Target-specific Global Code Improvement: Principles and Applications

Manuel E. Benitez
meb1u@virginia.edu
804-982-2200

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

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Current and future high-performance systems require language processors that can generate code that fully exploits the power of the underlying architecture. A key and necessary component of such language processors is a global code improver. This article describes the key principles behind the design and implementation of a global code improver that has been use to construct several high-quality compilers and other program transformation and analysis tools. The code improver, called *vpo*, employs a paradigm of compilation that has proven to be flexible and adaptable—all code improving transformations are performed on a target-specific representation of the program. The aggressive use of this paradigm yields a code improver with several valuable properties. Four properties stand out. First, *vpo* is language and compiler independent. That is, it has been used to implement compilers for several different computer languages. For the C programming language, it has been used with several front ends each of which generates a different intermediate language. Second, because all code improvements are applied to a single low-level intermediate representation, phase ordering programs are minimized. Third, *vpo* is easily retargeted and handles a wide variety of architectures. In particular, *vpo*'s structure allows new architectures and new implementations of existing architectures to be accommodated quickly and easily. Fourth and finally, because of its flexible structure, *vpo* has several other interesting uses in addition to its primary use in an optimizing compiler. This article describes the principles that have driven the design of *vpo* and the implications of these principles on *vpo*'s implementation. The article concludes with a brief description of *vpo*'s use as a back end with front ends for several different languages, and its use as a key component for the realization of several other applications.

1 Introduction

The ability to produce a high-quality compiler that can be adapted to current and future architectures is critical to building and marketing computer systems successfully. A key component of a high-quality, production compiler is a global optimizer. This article describes the architecture of a code improver called *vpo*.† The code improver has several interesting characteristics. First, it is easily retargetable. Production-quality compilers can be constructed in a matter of weeks. Second, it has been retargeted to a wide variety of machines. It is able to handle complex instruction set machines (CISCs) equally as well as reduced instruction set machines (RISCs). Third, its organization and structure are such that it is simple to modify so that it can accommodate new architectural features as they appear. Fourth, all code improvements are performed on machine- and language-independent representation that encodes machine-specific instructions. This allows *vpo* to be largely machine-independent, yet efficiently handle machine-specific features such as register allocation, instruction scheduling, memory latencies, multiple condition code registers, etc. It, in effect, improves object code. Similarly, *vpo* is largely language- and application- independent. It has been used to construction compilers for several imperative languages such as C, Pascal, and Ada. A PL/I compiler is currently under construction. Finally, because of its flexible structure and organization, *vpo* has been adapted for use in a variety of other contexts. It has been used to build machine-code-to-machine-code translators, to build emulators and simulators

†The more accurate terms code improver and code improvement are used throughout this paper instead of optimizer and optimization.

Technical Report CS-94-42
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

for both existing and proposed architectures, to gather trace information for use in cache and virtual memory trace-driven simulations, and for predicting execution times of straight-line code for hard-real-time systems.

A number of *vpo*'s code improvement algorithms have been presented elsewhere [DAVI94, BENE94b, BENE91, DAVI86]. This paper describes the principles that have driven the design of *vpo* and the implications of these principles on *vpo*'s architecture. The paper also describes *vpo*'s use in global, optimizing compilers for several languages and its use as a key component in several nontraditional applications. We know of no other code improvement system that has been used with as large a variety of front ends and languages, targeted to as many different machines, and is a key component of several other applications.

2 Design principles

vpo evolved into its current state over a period of about ten years. During this period, several rules or design principles have emerged. Three of these principles have played a key role in allowing *vpo* to grow and adapt to changing demands placed by new architectures and new applications. These three principles are:

- all code improvements are effectively machine-dependent,
- instruction selection must be done on demand, and
- all code improvements are important some of the time.

The following subsections elaborate on and provide justification for these principles.

