# VISTA: A System for Interactive Code Improvement†

WANKANG ZHAO[1], BAOSHENG CAI[2], DAVID WHALLEY[1], MARK W. BAILEY[1,3], ROBERT VAN ENGELEN[1], XIN YUAN[1], JASON D. HISER[4], JACK W. DAVIDSON[4], KYLE GALLIVAN[1], AND DOUGLAS L. JONES[5]

[1] Computer Science Department, Florida State University, Tallahassee, FL 32306-4530
E-mail: {wankzhao, whalley, engelen, xyuan, gallivan}@cs.fsu.edu

[2] Oracle Corporation, 4OP 955, 400 Oracle Parkway, Redwood City, CA 94065
E-mail: baosheng.cai@oracle.com

[3] Computer Science Department, Hamilton College, Clinton, NY 13323
E-mail: mbailey@hamilton.edu

[4] Computer Science Department, University of Virginia, Charlottesville, VA 22903
E-mail: {hiser, jwd}@virginia.edu

[5] Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801
E-mail: dl-jones@uiuc.edu

## ABSTRACT

Software designers face many challenges when developing applications for embedded systems. A major challenge is meeting the conflicting constraints of speed, code density, and power consumption. Traditional optimizing compiler technology is usually of little help in addressing this challenge. To meet speed, power, and size constraints, application developers typically resort to hand-coded assembly language. The results are software systems that are not portable, less robust, and more costly to develop and maintain. This paper describes a new code improvement paradigm implemented in a system called *vista* that can help achieve the cost/performance trade-offs that embedded applications demand. Unlike traditional compilation systems where the smallest unit of compilation is typically a function and the programmer has no control over the code improvement process other than what types of code improvements to perform, the *vista* system opens the code improvement process and gives the application programmer, when necessary, the ability to finely control it. In particular, *vista* allows the application developer to (1) direct the order and scope in which the code improvement phases are applied, (2) manually specify code transformations, (3) undo previously applied transformations, and (4) view the low-level program representation graphically. *vista* can be used by embedded systems developers to produce applications, by compiler writers to prototype and debug new low-level code transformations, and by instructors to illustrate code transformations (*e.g.*, in a compilers course).

† This work is supported in part by National Science Foundation grants EIA-9806525, CCR-9904943, and EIA-0072043.

## Category and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*graphical user interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*interactive environments*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*, *interactive systems*

## General Terms

Performance, Measurement

## Keywords

User-directed code improvement

## 1 INTRODUCTION

The problem of automatically generating acceptable code for embedded microprocessors is much more complicated than for general-purpose processors. First, embedded applications are optimized for a number of conflicting constraints. In addition to speed, other common constraints are code size and power consumption. For many embedded applications, code density and power consumption are often more critical than speed. In fact, in many applications, the conflicting constraints of speed, code density, and power consumption are managed by the software designer writing and tuning assembly code. Unfortunately, the resulting software is less portable, less robust, and more costly to develop and maintain.

Automatic compilation for embedded microprocessors is further complicated because embedded microprocessors often have specialized architectural features that make code improvement and code generation difficult [20, 19]. While some progress has been made in developing compilers and embedded software development tools, many embedded applications still contain substantial amounts of assembly language because current compiler technology cannot produce code that meets the cost and performance goals for the application domain.

In this paper we describe a new code improvement paradigm that we believe can achieve the cost/performance trade-offs (*i.e.*, size, power, speed, cost, *etc.*) demanded for embedded applications. A traditional compilation framework has a fixed order in which the code improvement phases are executed and there is no control over individual transformations, except for compilation flags to turn code improvement phases on or off. In contrast, our compilation framework, called *vista* (*vpo* Interactive System for Tuning Applications) gives the application programmer the ability to finely control the code-improvement process.

We had the following goals when developing the *vista* compilation framework. First, the user should be able to direct the order of the code improvement phases that are to be performed. The order of the code improvement phases in a typical compiler is fixed, which is unlikely to be the best order for all applications. Second, user-specified transformations should be possible. For instance, the user may provide a code sequence that *vista* inserts and integrates into the program. We are not aware of any compiler that allows a programmer such direct and fine control over the code improvement process. Third, the user should be able to undo code transformations previously applied since a user may wish to experiment with other, alternative phase orderings or types of transformations. In contrast, the effects of a code transformation cannot be reversed once it is applied in a typical compiler. Finally, the low-level program representation should appear in an easily readable display. The use of dynamically allocated structures by optimizing compilers and the inadequate debugging facilities of conventional source-level symbolic debuggers makes it difficult for a typical user to visualize the low-level program representation of an application during the compilation process. To assist the programmer when interacting with the optimization engine, *vista* should provide the ability for the programmer to view the current program representation, relevant compilation state information (*e.g.*, live registers, dominator information, loops, dead registers, *etc.*) and performance metrics.

Figure 1 illustrates the flow of information in *vista*. The programmer initially specifies a source file to be compiled. The programmer retains control over the code-improvement process by specifying requests to the compiler, which includes the order of the code improvement phases and actual transformations to be performed. The compiler responds to optimization requests by performing the specified actions and sending the program representation changes back to the viewer. Likewise, the compiler responds to queries by the viewer about the program representation state. If at any point the user chooses to terminate the session, *vista* saves the current sequence of applied program transformations in a file to enable future updates. The programmer may also wish to save multiple optimized versions to contrast their performance. In addition, the programmer may collect preliminary results about the performance of the generated code by requesting that the assembly code be instrumented with additional instructions that collect a variety of measurements during the program's execution.

Beyond *vista*'s primary purpose of supporting development of embedded systems applications, it has several other uses. First, *vista* can assist a compiler writer in prototyping and debugging new code transformations by manually specifying them. Second, it can help compiler writers to understand how different code improvements interact. Finally, instructors teaching compilation techniques can use the system to illustrate code transformations.

