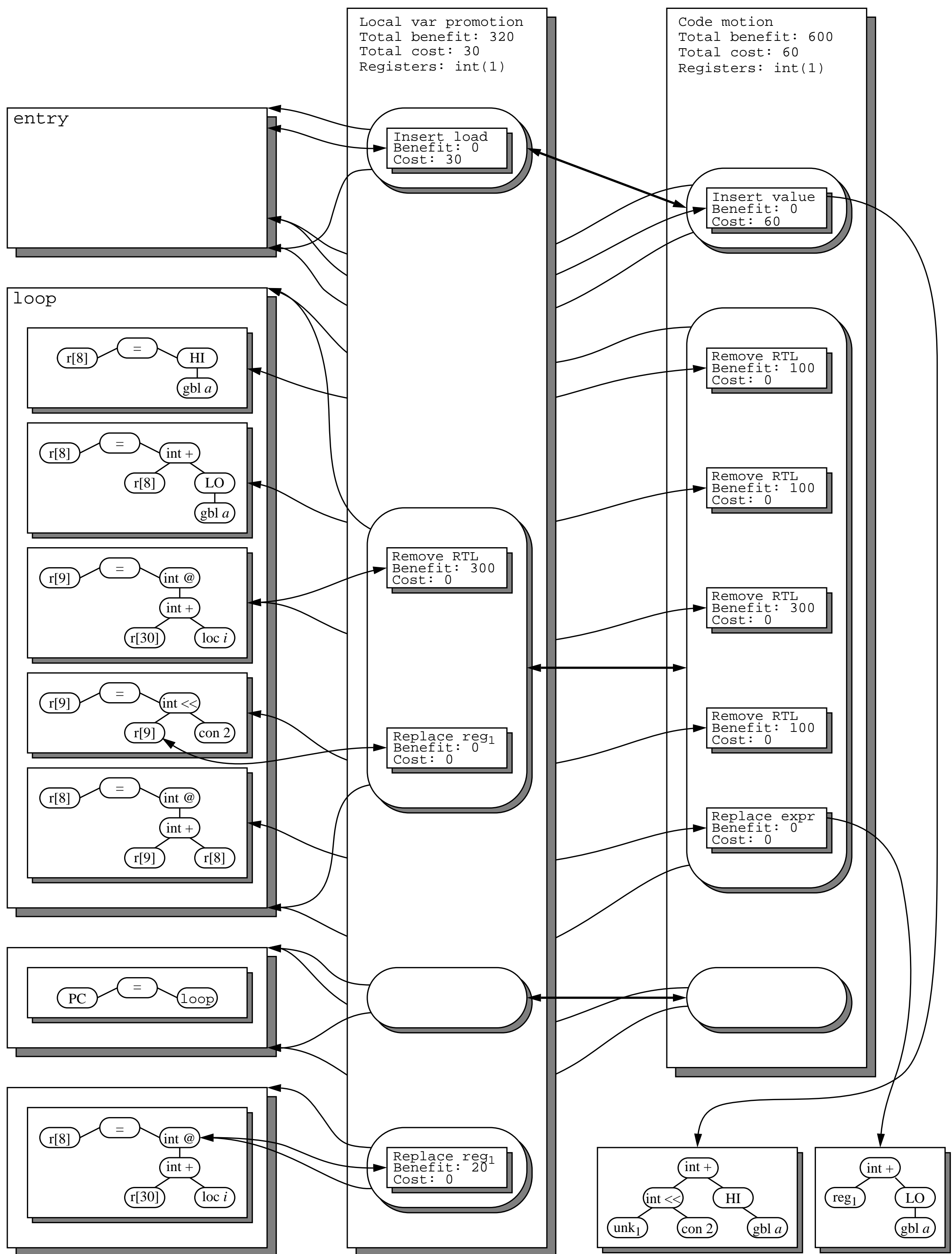

Figure 5: The transformation directory

tion to find groups of transformations that can share register and are together more beneficial, than any single transformation is.