2.1 Machine-dependence of code improvements

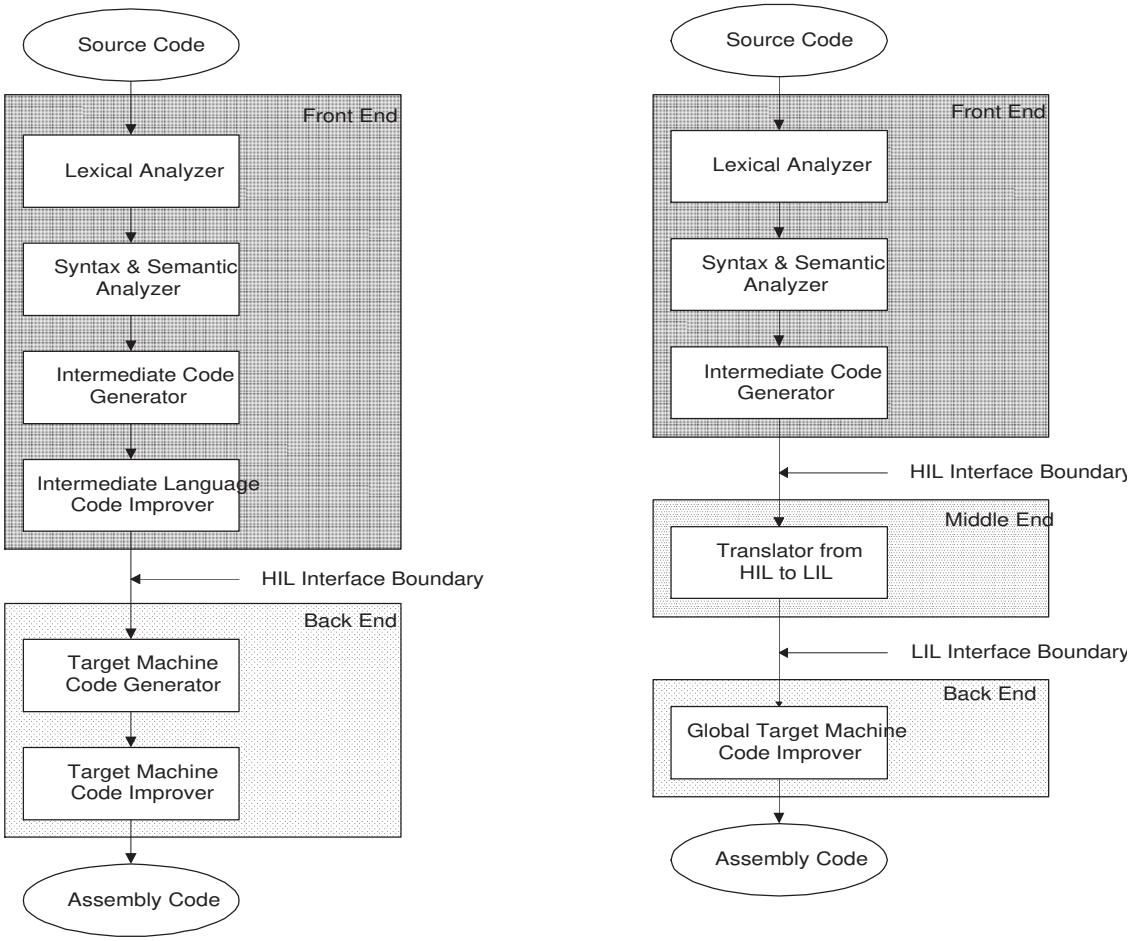
A global, optimizing compiler must perform a comprehensive set of code improvements in order to produce high-quality code for a wide range of machines. A partial list of code improvements that must be included in the compiler's repertoire is:

- register assignment and allocation,
- loop-invariant code motion,
- evaluation order determination,
- constant propagation,
- loop unrolling,
- inline function expansion,
- common subexpression elimination,
- induction variable elimination,
- constant folding,
- dead code elimination,
- instruction scheduling, and
- memory access coalescing.

This list of code improvements traditionally is divided into two groups: those that are considered to be *machine-dependent* and those that are *machine-independent*. Machine-dependent code improvements are those that, to be applied most effectively, require specific knowledge of the target machine. Obviously, code improvements such as register allocation and instruction scheduling are machine dependent. Somewhat less obvious, but no less machine dependent are inline function expansion and loop unrolling. Inline function expansion can be performed most effectively when details of the target machine's instruction cache is available[MCFA89, MCFA91]. Similarly, when unrolling a loop, the unroll factor depends on the number of target machine registers available, characteristics of the target machine's jump instructions, and characteristics of the instruction pipeline as well as the size of the instruction cache [WEIS87].

Machine-independent code improvements are those that do not require specific knowledge of the target machine, and they can be applied effectively independently of machine-dependent transformations. Examples from

the above list typically included in this group are loop-invariant code motion, induction variable elimination, common subexpression elimination, constant propagation, constant folding, and dead code elimination. Because of their presumed machine-independence, these code improvements are often applied to a high-level intermediate language representation of the source program. This division of code improvements results in a compiler that has a structure shown in Figure 1a.



a) Structure resulting from use of a single HIL

b) Structure resulting from use of a HIL and a LIL

Figure 1. Structure of two compiler organizations.

Unfortunately, these code improvements are not machine-independent. For example, loop-invariant code motion and induction variable elimination are machine dependent because their effective application requires knowledge of the addressing modes available on the target machine [BENE94b]. Similarly, common subexpression elimination and evaluation order determination are machine dependent because their effective application requires knowledge of the target machine's instruction set [DAVI84a, DAVI86]. Effective constant propagation requires knowledge of the calling convention, the machine's data paths, and the instruction set support for dealing with constants [BENE94b].

Constant folding is machine dependent because the addressing modes of many machines limit the size of an integer constant. Consequently, folding two integer constants may yield a result larger than the immediate addressing mode can handle. A reasonable solution is to fold constants early, and assume the back end will handle appropriately any large constants produced. After all, the back end must handle any large constants that appear in the source code (*i.e.*, they were not synthesized by constant folding). This approach solves half the problem. Because some machine-dependent code transformations (*e.g.*, inline function expansion and loop unrolling) will expose new opportunities for constant folding, to be most thorough, constant folding will still need to be done in the back end.