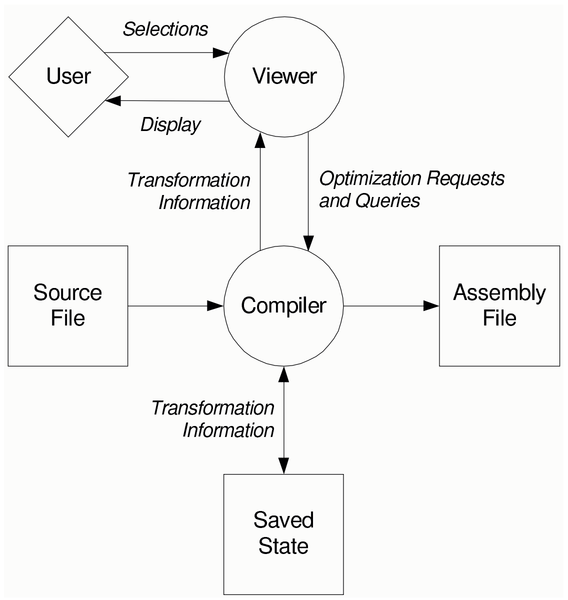

Selections
User
Viewer
Display
Transformation Information
Optimization Requests and Queries
Source File
Compiler
Assembly File
Transformation Information
Saved State

**Figure 1: Interactive code improvement process.**

The remainder of this paper is structured as follows. First, we review related work regarding alternative compilation paradigms and compiler user interfaces. Second, we describe the underlying structure of *vista*'s optimization engine. Third, we present the functionality of the types of compilation requests that a programmer may make and show how this functionality is achieved in the viewer. Fourth, we discuss implementation issues that were involved in developing an interactive code improvement system. Finally, we give the conclusions of the paper.

## 2 RELATED WORK

There exist systems that are used for simple visualization of the compilation process. The UW Illustrated Compiler [1], also known as *icomp*, is used in undergraduate compiler classes to illustrate the compilation process. The *xvpodb* system [6, 7] is used to illustrate low-level code transformations in the *vpo* compiler system [4]. *xvpodb* is also used when teaching compiler classes and to help ease the process of retargeting the compiler to a new machine or diagnosing problems when developing new code transformations.

There are also several systems that provide some visualization support for the high-level parallelization of programs. These systems include the *pat* toolkit [2], the *parafrase-2* environment [24], the *e/sp* system [8], a visualization system developed at the University of Pittsburgh [14], a visualization tool for the Zephyr system [18], and SUIF explorer [21]. All of these systems provide support for a programmer by illustrating the possible dependencies that may prevent parallelizing transformations from occurring. A user can inspect these dependencies and assist the compilation system by verifying whether a dependency is valid or can be removed. In contrast, *vista* supports low-level transformations and user-specified changes, which are needed for tuning embedded applications.

Because of the difficulty of producing code for embedded processors that meets the conflicting constraints of space, speed, and power consumption, there is a wide body of research that has advanced the state of the art of embedded systems compilation [20, 22, 15, 13, 25, 16]. There is also some work on experimenting with code improvement phase ordering and other techniques to produce better code. Coagulating code generators use run-time profiles to perform code improvements on the most frequent sections of the code before the less frequently executed sections [23]. Genetic algorithms have been used to experiment with different orders of applying code improvement phases in an attempt to reduce code size [9, 10]. Iterative compilation techniques have been used to determine good phase orderings for specific programs [11] and values for optimization parameters such as loop unroll factors and blocking sizes [17]. In contrast, *vista* allows a user to interactively specify the order and scope in which code improvement phases are applied.

## 3 *VISTA*'S OPTIMIZATION ENGINE

*vista*'s optimization engine is based on *vpo*, the Very Portable Optimizer [3, 5]. *vpo* has several properties that make it an ideal starting point for realizing the *vista* compilation framework. First, *vpo* performs all code improvements on a single intermediate representation called RTLs (register transfer lists). RTL is a machine- and language-independent representation of machine-specific instructions. The comprehensive use of RTL in *vpo* has several important consequences. Because there is a single representation, *vpo* offers the possibility of applying analyses and code transformations repeatedly and in an arbitrary order. In addition, the use of RTL allows *vpo* to be largely machine-independent, yet efficiently handle machine-specific aspects such as register allocation, instruction scheduling, memory latencies, multiple condition code registers, *etc.* *vpo*, in effect, improves object code. Machine-specific code improvement is important for embedded systems because it is a viable approach for realizing compilers that produce code that effectively balances target-specific constraints such as code density, power consumption, and execution speed.

A second important property of *vpo* is that it is easily retargeted to a new machine. Retargetability is key for embedded microprocessors where chip manufacturers provide many different variants of the same base architecture and some chips have application-specific designs.

A third property of *vpo* is that it is easily extended to handle new architectural features. Extensibility is also important for embedded chips where cost, performance, and power consumption considerations often mandate development of specialized features centered around a core architecture.

A fourth and final property of *vpo* is that its analysis phases (*e.g.*, dataflow analysis, control flow analysis, *etc.*) are designed so that information is easily extracted and updated. This property makes writing new code improvement phases easier and it allows the information collected by the analyzers to be obtained for display.

## 4 FUNCTIONALITY OF *VISTA*

In this section we describe the functionality of *vista* from the user's viewpoint. This functionality includes viewing the low-level representation, controlling when and where code improvement phases are applied, specifying code transformations, measuring performance, and undoing previously applied transformations.