Associated with some of the local items are instruction-level items that indicate which RTL nodes are affected by the transformation, describe the specific change that would be made to those nodes and contain the estimated cost and benefit values associated with that change. Cost and benefit values are obtained from the machine description information and multiplied by the loop nesting level the of the code because loop code is usually executed more frequently than straight-line code is. Each RTL node contains a list of all the instruction-level transformation items linked to it, so that if any one of the transformations that can be applied to it is performed, all of the other transformations that also reference it can be modified to reflect the impact of the transformation. For example, both of the transformations shown in Figure 5 affect the instruction that references variable *i* inside the loop. If the code motion transformation is applied, the entire RTL, along with the reference to the variable reference would be removed. In this situation, it would be necessary to locate the description of the local variable promotion transformation and update it. The update would entail removing the appropriate instruction-level transformation item from the local variable promotion description and updating the cost and benefit values to indicate that it is no longer as beneficial as it was prior to the code motion transformation.

The sum of the cost and benefit values of the instruction-level items that make up a transformation reflect the global costs and benefits of performing the transformation. The dispatcher uses these sums to decide which transformation to apply next. As the illustration shows, the code motion transformation has a total cost of 60 and a benefit of 600, which means that it will be selected by the dispatcher over the local variable promotion transformation with a benefit of 320 and a cost of 30. For the corresponding example on the CISC machine, these values will favor the local variable promotion transformation slightly over the code motion one, so that it will be selected, instead.

Each description in the directory indicates the number and type of registers needed to perform the transformation. The transformation dispatcher uses this information to attempt to allocate suitable registers just prior to attempting the transformation. This entails searching through the regions indicated by the local items to determine which hardware registers are not live within them. If sufficient hardware registers are unavailable, the transformation is postponed until another transformation releases enough registers to allow it to proceed.

What about transformations that improve the code even if they introduced a few spills? These transformations are still possible under *ntro*'s framework. The code improvement phases can produce transformations that are variations of each other with different register requirements, so that if the most beneficial variant is postponed, the less beneficial variant can be applied instead. For example, consider a pair of related transformations: one promotes a local variable to a register over the entire life of the function and the other over a single loop. The former is desirable because it does not require the variable to be exchanged between its home location in memory and a register at the boundaries of the loop. Its disadvantage is that it ties up a register over a greater area than the second option does in return for a marginal benefit. If the demand for registers is low, the transformation dispatcher will successfully obtain the register required by the former transformation and the latter transformation will be automatically deleted from the directory by the dispatcher as a side effect of applying the first transformation. When there are few registers or the demand for them is high, the former transformation is postponed because no register is available for the entire life of the function, but the latter transformation might succeed, because it spans a smaller region.

Some transformations initially consume registers, but trigger the elimination of a dead variable so that their net register consumption is zero or negative. On-demand register allocators like the one in *vpo* prevent these transformations from happening unless there are enough free registers available to "prime" the transformation process. The *ntro* framework will find registers for these transformations without the additional free registers because it performs the transformation using pseudo-registers before determining which registers can be used to perform the transformation. The pseudo-registers are immediately replaced with the allocated hardware registers if the allocation process is successful. Otherwise, the trial transformation is completely retracted.

Adding new optimizations to the *ntro* framework is not as difficult as its integrated nature might suggest. The transformation directory structure provides most of the transformation primitives (i.e. replace an expression with a register, remove an RTL, add an expression that computes a value, etc.) needed to accommodate most optimizations, and new primitives could be added as required. The most important requirement is that the optimization phase be able to detect all of the potential transformations by examining the code before any of the other global optimizations have been applied.

4 Results

We have implemented of most of the framework shown in Figure 3, with the exception of the common subexpression elimination and loop strength reduction phases. The framework has been ported to the SUN SPARC and the MIPS R3000 processors. Work on a Motorola 68030 version will begin soon.