On the other hand, dead code elimination, applied in isolation, is machine independent. We cannot think of any situation where a target machine's characteristics would warrant not deleting code that will never be executed. Unfortunately, dead code elimination interacts with machine-dependent code improvements such as inline function expansion, constant folding, and constant propagation. In conjunction with constant propagation, inline function expansion exposes new opportunities for dead code elimination by effectively propagating constants across calls. Similarly, constant folding may expose new opportunities for dead code elimination by allowing conditional expressions to be evaluated at compile time. Hence, effectively *there are no machine-independent code improvements.*

2.2 On-demand instruction selection

Traditional compilers perform target machine code generation after machine-independent code improvement (see Figure 1a). There are several problems with this organization. One problem is that comprehensive code generators operate locally and while they generate a locally best sequence, they often "go too far" in their attempt to best utilize the target machine's instruction set. Complicated addressing modes and complex instructions are emitted, when in fact, a better global sequence might use a more primitive addressing mode or a sequence of simple instructions. For example, an aggressive code generator might use a scaled addressing mode because locally this is the best choice. However, when information from predecessor and successor blocks is considered, a better choice might be a primitive sequence where a subcomputation is exposed and can be reused.

A second, more serious problem with the compiler organization in Figure 1a is that target machine code is not exposed to global code improvement. Code generators often introduce new expressions and new control-flow paths that were not visible in the intermediate language representation of the program. Better final code can be produced if the code emitted by the code generator is subjected to thorough global improving transformations.

A third problem is that any further changes in the target machine code that might be warranted, say because a machine-dependent transformation introduced an opportunity to use a better machine code sequence, must be handled in an ad hoc manner. A common solution is to use a pattern-directed, machine-specific peephole optimizer that is driven by a file of patterns. There are three well-known problems with this approach. First, it is impossible, a priori, to determine what post-code generation transformations might be made and construct a comprehensive set of patterns. Second, the ad-hoc approach hampers retargetability. Each new target machine requires a new pattern file. Automatically generating a set of patterns is possible, but suffers from incompleteness[DAVI84b]. Third, global program information has been lost. Typical peephole optimizers operate locally on small sections of code.

The ability to do instruction selection "on-demand" solves these problems. By on-demand, we mean that at any point during processing by the back end, an instruction selection phase can be invoked to determine a new target

machine code sequence. On demand instruction selection is provided by a retargetable peephole optimizer that employs global data-flow information. Section 3 describes how on-demand instruction selection in *vpo* solves the previously described problems.

2.3 Relative importance of code improvements

Section 2.1 contained a list of some common code improvements. A question that often arises is which code improvements are important? Or, put another way, which code improvements should be implemented? Clearly, register allocation is so important that any good compiler must include this code improvement. But what about the others? We have measured the impact of the listed code improvements on a wide variety of machines, with a variety of languages, and in a variety of application domains (*e.g.*, real-time software, digital signal processing, image processing, scientific codes, business applications, *etc.*). From these measurements, we can make the following observations:

- any single code improvement will only affect a subset of the programs to which it is applied,
- for some programs the effect will be small and for others it will be large, and
- the effect of applying a set of code improvements is often greater than the sum of the individual effects.

Thus, a good optimizing compiler uses a collection of code improvements where each transformation produces a small benefit most of the time and a large benefit occasionally. Furthermore, many code improvements interact in unpredictable ways. Forgoing one may reduce the effectiveness of another.

3 Architecture of *vpo*

Of the three principles listed, the characterization of all code improvements as machine dependent has the most impact on *vpo*'s architecture. We need a intermediate representation and implementation that permits code improvements to be done in such a way that target machine characteristics can be taken into account. We also want an implementation that can be quickly retargeted to a new machine. While these may seem like contradictory requirements, *vpo* demonstrates that such an implementation is possible, and the resulting code improver is highly effective.

3.1 Intermediate representation

We require a representation that supports the application of both local and global code improvements at the level of the target machine. The representation used is called RTL (Register Transfer Lists). RTL consists of a representation for machine instructions and a method of conveying source language information to *vpo*. The part of RTL that encodes machine instructions is based on the ISP notation of Bell and Newell. Original RTL descriptions were ad hoc [DAVI85]. Over the years, this portion of the RTL representation has evolved into a compact, well-defined language for describing the instruction sets of machines [BENE94a]. Over 20 architectures have been described.

However, to be most effective, a global code improver needs more than target machine information. It needs control over the allocation of objects and information about the source program. Most code improvers do not participate in the placement of data. For global data, for example, the code generator simply emits the necessary assembly language directives that allocate space for program objects without regard to their relative location. However, for many of today's high performance processors, proper location of data can affect overall performance [DAVI94]. In many cases, a global code improver, using information gathered during its data-flow analysis phase, can place data so that it can be referenced more efficiently. Similarly, to be effective and perform safe code transformations, a code

improver needs some information from the source program. For example, the C language contains the `volatile` type qualifier. When applied to an object declaration, this qualifier informs an implementation that the value of the object may be altered by outside agents. *vpo* must know which objects are volatile so that these objects do not participate in code transformations that can change when or where an object is referenced. Similarly, some languages have pointers. Unconstrained use of pointers makes some code improvements difficult to apply safely. On the other hand, many languages are strictly typed. If type information is available, a code improver can often make transformations that, without type information, it would be forced to forgo. There are other examples where information available at the source level can impact the effectiveness of code improvements (*e.g.*, Fortran equivalence statements, C++ reference type, Ada pragmas, *etc.*).

vpo's RTL language includes directives that allow a front end, if desired, to supply relevant source language information to *vpo*. Additionally, directives are available for conveying the static memory allocation requirements of a program to *vpo*. It uses this information to place data and perform more effective code transformations.