### 4.1 Viewing the Low-Level Representation

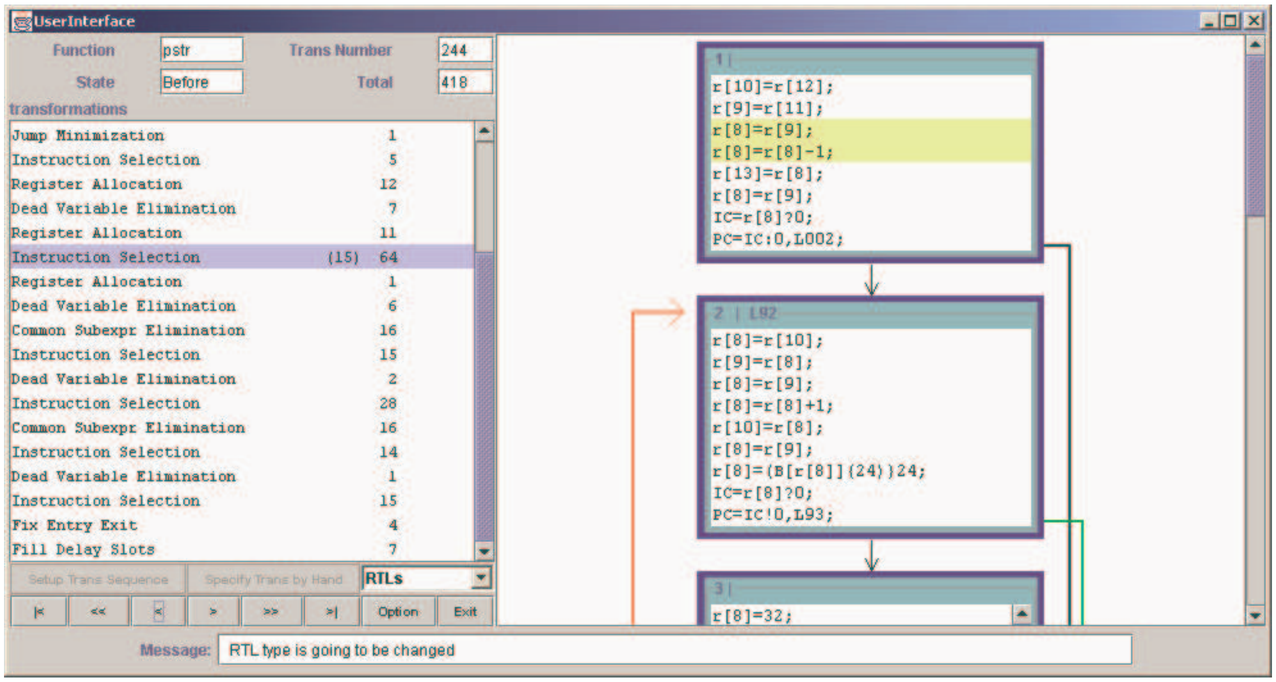
Figure 2 shows a snapshot of the *vista* viewer that supports interactive code improvement. The program representation appears in the right window of the viewer and is shown as basic blocks in a control flow graph. Within the basic blocks are machine instructions. The programmer may view these instructions as either RTLs or assembly code. Displaying the representation in RTLs may be preferred by compiler writers, while assembly may be preferred by embedded systems application developers who are familiar with the assembly language for a particular machine. In addition, *vista* provides options to display additional information about the program that a programmer may find useful.

The left window varies depending on the viewer's mode. Figure 2 shows the default display mode. The top left of the viewer screen shows the name of the current function and the number of the current transformation. A transformation consists of a sequence of changes that preserve the semantics of the program. The viewer can display a transformation in either the *before* or *after* state. In the *before* state, the transformation has not yet been applied. However, the instructions that are to be modified or deleted are highlighted. In the *after* state, the transformation has been applied. At this point, the instructions that have been modified or inserted are highlighted. This highlighting allows a user to quickly and easily understand the effects of each transformation. *Before* and *after* states are used in other graphical compilation viewers [6, 7].

The bottom left window contains the viewer control panel. The '>', '>>', and '>|' buttons allow a user to advance through the transformations that were performed. The '|<', '<<', and '<' buttons allow the user to back through the transformations and are described in further detail in Section 4.5. The '>' button allows a user to display the next transformation. A user can display an entire transformation (*before* and *after* states) with two clicks of this button. The '>>' button allows a user to advance to the next phase. A phase is a sequence of transformations applying the same type of transformation. The '>|' button allows the user to advance beyond all transformations and phases.

Shown in the left window is the list of code improvement phases that the compiler has performed, which includes phases that have been applied in the viewer and the phases yet to be applied by the viewer. For instance, the state represented in Figure 2 is in the *before* state of transformation 15 of the sixth phase. We have found that displaying the list of code improvement phases in this manner helps to give a user some context to the current state of the program.

When viewing a program, the user may also change the way the program is displayed and may query the compiler for additional information. When control flow is of more interest than the specific machine instructions, the user can choose to display only the basic block structure without the machine instructions. This makes it possible to display more basic blocks on the screen and gives the user a wider view of the program as shown in Figure 3. The user may also query the compiler for the list of loops in a function, a list of registers that are live at a particular point, basic block information such as dominators, successors, predecessors, and RTL information such as registers that are dead.



Function: pstr
State: Before
Trans Number: 244
Total: 418

```
Jump Minimization                1
Instruction Selection            5
Register Allocation             12
Dead Variable Elimination        7
Register Allocation             11
Instruction Selection      (15) 64
Register Allocation              1
Dead Variable Elimination        6
Common Subexpr Elimination      16
Instruction Selection           15
Dead Variable Elimination        2
Instruction Selection           28
Common Subexpr Elimination      16
Instruction Selection           14
Dead Variable Elimination        1
Instruction Selection           15
Fix Entry Exit                   4
Fill Delay Slots                 7
```

```
r[10]=r[12];
r[9]=r[11];
r[8]=r[9];
r[8]=r[8]-1;
r[13]=r[8];
r[8]=r[9];
IC=r[8]?0;
PC=IC:0,L002;
```

```
r[8]=r[10];
r[9]=r[8];
r[8]=r[9];
r[8]=r[8]+1;
r[10]=r[8];
r[8]=r[9];
r[8]=(B[r[8]]{24)}24;
IC=r[8]?0;
PC=IC!0,L93;
```

```
r[8]=32;
```

Message: RTL type is going to be changed

**Figure 2: Showing the history of code improvement performed by the compiler on the SPARC.**

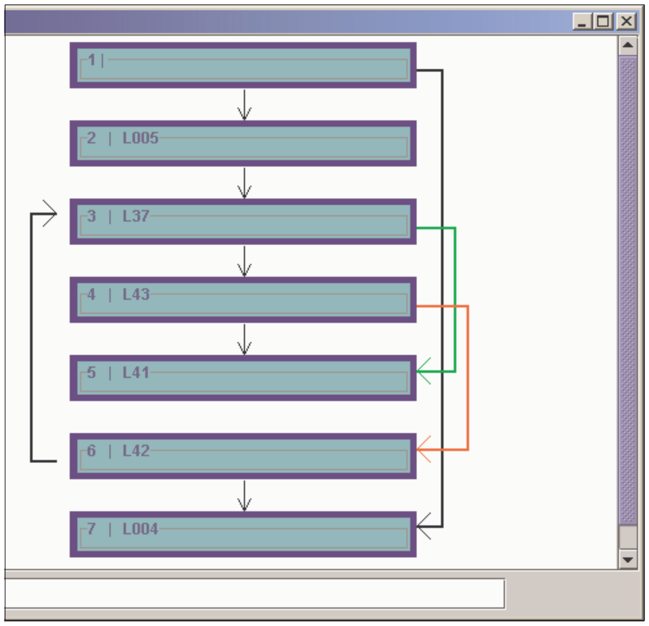


**Figure 3: Displaying only program control flow.**

## 4.2 Directing the Order of Phases

Generally, a programmer has little control over the order in which a typical compiler applies code improvement phases. Usually the programmer may only turn a compiler code improvement phase on or off for the entire compilation of a file. For some functions, one phase ordering may produce the most suitable code, while a different phase ordering may be best for other functions. Consider Figure 4, which shows effect that cross jumping has on 75 ARM object files. The effect is illustrated as a ratio of code size when cross jumping is performed after common subexpression elimination versus before common subexpression elimination. While most of the programs showed a greater reduction in code size after common subexpression elimination, there were some programs in which a smaller executable was achieved when cross jumping was performed before common subexpression elimination. *vista* provides the programmer with the flexibility to specify what code improvements to apply to a program region and the order in which to apply them. A knowledgeable embedded systems application developer can use this capability for critical program regions to specify that the most beneficial transformations are applied in the most advantageous order.