Although in its current state, *ntro* does not yet perform enough global optimizations to make it competitive against many of the current production compilers, we have already seen some encouraging results. One such result is *ntro*'s ability to perform well even when the register demand is high. We simulated this condition by artificially reducing the number of allocable registers available to the compiler and comparing *ntro*'s performance against that of *vpo*. Our preliminary results suggest that *ntro*'s performance with only four allocable registers exceeds that of *vpo* with six or more. *We will be able to supply concrete numbers in the final paper.*

In its current state, the average *vpcc*/*ntro* compiler comprises 49,500 lines of **C** code. Of these, the machine-independent front-end takes up 8,869 lines. The code expander contains 4,170 lines of machine-independent code and an average of 2,400 lines of machine-specific code. The optimizing back-end consists of 30,498 lines of machine-independent code and an average of about 3,500 lines of machine-specific code and description text.

Retargeting the system entails modifying the machine-specific portions of the code expander and the back-end and supplying a simple description of the commands that a simple driver program must issue in order to execute all of the different components (i.e. pre-processor, front-end, back-end, assembler, linker) to complete the compilation process. The machine-specific portion of the code expander consists of 60 fairly simple (5 to 50 lines of **C**) code expansion routines. On machines where arguments are passed in registers, one of these routines often grows to about 200 lines of code that enumerates all of the possible cases prescribed by the calling convention. The machine-specific part of the back-end consists of a machine-description grammar, which averages about 350 lines of text, a simple 20 to 40 line register description, and about 3,000 lines of **C** declarations and code that are used to perform check the semantics of the machine instructions, translate RTLs to appropriate assembly code and add prologue and epilogue code for each function. The retargeting process can be done in two or three weeks by someone who is familiar with the system.

The speed of the prototype system is tolerable (hundreds of lines per second) on a SPARC 2 when compiling small to medium-size functions (200 lines or less), but the algorithms that we are using to update and maintain the web and the value graph do not scale as gracefully to large functions as we would like. The amount of memory required to compile a large function (2500 lines) rarely exceeds 16 megabytes, which is well within the limitations of small workstations. We expect to bring the compilation time of large functions more in line with that of average optimizing compilers as the implementation matures. This was certainly true of the *vpo* system.

In the future, we hope to be able to add a more comprehensive set of optimizations to *ntro*. We would also like to be able to integrate instruction scheduling into the framework so that its effects factor into the cost and benefit values of each transformation in the same way that instruction selection and dead variable elimination effects do.

5 Conclusion

We have presented a framework that exploits the accurate cost and benefit estimates available to a machine-level optimizer. This framework is highly-retargetable because it uses an internal representation that allows most of the optimization algorithms to be machine-independent. Also, its ability to dynamically tailor the order in which transformations are applied according to the requirements of each individual function eliminates the need to determine an ordering for each target architecture. The framework is capable of generating high-quality code even when the demand for register is high, which is advantageous when producing code for a machine that has few allocable registers and when the source code provides many opportunities to apply optimizations. In addition, the framework's ability to consider the cost and benefits of a set of potential code improvements and selecting those that show the most promise is similar to the approach taken by human assembly language programmers to improve assembly code.

6 References

[BD88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the ACM SIGPLAN '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. W. Wegman and F. K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems*, 13, 4, October 1991, 451-490.

[CH84] F. C. Chow and John Hennessy, Register Allocation by Priority-based Coloring, *Proceedings of the ACM SIGPLAN '84 Symposium on Programming Language Design and Implementation*, Montreal, Canada, June 1984, 222-232.

[DF80] J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Transactions on Programming Languages and Systems*, 2, 2, April 1980, 191-202.

[DF84] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *ACM Transactions on Programming Languages and Systems*, 6, 4, October 1984, 505-526.

[Dav86] J. W. Davidson, A Retargetable Instruction Reorganizer, *Proceedings of the ACM SIGPLAN '86 Symposium on Programming Language Design and Implementation*, Palo Alto, CA, June 1986, 234-241.

[KR78] B. W. Kernighan and D. M. Ritchie, The **C** Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Sta89] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Cambridge, MA, 1989.

[TSK+83] A. S. Tanenbaum, H. van Staveren, E. G. Keizer and J. W. Stevenson, A Practical Tool Kit for Making Portable Compilers, *Communications of the ACM*, 26, 9, September 1983, 654-660.