3.2 Implementation overview

The overall organization of a global optimizing compiler that uses *vpo* is shown in Figure 1b. In this organization, there are two intermediate languages. The high-level intermediate language (HIL) corresponds to the intermediate language often used in a compiler with a traditional organization [CHOW83, NELS79, TANE82]. The HIL serves to make the front end machine independent so that it can be used for a variety of target architectures with as little modification as possible. The low-level intermediate language (LIL) is RTL. An RTL representation of a program is supplied to *vpo* via a file interface. While this has a minor impact on compile-time efficiency, the benefits far outweigh any disadvantages (see Section 4).

After reading the RTL file and building the necessary internal data structures, a central code improvement routine is invoked per function. A high-level version of this routine is shown in Figure 2. The first steps are to build the control flow graph and perform control-flow improvements. Examples of these transformations include removing unreachable code, removing useless jumps by rearranging basic blocks, and removing branch chains. Following these actions, local data flow analysis is performed to set up def-use information. Using this information, a preliminary pass of instruction selection is done. At this point, the preliminary analysis necessary for creating the Static Single Assignment form [CYTR91] is performed (lines 7—10). We have found the SSA form and the RTL representation to be a particularly good match. Following the SSA construction, global def-use information is collected and instruction selection is re-invoked. At this point, local register allocation is performed. This maps any pseudo-registers to hardware registers. If the local register allocator assigns new hardware registers, instruction selection is redone. At this point, it is worth noting that instruction selection is not performed over the entire function each time. Rather, as transformations make changes, these RTLs are marked, and only regions of code that have changed are processed again. The next step is to find all the loops in the program. The information gathered by the *BuildDominatorTree* and *FindDominanceFrontiers* is used to do this. The routine *EstimateExecutionFrequency* weights loops according to their estimated frequency. Static analysis or profile-collected information can be used.

Lines 19 through 38 are a loop where selected code improvements are applied and re-applied until the code converges. *ColorLocalVariables* uses a graph coloring algorithm to assign hardware registers to a local variable that has been allocated to a register. *CommonSubexpressionElimination* does common subexpression elimination as well as

```
Line
 1.   proc Improve is
 2.        BuildControlFlowGraph()
 3.        ControlFlowTransformations()
 4.        SetLocalLinks()
 5.        InstructionSelection()
 6.        EvaluationOrderDetermination()
 7.        BuildDominatorTree()
 8.        FindDominanceFrontiers
 9.        LiveVariableAnalysis()
10.        BuildMinimalSSAForm()
11.        SetGlobalLinks()
12.        InstructionSelection()
13.        LiveVariableAnalysisUpdate()
14.        if LocalRegisterAssignment() then
15.             InstructionSelection()
16.             endif
17.        FindLoops()
18.        EstimateExecutionFrequency()
```

```
Line
19.        do
20.             A ← False
21.             do
22.                  C ← False
23.                  LiveVariableAnalysisUpdate()
24.                  A ← DeadVariableElimination()
25.                  if ColorLocalVariables() then
26.                       C ← InstructionSelection()
27.                       A ← True
28.                       endif
29.                  while C
30.             if A then
31.                  C ← CommonSubexpressionElimination()
32.                  LiveVariableAnalysisUpdate()
33.                  C ← C ∨ DeadVariableElimination()
34.                  C ← C ∨ LoopTransformations()
35.                  C ← C ∨ InstructionSelection()
36.                  C ← C ∨ InlineFunctions()
37.                  endif
38.             while C
39.        ControlFlowTransformations()
40.        InsertFunctionPrologueandEpilogue()
41.        InstructionSelection()
42.        InstructionScheduling()
43.        endproc
```

Figure 2. Driver loop for *vpo*.

constant and copy propagation. *LoopTransformations* performs induction variable elimination, software pipelining, loop-invariant code motion, loop unrolling, and memory access coalescing. After the code has converged, the *ControlFlowTransformations* routine is invoked because many of the previous transformations may have made changes in the control flow. Following this, the prologue and epilogue code for the function is generated, and a final pass of instruction selection is performed to "stitch" this code in. *Improve* ends by invoking an instruction scheduler. For machines that do not require instruction scheduling, this routine will simply return.

The above structure has several advantages. First, phase-ordering problems are largely eliminated. As changes are made to the program, improvements previously applied are reapplied. Second, implementation of the code improvement algorithms is simplified. This is because they can assume that other transformations will be applied. For example, induction variable elimination (done in *LoopTransformations*) need not worry about eliminating dead variables that might be introduced. This will be handled by *DeadVariableElimination*. Additionally, the ability to do instruction selection on-demand also simplifies the implementation of the code improvements and it makes *vpo* easier to retarget. Many code transformations must insert new instructions (*e.g.*, loop-invariant code motion inserts code in a preheader, induction variable elimination requires the computation of a induced address, *etc.*). With *vpo*'s organization, these insertions can be sequences of simple, primitive instructions. Later, instruction selection will choose the most efficient target machine instructions. Because simple instructions are used, it means that the portion of the improver that inserts these instructions can often be made machine-independent. For example, most machines support register-to-register copies, register indirect addressing mode, and register-to-register ALU operations. A third advantage is that it is very simple to add new code improvements [MUEL92]. This advantage should not be taken

lightly. Consider that in the last ten years machines have been have brought to market that included the following features or implementation techniques:

- register windows,
- superscalar implementation,
- Very Long Instruction Word (VLIW),
- 64-bit implementation,
- branch with annul,
- exposed pipelines,
- large register sets (32 or more),
- superpipelined implementation,
- condition code registers,
- delayed branches,
- speculative execution, and
- exposed cache.