Figure 5 shows the user selecting code improvement phases. The user can make selections from all of the different required and optional phases that the back end of the compiler applies. As each phase is selected, *vista* adds it to a numbered list of code improvement phases. In addition, the user may specify phase order *control*. For instance, the example in Figure 5 shows that the user specified that as long as changes to the representation are detected, the compiler should repeatedly perform *register allocation*, *common subexpression elimination* and *instruction selection*. Thus, we are in essence providing the programmer with the ability to program the compiler in a code improvement ordering phase language. Once the user confirms the selection of the sequence of phases, this sequence is sent to the compiler, which performs the phases in the specified order and sends a series of messages back to the viewer describing the resulting program representation changes.

As shown in the upper window, the user is prevented from selecting some of the phases at particular points in the compilation. This is due to compiler restrictions on the order in which it can perform phases. Although we have tried to keep these restrictions to a minimum, some restrictions are unavoidable. For instance, the compiler

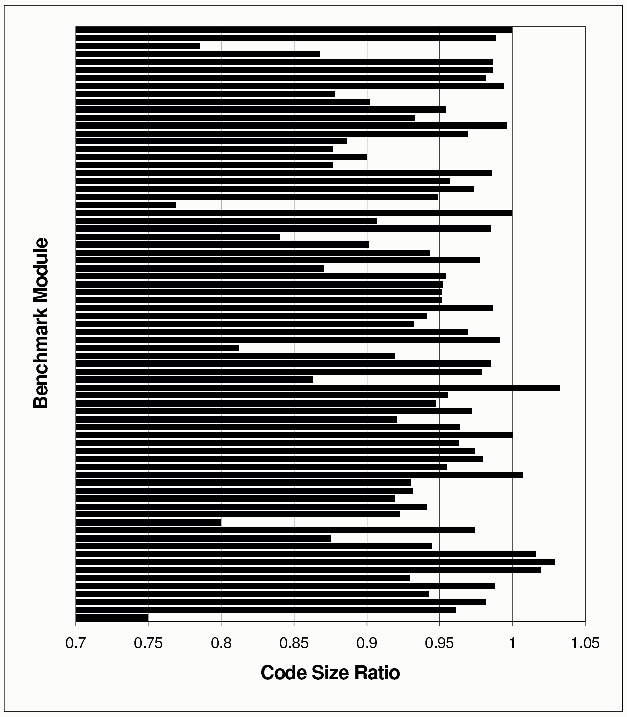
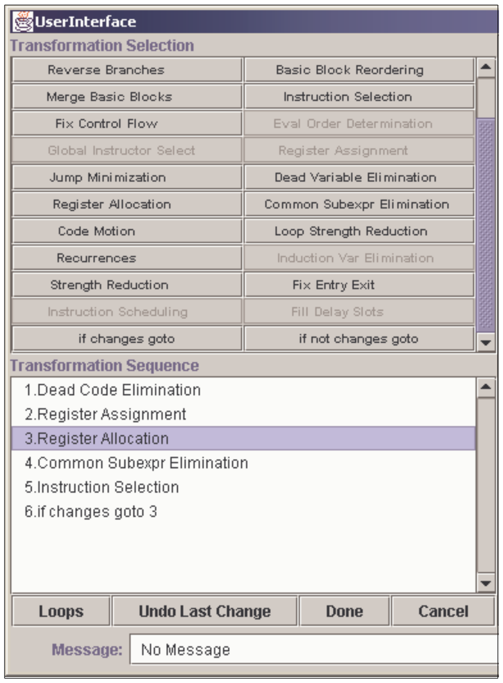


**Figure 4: Ratio of applying cross-jumping after CSE to applying cross-jumping before CSE.**



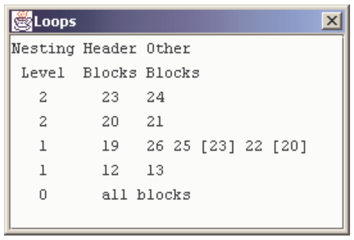
1.Dead Code Elimination
2.Register Assignment
3.Register Allocation
4.Common Subexpr Elimination
5.Instruction Selection
6.if changes goto 3

Message: No Message

**Figure 5: Selecting an iterative sequence of phases.**

does not allow the *register allocation* phase (allocating variables to hardware registers) to be selected until the *register assignment* phase (assigning pseudo registers to hardware registers) has been completed*. Likewise, the user may only request that the compiler perform register assignment once.

In addition to specifying the order of the code improvement phases, a user can also restrict the scope in which a phase is applied to a region of code. This feature allows a user to use resources, such as registers, based on what she considers to be the critical portions of the program. The user can restrict the region to a set of basic blocks by either clicking on blocks in the right window or clicking on loops in a loop report similar to the one shown in Figure 6. We believe that a set of blocks and loops are natural units for which users would wish to restrict the scope of phases. A few phases cannot have their scope restricted due to the method in which they were implemented in the compiler or how they interact with other phases (*e.g.*, *filling delay slots*). Note that by default the scope in which a phase is applied is unrestricted (*i.e.*, the entire function).



```
Nesting Header Other
 Level  Blocks Blocks
   2      23   24
   2      20   21
   1      19   26 25 [23] 22 [20]
   1      12   13
   0      all blocks
```

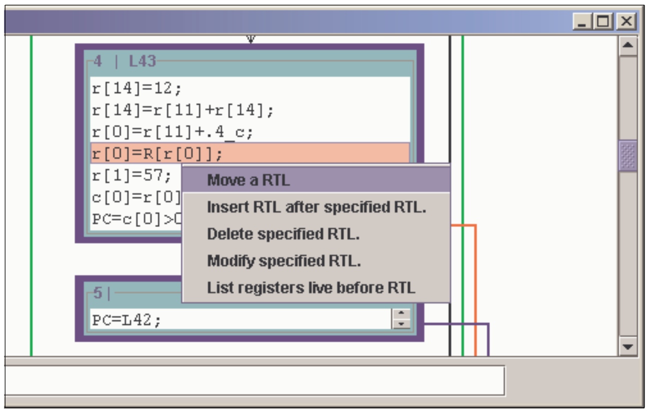
**Figure 6: Loop report indicating member basic blocks.**

## 4.3 User-Specified Code Transformations

Despite advances in code generation for embedded systems, knowledgeable assembly programmers can always improve code generated by current compiler technology. This is likely to be the case because the programmer has access to information the compiler does not. In addition, many embedded architectures have special features (*e.g.*, zero overhead loop buffers, modulo address arithmetic, *etc.*) not commonly available on general-purpose processors. Automatically exploiting these features is difficult due to the high rate at which these architectures are introduced and the time required for a highly optimizing compiler to be produced. Yet generating an application entirely in assembly code is not an attractive alternative due to the labor involved. It would be desirable to have a system that not only supports traditional compiler code improvement phases but also supports the ability to manually specify transformations.

Figure 7 shows how *vista* supports the application of user-specified code transformations. When the user points to an instruction, the viewer displays the list of possible user-specified changes for that instruction. As the user selects each change, the change is sent to the compiler, which checks it for validity. For instance, if an instruction is inserted, then the syntax is checked to make sure it is valid. Transformations to basic blocks (insert, delete, label) are also possible. A number of semantic checks are also necessary. For instance, if the target of a branch is modified, then the compiler checks to ensure that the target label in the branch is actually a label of a basic block. The compiler responds to each change by indicating if the change is valid and by sending the appropriate change messages to the viewer so it can update the presented program representation. The approach of immediately querying the

* Unlike textbook compilers, *vpo* performs register allocation after register assignment.


Move a RTL
Insert RTL after specified RTL.
Delete specified RTL.
Modify specified RTL.
List registers live before RTL

```
4 | L43
r[14]=12;
r[14]=r[11]+r[14];
r[0]=r[11]+.4_c;
r[0]=R[r[0]];
r[1]=57;
c[0]=r[0]
PC=c[0]>C

5 |
PC=L42;
```

**Figure 7: Manually specifying a transformation on the ARM.**

compiler when each change is specified ensures that the user does not create an invalid program representation.

The user also can query the compiler for information that may be helpful when specifying a transformation. For instance, a user may wish to know which registers are live at a particular point in the control flow. The query is sent to the compiler, the compiler obtains the requested information (calculating it only if necessary) and sends it back to viewer. Thus, the compiler can be used to help ensure that the changes associated with user-specified transformations are properly made and to guide the user in generating valid and more efficient code.

### *4.3.1 User-Specified Transformation Examples*

Common examples of programmer code improvements for embedded applications are ones that exploit knowledge that the programmer has that the compiler does not. One class of such transformations are value-dependent transformations. Consider the binary search function shown in Figure 8(a). In this example, which was extracted from an embedded application, a small global table (32 entries) of short integers is searched. The programmer has kept the table small by using shorts. The programmer has gone a step farther to conserve space by using bytes as the indices into the array. The RTLs the ARM compiler generates are shown in Figure 8(b).

Close examination of the generated RTLs yields several opportunities for improvement. The index variable *c* must be kept in the range of an unsigned byte. The compiler generates the necessary masks when *c* is used. However, since the programmer knows *c*'s range is 0–31, the masking instructions are not necessary, so they can be removed. This removes two instructions from the loop (line 9 and 19) and one instruction outside the loop (line 30). In addition, line 17, which adds the sign bit for signed division (for the division by 2), is unnecessary since the expression never yields a negative result. These are important transformations for the function because they reduce the number of instructions in the loop from 16 to 13 (an 18% reduction).

One further improvement can be made to the function. Since the function takes a short as a parameter, the procedure calling convention requires that the caller promote the parameter to a long. The callee then must zero-extend *data* to ensure that it falls in range of a short (lines 2 and 3). If the programmer knows that the function will always be called with values of *data* that are in range, then the zero-extension can be removed. This removes two additional instructions.

It may be possible that some of these transformations can be automated when global analysis is performed. However, even with aggressive optimization technology, there will always be situations where a programmer concerned with meeting a particular constraint may wish to make modifications to the code produced by the compiler.

```
unsigned char binary_search(short data) {
  unsigned char u,l,c;
  u=32,l=0,c=u/2;
  for(;;) {
    if(data<table[c]) {
        u=c;
        c=(l+u)/2;
      }
    else {
      l=c;
      c=(u+l)/2;
    }
    if((u-c)==1 || (c-l)==1) break;
  }
  return table[c]==data ? c : c-1;
}
```

**(a) Source code.**

```
binary_search:
   1  r[1]=r[0];
   2  r[12]=r[1]{16;
   3  r[1]=r[12]}16;
   4  r[12]=LA[L1];
   5  r[4]=LA[r[12]];
   6  r[2]=32;
   7  r[3]=0;
   8  r[0]=16;
 L4
   9  r[0]=r[0]&255;
  10 r[12]=r[4]+(r[0]*2);
  11 r[12]=(W[r[12]]{16)}16;
  12 c[0]=r[1]?r[12];
  13 r[2]=c[0]<0,r[0],r[2];
  14 r[12]=c[0]<0,r[3]+r[0],r[12];
  15 r[3]=c[0]`0,r[0],r[3];
  16 r[12]=c[0]`0,r[2]+r[0],r[12];
  17 r[12]=r[12]+(r[12]"31);
  18 r[12]=r[12]}1;
  19 r[0]=r[12]&255;
  20 r[12]=r[2]-r[0];
  21 c[0]=r[12]?1;
  22 r[12]=c[0]!0,r[0]-r[3],r[12];
  23 c[0]=c[0]!0,r[12]!1,c[0];
  24 PC=c[0]!0,L4;
  25 r[12]=r[4]+(r[0]*2);
  26 r[12]=(W[r[12]]{16)}16;
  27 c[0]=r[12]?r[1];
  28 r[12]=c[0]:0,r[0],r[12];
  29 r[12]=c[0]!0,r[0]-1,r[12];
  30 r[0]=r[12]&255;
  31 PC=RT;
```

**(b) RTLs corresponding to the source code.**

**Figure 8: Example of a user-specified transformation.**

```
rowArray = malloc( .... );
...
for( row = 1 ; row <= numRows ; row++ ) {
   rowArray[row].endx1   = -1;
   rowArray[row].startx2 = -1;
}
```