Each of these features or implementation techniques requires special handling by the compiler to fully exploit the target machine. Furthermore, in many application domains, special-purpose code improvements that have little effect on general-purpose codes, are required. For example, a compiler used to develop signal processing applications must be able to do software pipelining, handle recurrences, and control placement of data. All code improvements are important sometime. Consequently, the code improver must be improved and enhanced constantly to keep pace with innovations in computer architecture and to be usable in a variety of application domains. A final advantage is that it is possible to automate debugging *vpo* [WHAL94].

4 Applications

4.1 Retargetable Compilers

Obviously, the primary application of *vpo* is to build optimizing compilers, and it has proven to be very successful. Using a front end that emits code for an abstract stack machine, *vpo* has been used to build K & R C compilers for the following architectures:

- VAX-11
- Intel 80386
- Harris HCX-9
- CDC 176
- MIPS R2000/R3000
- AT&T DSP32
- Intergraph Clipper
- Hewlett-Packard PA-RISC
- Motorola 68020
- Concurrent 3230
- National Semiconductor 3200
- IBM RT
- Motorola 88100
- Sun Sparc
- AT&T 3B15
- DEC Alpha

Work is underway to build compilers for the IBM RS/6000, Motorola's 65HC05 and DSP 56000, as well as some experimental machines.

Typically, it takes about a month of hard work to bring up a complete, tested compiler for a new machine. Code quality is not sacrificed for ease of retargeting. *vpo* is able to generate very high-quality code—code that is as good as the production compiler for most machines. Using the C component of the $SPEC89_{int}$ benchmark suite, Figure 3 compares *vpo*-based compilers we actively maintain to several production compilers on five machines. In terms of compile time, on the Sun3 and Sparc, the *vpo*-based compiler and the native compiler run at about the same speed. On the SGI, the native compiler is approximately 13 percent faster than *vpo*. On all hosts, *gcc* is consistently 10 to 15 percent faster. *(Note: All compiles were done with the -O option. The full paper will contain detailed statistics including the versions of the compilers used.)*

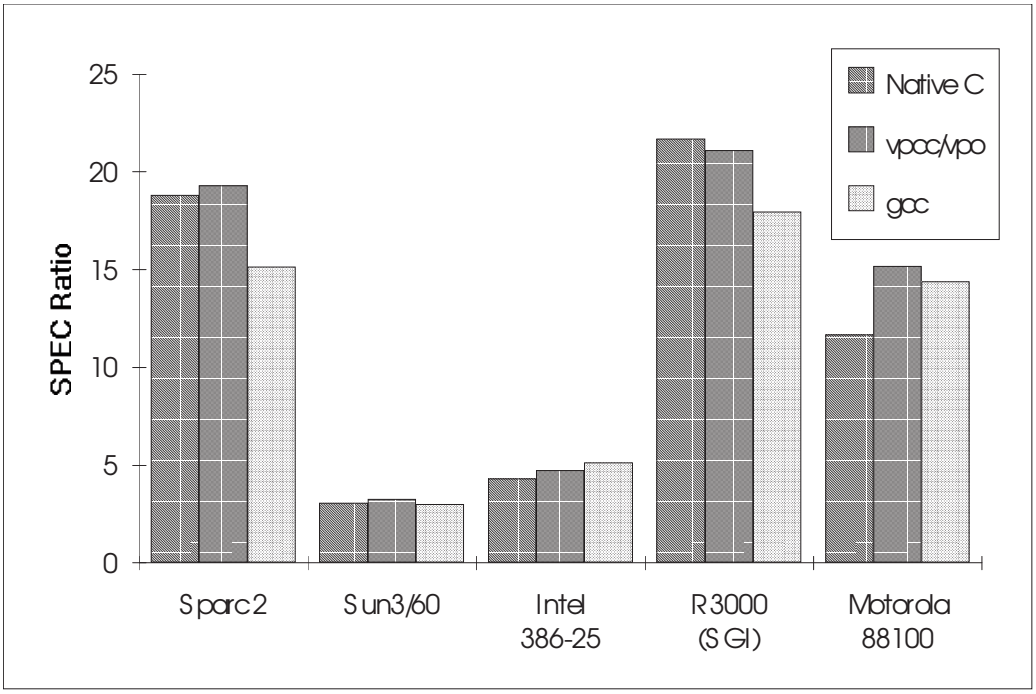


Figure 3. Comparison of run-time performance on $SPEC89_{int}$.