**(a) Source code.**

```
L3
  22    r[1]=R[r[5]];
  23    r[0]=-1;
  24    r[3]=24;
  25    r[2]=R[r[4]];
  26    r[1]=r[1]+(r[2]*r[3]);
  27    R[r[1]+8]=r[0];
  28    r[1]=R[r[5]];
  29    r[2]=R[r[4]];
  30    r[1]=r[1]+(r[2]*r[3]);
  31    R[r[1]+12]=r[0];
  32    r[1]=LA[r[6]+8];
  33    r[0]=R[r[4]];
  34    r[2]=r[0]+1;
  35    R[r[4]]=r[2];
  36    r[1]=R[r[1]];
  37    c[0]=r[2]?r[1];
  38    PC=c[0]'0,L3;
```

**(b) RTLs corresponding to the source code.**

**Figure 9: Example of a user-specified transformation.**

Figure 9 shows another example where a knowledgeable programmer could overcome shortcomings in the alias analysis of a compiler. Figure 9(a) shows a source code excerpt from the *twolf* benchmark. A data structure, *rowArray*, is dynamically allocated. The *row* variable is a global scalar. Figure 9(b) shows the corresponding ARM instructions that are produced for this code portion. Without aggressive interprocedural alias analysis, the compiler cannot determine that the assignment to the field *endx1*, the store instruction at line 27, does not overwrite the global scalar *row*. If a user could determine that this is safe, then the RTLs at lines 28, 29, and 30 could be eliminated and the value in `r[1]` could be reused. Embedded systems application developers may often be willing to perform these types transformations to improve the performance of an embedded application in a product where millions of units may be sold.

Providing user-specified transformations as *vista* does has an additional benefit. After the programmer has identified and performed a transformation, the optimization engine can be called upon to further improve the code. Such user-specified transformations may reveal additional opportunities for the optimization engine that were not available without programmer knowledge. In this way, the optimizer and programmer can, jointly, generate better code than either the programmer or the optimizer could have generated separately.

### *4.3.2 Validating User-Specified Transformations*

A user can easily introduce errors in a program when specifying transformations. Transformations can be classified as required or optional. An optional transformation consists of a sequence of changes, where the program representation before and after the sequence should be semantically equivalent and hopefully improved (faster, smaller, less power). We have developed a system that attempts to validate optional transformations. The user advances or backs up to the point that the transformation in question is displayed. The user has the option to send a validation request for the current transformation to the compiler. The compiler sends one of three responses: (1) the transformation was validated, (2) the transformation was not validated, or (3) the region associated with the transformation could not be validated. The third response is sometimes given since we currently do not validate transformations that span multiple loop levels. Besides validating user-specified transformations, we have also validated a number of conventional compiler transformations, which include *algebraic simplification of expressions*, *basic block reordering*, *branch chaining*, *common subexpression elimination*, *constant folding*, *constant propagation*, *unreachable code elimination*, *dead store elimination*, *evaluation order determination*, *filling delay slots*, *induction variable removal*, *instruction selection*, *jump minimization*, *register allocation*, *strength reduction*, and *useless jump elimination*. More details about validating transformations using this system can be found elsewhere [26].

### *4.3.3 Prototyping New Code Improvements*

The ability to specify low-level code transformations has another important, more general, application of user-specified transformations. Unlike high-level code transformations, it is difficult to prototype the effectiveness of low-level code transformations.

There are two factors that make prototyping low-level code transformations difficult. First, many low-level code improvements exploit architectural features that are not visible in a high-level representation. For example, machine-dependent code improvements, such as register allocation, do not have equivalent source code transformations. In such cases, the source code cannot be hand-modified to measure the effectiveness of the transformation. Second, low-level code transformations are often only fully effective when performed after other, specific transformations have been applied. For example, branch chaining may reveal additional opportunities for unreachable code elimination. For such cases, it may be possible to perform the transformation in source code, but it is not possible to prototype its effectiveness accurately at the source level since opportunities will be missed.

One prototyping strategy is to generate low-level code, write it to a file, manually perform the code improvement, read the low-level code back in, and perform any additional code improvements. This process can only work if the code improver accepts the same representation it generates. Although *vpo* uses a single low-level representation, the RTLs it accepts use pseudo registers while the RTLs it generates use hardware registers. Often, there are other phase order limitations that prevent the use of this strategy. By opening the code improvement process to user-specified changes, *vista* provides a general solution to prototyping and measuring the effectiveness of new low-level code transformations.

## 4.4 Visualizing Performance

An important requirement of interactive code improvement is the ability to measure and visualize the performance of the program being improved. It is only by examining the performance of the program that the user can understand the effectiveness of their user-specified code improvements and their choices in phase ordering. *vista* provides the mechanisms necessary to easily gather and view this information.

Before applying any code transformations, the user may wish to know where in the program they should focus their efforts. At any point during the code improvement process, the user may gather this information by directing *vista* to produce assembly that is instrumented with performance measuring code. This code, when executed, produces instruction-level performance measurements using the *ease* system [12]. The resulting performance measurements are read by the compiler and transmitted to the viewer. The viewer, in turn, annotates the display with the performance measurements.

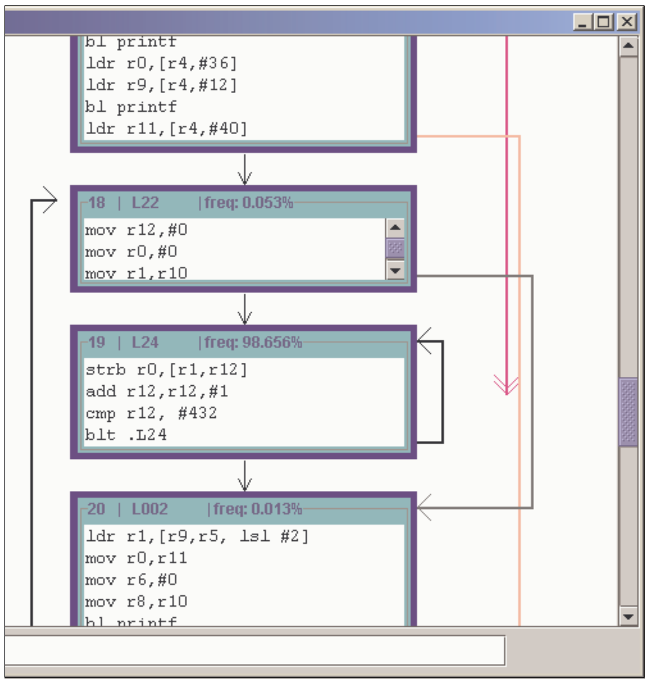
Figure 10 shows the control flow graph after gathering performance measurements. In the current implementation, execution counts are gathered for each basic block. The viewer receives these measurements for the entire function and computes the percentage of executed instructions for each basic block. This percentage is displayed for each basic block in the function so the user can quickly identify portions of the program that are executed frequently. This makes it easy for the user to identify the locations in the program that are most likely to benefit from additional tuning. The user can then improve the code—either by using an existing code improvement phase or by a user-specified code improvement—and take another set of measurements to glean the benefit of the improvement.