Because of the RTL file interface and its underlying structure, *vpo* is front end as well as language independent. Indeed, it has been used with a variety of C front ends each of which generates a different intermediate representation. It has been used with *lcc*'s front end (which produces DAGs) [FRAS95]. In this case, the middle end (see Figure 1b) traverses the DAGs emitting naive RTLs for the target machine. Work is underway with *lcc* to emit RTLs via an IBURG-produced code generator. The work with *lcc*'s front end illustrates *vpo*'s flexibility. The front end does some low-cost, high-yield code improvements [HANS83]. This in no way hinders the operation of *vpo*. It has also been used with an ANSI C front end developed by Hewlett-Packard Corporation that emits HPcode-Plus, a proprietary intermediate language similar to MIPS Ucode [CHOW83], and an ANSI C front end which produces trees from the Edison Design Group.

In addition to the C compilers, *vpo* has been used to produce global optimizing compilers for several other imperative languages. It has been used to produce a validated Ada compiler. This compiler used a front end that produced Diana, the standard intermediate language developed for Ada. It has been used to realize a Pascal compiler in conjunction with a front end that produced Pcode. Work is currently underway by Uniprise Systems, Inc. to produce a PL/I compiler using *vpo*. Their compiler front end produces a quad-like intermediate language.

4.2 Object code conversion

Another application that *vpo* has been used for is conversion of programs that run on one processor to run on another processor. For example, suppose there exists an object program that runs on machine *X* and it is desirable to run that program on machine *Y*. If the source code is available, the easiest approach is to simply recompile the program on machine *Y*. For various reasons, this may not be possible. All or part of the source code may not be available (some

substantial portion of the code could be written in assembly language for performance reasons), the source code may be very machine-dependent, or it may be desirable to have the program run as if it were running on the original machine. A naive approach is to translate each *X* machine instruction in the original program, to an sequence of machine instructions for machine *Y*. One problem, of course, is that the resulting program will run more slowly than the original program if the two processors have about the same performance. If machine Y is faster than X, while the translated program may run faster, it will not achieve nearly the same performance as a "native" application. Typically, programs translated naively many run ten to fifteen times slower than they did on machine *X* (the performance depends on how different the architectures *X* and *Y* are and the effort expended on the translator).

Using *vpo*, the disparity in the run-times of the two programs can be reduced. Rather than translate X's machine instructions directly to Y machine instructions, RTLs for Y are emitted. *vpo* is applied to the resulting code to yield an optimized version of the program for the new machine. Experiments indicate that reductions in the run-time on the order of two to six times are possible. One application of this approach has been used to develop avionics software for an advanced aircraft, the Swedish JAF 39. The avionics and flight-control systems for this aircraft use several special-purpose microprocessors called the DA28. While necessary for the onboard avionics, developing software in this environment is difficult and expensive. An option considered was to take an aircraft and chop the wings off and reassemble it in a building. A more cost effective solution was to develop the software using a commercial microprocessor-based system simulating the several DA28s on the aircraft. To accomplish and obtain satisfactory performance, the DA28 object code for the avionics and flight-control systems were translated to RTLs for a MIPS R3000-based Silicon Graphics system. The resulting code was optimized by *vpo*. It is estimated that this particular Silicon Graphics system is rated at 60 native MIPS while the DA28 is rated at 2 native MIPS. Simulating a single DA28 assuming a one-to-one mapping from a DA28 instruction to a R3000 instruction would yield a speedup of 30. However, the R3000 must simulate several DA28s and the DA28 is a CISC machine, so the initial mapping was on the order of one DA28 instruction to 15 R3000 instructions. Using *vpo*, it was possible to reduce this by roughly a factor of three. Another advantage of this approach is that the same front end is used for development and production of the native software.

In addition to the above application, this approach has also been used to implement translators that take existing applications that run on older CISC architectures, and convert them to run on RISC architectures. Because current RISC architectures are much, much faster than the older CISC architectures (approaching 100X), the applications typically run faster on the RISC machines even using a naive translation approach. However, when compared to a "native" application, the naively translated application runs much slower (sometimes by as much as a factor of ten). It has been reported that *vpo* is able to reduce this to a factor of three. (*Note: We are getting permission to supply more details).*

4.3 Architecture and performance evaluation

Today's high-performance processors rely heavily on optimizing compiler technology to produce code that exploits the processors capabilities [HENN90]. To explore and evaluate new architectures, it has become crucial to have access to optimizing compiler technology. As shown above, *vpo* provides this access. In addition, it is necessary to have tools and techniques that help collect detailed measurements of an architecture's dynamic behavior running real programs.

Unfortunately, gathering detailed measurements of the execution behavior of an instruction set architecture can be difficult. A major problem is that gathering detailed dynamic measurements of an architecture using typical user programs reading typical data sets can consume significant computation resources. For example, a popular way to gather execution measurements is to simulate the architecture. This technique is often used when the architecture in question does not yet exist, or is not yet stable and available for production use. Depending on the level of the simulation, programs can run 100 to 500 times slower than directly-executed code [HUGU87]. Tracing is another alternative one can use if the architecture being measured exists, is accessible, and tracing is possible on that machine. Tracing can be even slower than simulation. Because of the large performance penalties with these methods, the tendency is to use small programs with small data sets. The relevance of measures collected this way is always subject to question.