The current implementation allows the user to visualize performance, but it does not directly support the visualization of performance *improvement*. Clearly, it is desirable to see how individual transformations improve (or degrade) the performance of the program. We plan to address this need in two ways. First, the viewer can often derive the change in performance from applying a valid transformation. For example, if an instruction is removed from a basic block, the instruction counts can easily be updated. The same is true when an instruction is moved from one basic block to another. On the other hand, when control flow is modified or branch results are affected, changes in performance may be difficult to derive. In such cases, we plan to automatically capture new performance measurements (when directed to do so by the user) in order to accurately display the impact transformations have on performance.

## 4.5 Undoing Transformations

An important design issue for an interactive code improvement system is how to allow an embedded systems application developer to experiment with different orderings of phases and user-specified transformations in an attempt to improve the generated code. In order to support such experimentation, we provide the ability for the user to reverse previously made decisions regarding phases and transformations that have been specified.

This reversing of transformations is accomplished using the ('|<', '<<', '<') buttons. These buttons allow a user to view the transformations that were previously applied. The '<' button allows a user to display the previous transformation. The '<<' button allows a user to back up to the previous code improvement phase. Likewise, the '|<' button allows the user to view the state of the program before any transformations have been applied. The ability to back up and view previously applied transformations is very useful for



```
bl printf
ldr r0,[r4,#36]
ldr r9,[r4,#12]
bl printf
ldr r11,[r4,#40]
```

```
18 | L22 |freq: 0.053%
mov r12,#0
mov r0,#0
mov r1,r10
```

```
19 | L24 |freq: 98.656%
strb r0,[r1,r12]
add r12,r12,#1
cmp r12, #432
blt .L24
```

```
20 | L002 |freq: 0.013%
ldr r1,[r9,r5, lsl #2]
mov r0,r11
mov r6,#0
mov r8,r10
bl printf
```

**Figure 10: Visualizing performance on the ARM.**

understanding how code was generated or to grasp the effects of individual transformations.

If the user invokes a code improvement phase or user-specified transformation while viewing a prior state of the program, then the subsequent transformations must be removed before the new transformation can be performed. The user must confirm this action before the compiler is directed to make this adjustment. Thus, the user has the ability to permanently undo previously applied phases and transformations.

The ability to undo transformations is also useful in a batch code improvement environment. A traditional compiler can use this feature to exhaustively attempt a variety of code improvements and select the phase ordering that produces the most effective code. In addition, it is sometimes easier to perform a portion of a transformation before completely determining whether the transformation is legal or worthwhile. Being able to revoke changes to the program will facilitate the development of such transformations.

## 5 IMPLEMENTATION ISSUES

We implemented the viewer using Java to enhance its portability. We used Java 1.2, which includes the Java Swing toolkit that is used to create graphical user interfaces. The aspects of the interface that limit its speed are the displaying of information and the communication with the compiler. Thus, we have found that the performance of the interface was satisfyingly fast, despite having not been implemented in a traditionally compiled language.

We separated the compiler and the viewer into different processes for several reasons. First, the use of separate processes provides additional flexibility. For instance, the sequence of change messages sent from the compiler to the viewer can be saved and a simple simulator has been used instead of the compiler to facilitate demonstrations of the interface. Likewise, a set of user commands can be read from a file by a simple simulator that replaces the viewer, which can be used to support batch mode experimentation with different phase orderings. Second, we were concerned that the amount of memory used by the compiler and the viewer may be excessive for a single process. Separating the compiler and the viewer into separate processes allows users to access the interactive code improvement system on a different machine from which the compiler executes. The communication between the compiler and the viewer was accomplished using TCP/IP sockets.

*vpo* required numerous modifications to support interactive code improvement. First, the high-level function in *vpo* to perform the code improvement phases for a function had to be rewritten. *vpo* had a fixed order, depending upon the compilation flags selected, in which code improvement phases were attempted. Figure 11 shows the revised logic used for responding to user requests. A user can request to apply a sequence of code improvement phases, can manually specify a transformation, can undo previously applied transformations, or can query the compiler for some information about the current program representation.

After composing a sequence of code improvement phase commands, the viewer sends the sequence to *vpo* and the compiler interprets these commands until an *exit* command is encountered. The *branch* command allows a user to specify a transfer of control based on whether or not changes to the program were encountered. Before each code improvement phase, *vpo* performs the analysis needed for the phase that is not already marked as valid. After performing the code improvement phase, *vpo* marks which analysis could possibly be invalidated by the current phase. Identifying the analysis needed for each phase and analysis invalidated by each phase was accomplished in a table-driven fashion.

We also had to identify which phases were required during the compilation (*e.g.*, *fix entry/exit*), which code improvement phases could only be performed once (*e.g.*, *fill delay slots*), and the restrictions on the order in which code improvement phases could be applied. Fortunately, many ordering restrictions required by other compilers are not required in *vpo* since all code improvement phases are applied on a single program representation (RTLs).

Each change associated with a user-specified transformation has to be reflected in the program representation in the compiler. If an instruction was inserted or modified, the RTL or assembly instruction specified had to be converted to the encoded RTL representation used in the compiler. We developed two translators for this purpose. The first translator converts a human generated RTL into an encoded RTL. The second translator transforms an assembly instruction into an encoded RTL. After obtaining the encoded RTL, we used the machine description in the compiler to check if the instruction specified was legal for the machine. Before performing the change, the compiler checks if the change requested is valid. This check includes ensuring not only that the syntax of an instruction is valid, but also that its semantics are valid with regard to the rest of the program. If the user requested change does not cause the program to be in an invalid state, then the change is performed by the compiler.