Fortunately, *vpo* provides a framework for easily and efficiently capturing measurements of the dynamic behavior of an architecture. During the course of processing a program, *vpo* collects a wealth of information about the program. It constructs the flow graph which consists of basic blocks connected by flow arcs, it locates and identifies loops, and it selects and identifies target machine instructions. This information along with the description of the machine's instruction set (MD) can be used to collect statistics on the dynamic behavior of an existing or proposed architecture. Measurements of

- instruction path length,
- instruction type distribution,
- memory reference size distribution,
- register usage,
- data type distribution,
- average number of instruction between branches,
- instruction path size,
- addressing mode distribution,
- memory reference address distribution,
- condition code usage,
- conditional branches taken, and
- loop execution frequencies

are gathered routinely. Other measurements of interest can usually be obtained with modest additional effort. Implementation and performance details of the system can be found in WHAL90 and DAVI91a. With some extensions the system can be used to gather information so that execution times of straight-line code sequences can be predicted in hard-real-time systems [HARM92].

vpo's framework also permits analysis of the instruction set architecture of uninstantiated architectures [DAVI90, DAVI91b, ALEX93]. This allows early design decisions to be evaluated using applications that will be run on the architecture. Normally the last step of the code improvement process is to translate the RTLs to assembly language for the target machine. When the target machine does not exist, rather than emit assembly code for the target machine, assembly code for a host machine is emitted that emulates the non-existent target machine instructions. Information about the effects of the instruction are emitted as if the target architecture existed.

An increasingly important aspect of machine design is the construction of a memory hierarchy that meets the processor's demand for instructions and data. One technique for evaluating memory hierarchy performance is to use trace-driven simulation [SMIT82]. In this approach, traces of the addresses fetched by programs are captured and used to simulate the cache or memory system. Alternative designs can be simulated to determine what gives the best performance for a particular design. One difficulty with trace-driven simulation is the expense of gathering traces. *vpo* supports the efficient gathering and analysis of traces [WHAL93]. *(Note: Details to be included in full paper. We also describe vpo's use in developing hard-real-time systems[ARNO94, MUEL94].)*

5 Related Work

Auslander and Hopkins describe the implementation of the PL.8 compiler [AUSL82]. The PL.8 code improver employed phase iteration, and the authors also noted that this greatly simplified the implementation of the various code improving transformations. While the PL.8 compiler used a low-level intermediate language, it did not represent target machine instructions. The final phase of the code improver was responsible for mapping the optimized intermediate language to target machine instructions. Consequently, as Auslander and Hopkins note, the compiler is biased in favor of target machines that are similar to the regular and simple register-to-register intermediate language.

Johnson and Miller describe a global code improver that operated at the machine-code level [JOHN86]. While the code improver supported several front ends, it only generated code for the Hewlett-Packard Precision family of machines. Much like *vpo*, the code improver relied on the front ends to collect and pass information about the set of memory locations actually or potentially touched by each variable reference or pointer dereference.

Perhaps the most closely related work is the GNU C compiler [STAL92]. This is not surprising as both the GNU C compiler (*gcc*) and *vpo* are descendents of a compiler that used RTLs and performed local transformations [DAVI81, DAVI84a]. While *gcc* provides support for multiple front ends such as C, C++, and Objective C, these front ends are tightly integrated with the compiler. No independently developed front end has been used with *gcc*'s code improver. This is because the implementation does not provide a convenient interface to the back end. This also prevents *gcc* from being used for other applications. Like *vpo*, *gcc* has been retargeted to many machines—23 at last count. *gcc*'s code improver is quite large. It is 47,000 lines of code (this does not include machine descriptions or the tools to process them). In contrast, *vpo* is 30,000 lines. While *gcc* includes many code improvements, it does not employ phase iteration. Rather, a few key phases are simply redone. This may explain *gcc*'s better compile time, but slightly less efficient code for four of the five machines measured in Figure 3.

6 Summary

We know of no other code improvement system that 1) has been used with a variety of front ends and languages, 2) has been targeted to a wide array of architectures, and 3) has been used to develop other additional applications such as object-code-to-object code translators and performance evaluation tools. *vpo*'s utility, flexibility, and adaptability are due to its use of a low-level intermediate language that represents target-specific instructions and its overall structure. *vpo*'s structure and intermediate representation are due to the aggressive application of three design principles. First, there are no machine-independent code improvements. Second, instruction selection is necessary throughout the entire code improvement process, and third, a code improver for use in a wide variety of application domains must support a comprehensive suite of code improvements.

7 References

[ALEX93] Alexander, M. J., Bailey, M. W., Childers, B. R., Davidson, J. W., and Jinturkar, S., 'Memory Bandwidth Optimizations for Wide-Bus Machines', in *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, January 1993, pp. 401—423.

[ARNO94] Arnold, R., Mueller, F., Whalley, D., and Harmon, M., 'Bounding Worst-Case Instruction Cache Performance,' to appear in *Proceedings of the Twelveth IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[AUSL82] Auslander, M. and Hopkins, M., 'An Overview of the PL.8 Compiler,' *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, June 1982, pp. 22—31.

[BENE94a] Benitez, M. E., *Retargetable Register Allocation*, Ph.D. Dissertation, University of Virginia, 1994.

[BENE94b] Benitez, M. E. and Davidson, J. W., 'The Advantages of Machine-Dependent Global Optimization,' in *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures*, Zurich, Switzerland, March 1994, pp. 105—124.

[BENE91] Benitez, M. E. and Davidson, J. W., 'Code Generation for Streaming: an Access/Execute Mechanism,' in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991, pp. 132—141.

[CHOW83] Chow, F. C., *A Portable Machine-Independent Global Optimizer—Design and Measurements*, Ph.D. Dissertation, Stanford University, 1983.

[CYTR91] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,' *ACM Transactions on Programming Languages and Systems* **2**(2), pp. 451—490.

[DAVI94] Davidson, J. W. and Jinturkar, S., 'Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses,' *Proceedings of the SIGPLAN '94 Symposium on Programming Language Design and Implementation,* Orlando, FL, June 1994, pp. 186—195.

[DAVI91a] Davidson, J. W. and Whalley, D. B., 'A Design Environment for Addressing Architecture and Compiler Interactions,' *Microprocessors and Microsystems,* **15**(9), November 1991, pp. 459—472.

[DAVI91b] Davidson, J. W. and Whalley, D. B. 'Methods for Saving and Restoring Register Values across Function Calls,' *Software—Practice & Experience* **21**(2), February 1991, pp. 149—165.

[DAVI90] Davidson, J. W. and Whalley, D. B., 'Reducing the Cost of Branches by Using Registers,' *Proceedings of the 17th International Symposium on Computer Architecture,* May 1990, pp. 182—191.

[DAVI86] Davidson, J. W., 'A Retargetable Instruction Reorganizer,' *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* **21**(7), June 1986, pp. 23—241.

[DAVI85] Davidson, J. W., *'Simple Machine Description Grammars'* Technical Report TR85-22, Department of Computer Science, University of Virginia, Charlottesville, VA., November 1985.

[DAVI84a] Davidson, J. W. and Fraser, C. W., 'Code Selection through Object Code Optimization,' *ACM Transactions on Programming Languages and Systems* **6**(4), October 1984, pp. 7—32.

[DAVI84b] Davidson, J. W. and Fraser, C. W., 'Automatic Generation of Peephole Optimizations,' in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 111—116.

[DAVI81] Davidson, J. W., *Simplifying Code Generation through Object Code Optimization*, Ph.D. Dissertation, University of Arizona, 1981.

[FRAS95] Fraser, C. W. and Hanson, D. R., *A Retargetable C Compiler: Design and Implementation*, Benjamin Cummings, Redwood City, CA, 1995.

[HANS83] Hanson, D. R., 'Simple code optimizations,' *Software—Practice & Experience* **15**(12), pp. 1205—1212.

[HARM92] Harmon, M. G., Baker, T. P. and Whalley, D. B., 'A Retargetable Technique for Predicting Execution Time,' *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1992, pp. 68—67.

[HENN90] Hennessy, J. L. and Patterson, D. A., *Computer Architecture: a quantitative approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[HUGU87] Huguet, M., Lang, T., and Tamir, Y., 'A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements,' *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques,* St. Paul, MN, June 1987, pp. 14—25.

[JOHN86] Johnson, M. S. and Miller, T. C., 'Effectiveness of a Machine-Level, Global Optimizer,' *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, pp. 99—108.

[MCFA91] McFarling, S., 'Procedure Merging with Instruction Caches,' *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, June, 1991, pp. 71—79.

[MCFA89] McFarling, S., 'Program Optimization for Instruction Caches,' *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989, pp. 183—191.

[MUEL94] Mueller, F., *Static Cache Simulation and Its Application*, Ph.D. Dissertation, Florida State University, August 1994.

[MUEL92] Mueller, F. and Whalley, D. B., 'Avoiding Unconditional Jumps by Code Replication,' *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992, pp. 322—330.

[NELS79] Nelson, P. A., 'A Comparison of PASCAL Intermediate Languages,' *Proceedings of the SIGPLAN Symposium on Compiler Construction*, Denver, CO, August 1979, pp. 208—213.

[SMIT82] Smith, A. J., Cache Memories, *Computing Surveys* **14**(3), pp. 473—530.

[STAL92] Stallman, R., *Using and Porting GNU CC*, Free Software Foundation, 1992.

[TANE82] Tanenbaum, A. S., Staveren, H. V., and Stevenson, J. W., 'Using Peephole Optimization on Intermediate Code,' *Transactions on Programming Languages and Systems* **4**(1), January 1982, pp. 21—36.

[WEIS87] Weiss, S. and Smith, J. E., 'A Study of Scalar Compilation Techniques for Pipelined Supercomputers,' *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987, pp. 105—109.

[WHAL94] Whalley, D. B., 'Automatic Isolation of Compiler Errors,' to appear in *ACM Transactions on Programming Languages and Systems*.

[WHAL93] Whalley, D. B., 'Techniques for Fast Cache Performance Evaluation,' *Software—Practice & Experience* **23**(1), pp. 95—118.

[WHAL90] Whalley, D. B., Ease: An Environment for Architecture Study and Experimentation, Ph.D. Dissertation, University of Virginia, 1990.