A user can also request to undo one or more previously applied transformations. In order to accomplish such requests, both the compiler and the viewer keep a history of changes to the program. Figure 12 depicts a linked list of the history of changes. All

```
Read request from the user.
WHILE user selects additional requests to be performed DO
    IF user selected a sequence of phases THEN
        pc = 0.
        WHILE commands[pc].oper != EXIT DO
            IF commands[pc].oper == BRANCH THEN
                Adjust pc according to branch command
                CONTINUE;
            END IF
            Perform analysis needed for current phase.
            SWITCH (commands[pc].oper)
                CASE BRANCH_CHAINING:
                    remove branch chains.
                ...
            END SWITCH
            Mark analysis invalidated by current phase.
            pc += 1.
        END WHILE
    ELSE IF user selected a transformation change THEN
        Perform transformation change selected by the user.
    ELSE IF user selected to reverse transformations THEN
        Reverse transformations as specified by the user.
    ELSE IF user requested a query THEN
        Calculate query information and send to viewer.
    END IF
    Read request from the user.
END WHILE
```

**Figure 11: Algorithm for performing user requests.**

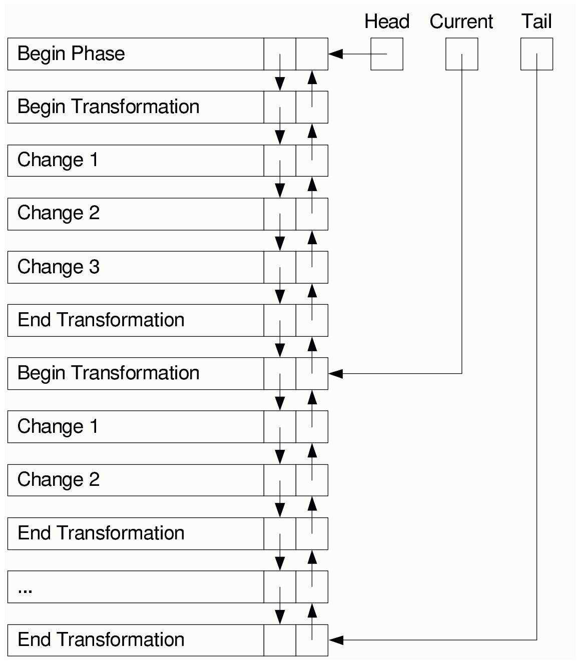


**Figure 12: Data structure used for undoing transformations.**

changes (additions and deletions) to the list occur at the tail. The viewer allows a user to apply or undo transformations using this list. Enough information regarding each change must be saved so its effect can be undone. For instance, if a change reflects a modification to an instruction, then the compiler and viewer must save the old version of the instruction before the modification so its effect can be reversed if requested by the user. The viewer updates its representation of the program to reflect applying or undoing transformations. The current point in Figure 12 indicates the point in the history of transformations that is kept in the viewer where the user is viewing the representation. If a user selects a different phase or transformation to perform when she has not currently applied all of the transformations received, then the compiler is instructed to undo the appropriate number of transformations to reflect the current representation shown in the viewer.

Once a user decides to proceed to compile the next function or terminate a compilation session, the state of the program representation of the function needs to be saved. The compiler saves this state by writing the linked list of transformation changes to a file. When a user initiates a compilation session for the same compilation unit, this file is read and these initial transformation changes are automatically applied. Thus, a user can easily make updates to a compilation of a file over a number of sessions.

## 6 CONCLUSIONS

We have described a new code improvement paradigm that changes the role of low-level code improvers. This new approach can help achieve the cost/performance trade-offs that are needed for tuning embedded applications. By adding interaction to the code improvement process, the user can gain an understanding of code improvement trade-offs by examining the low-level program representation, directing the order of code improvement phases, applying user-specified code transformations, and visualizing the impact on performance. This system can be used by embedded systems developers to tune application code, by compiler writers to prototype, debug, and evaluate proposed code transformations, and by instructors to illustrate code transformations.

## References

[1] K. Andrews, R. Henry, and W. Yamamoto. Design and implementation of the UW illustrated compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 105–114, June 1988.

[2] B. Appelbe, K. Smith, and C. McDowell. Start/pat: a parallel-programming toolkit. *IEEE Software*, 6:29–40, 1988.

[3] M. E. Benitez. *Retargetable Register Allocation*. PhD thesis, University of Virginia, 1994.

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[5] M. E. Benitez and J. W. Davidson. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 International Conference on Programming Languages and Architectures*, pages 105–124, March 1994.

[6] M. Boyd and D. Whalley. Isolation and analysis of optimization errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–35, June 1993.

[7] M. Boyd and D. Whalley. Graphical visualization of compiler optimizations. *Journal of Programming Languages*, 3:69–94, 1995.

[8] J. Browne, K. Sridharan, J. Kiall, C. Denton, and W. Eventoff. Parallel structuring of real-time simulation programs. In *COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference*, pages 580–584, 1990.

[9] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[10] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pages 1–9, May 1999.

[11] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 2002 (to appear).

[12] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, pages 459–472, November 1991.

[13] Guido Costa Souza de Araujo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University, June 1997.

[14] C. Dow, S. Chang, and M. Soffa. A visualization system for parallelizing programs. In *Proceedings of Supercomputing*, pages 194–203, 1992.

[15] Christine Eisenbeis and Sylvain Lelait. LoRA: a package for loop optimal register allocation. In *3rd International Workshop on Code Generation for Embedded Processors, Witten, Germany*, March 1998.

[16] Daniel Kästner. Ilp-based approximations for retargetable code optimization. In *Proceedings of the 5th International Conference on Optimization: Techniques and Applications*, 2001.

[17] T. Kisuki, P. Knijnenburg, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 237–248, October 2000.

[18] Jeffrey L. Korn and Andrew W. Appel. Traversal-based visualization of data structures. In *IEEE Symposium on Information Visualization*, pages 11–18, October 1998.

[19] Haris Lekatsas. *Code compression for embedded systems*. PhD thesis, Princeton University, 2000.

[20] Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Boston, 1997.

[21] Shih-Wei Liao, Amer Diwan, Jr. Robert P. Bosch, Anwar Ghuloum, and Monica S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, 1999.

[22] Peter Marwedel and Gert Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[23] W. G. Morris. CCG: A prototype coagulating code generator. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 45–58, June 1991.

[24] D. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 39–48, 1989.

[25] Ashok Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*. PhD thesis, Princeton University, November 1998.

[26] R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pages 206–210, June 2000.