Using De-optimization to Re-optimize Code

Stephen Hines, Prasad Kulkarni,
David Whalley
Computer Science Dept.
Florida State University
Tallahassee, FL 32306-4530
(hines, kulkarni, whalley)@cs.fsu.edu

Jack Davidson
Computer Science Dept.
University of Virginia
Charlottesville, VA 22904-4740
jwd@virginia.edu

ABSTRACT

The nature of embedded systems development places a great deal of importance on meeting strict requirements in areas such as static code size, power consumption, and execution time. In order to meet these requirements, embedded developers frequently generate and tune assembly code for applications by hand, despite the disadvantages of coding at a low level. The phase ordering problem is a well-known problem affecting the design of optimizing compilers. Hand-tuned code is susceptible to an analogous problem to phase ordering due to the process of iterative refinement, but there has been little research in mitigating its effect on the quality of the generated code. This paper presents an extension of the VISTA framework for investigating the effect and potential benefit of performing de-optimization before re-optimizing assembly code. The design and implementation of algorithms for de-optimization of both loop-invariant code motion and register allocation, along with results of experiments regarding de-optimization and re-optimization of previously generated assembly code are also presented.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms

Algorithms, Experimentation, Performance

Keywords

Assembly Translation, De-optimization, Phase Ordering.

1. INTRODUCTION

The phase ordering problem is a long-standing problem involved in the development of compilers and related tools [18]. Simply put, the phase ordering problem is that there exists no single sequence of optimization phases that will produce optimal code for every function in every application on every architecture. Different optimizations can enable or disable further optimizations depending on the characteristics of the function being compiled as well as the target architecture [19]. These enabling or disabling factors can greatly impact the design and implementation of optimizing compilers. One of the most critical factors is register pressure. Many optimization phases consume registers, leaving few registers available for later transformations. Depending on the chosen optimization phase order, later phases may be prohibited from having any effect at all. Embedded systems are more susceptible to the phase ordering problem since they typically have non-orthogonal instruction sets and fewer registers, contributing to even greater register pressure.

One approach towards minimizing the effects of the phase ordering problem is to produce a compiler with the ability to apply phases repeatedly until no additional improvements can be obtained. Very Portable Optimizer (VPO) attempts to provide these exact features [1]. VISTA is an optimizing compiler framework that incorporates VPO and employs iteration of optimization phase sequences and a genetic algorithm search for effective phase sequences in an effort to minimize the effects of the phase ordering problem [14].

With embedded software development, size and timing constraints are both more important and more stringent than they are in traditional software. This increased focus forces a great deal of embedded applications development to be done in assembly language, since the ability to hand-tune code can produce smaller and faster executables than using a high-level language with a good optimizing compiler. While hand-tuning assembly code may appear to be an adequate solution for the requirements of embedded software development, it is still subject to a similar phase ordering problem albeit in a slightly different manner. Rather than an explicit phase ordering being applied to the function, the programmer attempts to improve the code based on intuition and educated decisions. These design decisions are similar to phase ordering decisions made by an optimizing compiler, so it is possible that a better solution exists even with hand-tuned assembly code.

We propose using the facilities of VISTA to further tune optimized assembly code. A translator is constructed to convert assembly code to VISTA's RTL input format. To handle the phase ordering problem, we will apply the concept of de-optimization. Prior optimizations that affect the phase ordering problem can be undone in a safe manner, so that different phase sequences can then be applied and

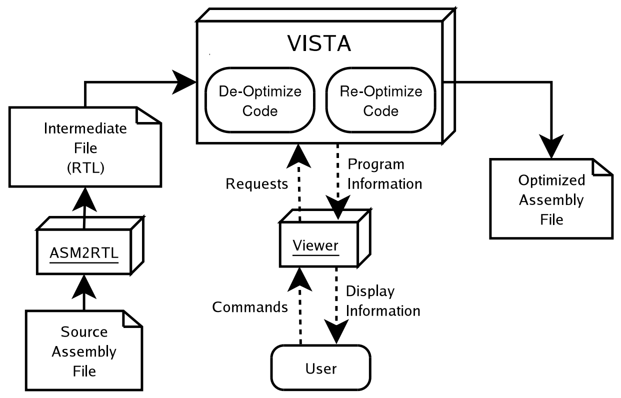

Figure 1: De-optimization and Re-optimization

tested. Both *loop-invariant code motion* and *register allocation* are de-optimized since they have considerable impact on register pressure and thus the phase ordering problem. Other optimizations are not selected because they either do not have much phase ordering impact, or there is no way to reconstruct the necessary information (*dead assignment elimination*). After de-optimization, VISTA can retune the code sequence using its genetic algorithm search for effective optimization sequences. Figure 1 shows the process of re-optimizing assembly code with VISTA.

The remainder of this paper can be broken down as follows: Section 2 reviews some related work in the areas of assembly translation and de-optimization. Section 3 discusses the experimental setting employed to investigate the benefits of de-optimization. We cover the process of assembly translation and the implementation of the ASM2RTL translator suite in Section 4. Section 5 focuses on the actual design and development of de-optimizations. We examine the results of performing the experiments in Section 6. Finally, Section 7 presents our concluding remarks regarding the use of de-optimizations for improving the re-optimization of code.

2. RELATED WORK

This section focuses on research areas that overlap in some manner with this work. To the best of our knowledge, there has been no other research in using de-optimizations to enhance the re-optimization of assembly code. However, there has been related work in the fields of assembly translation and de-optimization, particularly for facilitating reverse engineering, legacy binary translation, link-time optimization, and symbolic debugging of optimized code.

Binary translation is the process of converting an executable program from one particular platform to a different platform. Platform differences can include instruction set, application binary interface (ABI) specifications, operating system, and executable file format. The translation of binary programs has particular importance in the porting of legacy software for which source code is unavailable. There are a few key differences between binary translation and the assembly translation tools discussed in this work. With many binary translators, the stack layout remains constant and thus arrays and structures cannot be accidentally placed incorrectly. This limits the effectiveness of re-optimizations since even scalar local variables would be unable to be adjusted. Our assembly translation tools work at the level of assembly code and not binary executables.

Existing systems that perform binary translation include the Executable Editing Library (EEL) and the University of Queensland Binary Translator (UQBT). EEL is a library developed to facilitate the construction of tools that edit executables such as binary translators and optimizers [15]. EEL has also been used to construct programs that instrument executables for the purpose of gathering performance data. UQBT is a binary translator developed at the University of Queensland that was designed to be easily retargeted [4]. Similar to the work described in this paper, UQBT accepts SPARC and ARM input files among other formats. UQBT features several backends generating various types of output suitable for translation purposes, including Java virtual machine language (JVML), C, object code and VPO RTL. Although UQBT did have some success with smaller programs using the RTL backend, the overall RTL target required too much analysis information to be useful within the UQBT framework. Annotations supplied statically to our tools helps to alleviate some of the problems encountered by UQBT. Additionally, our translator can always fall back on translations that maintain program semantics at reduced opportunity for improvement such as keeping all local variables in the same initial order on the stack.

Link-time optimizations are closely related to the process of binary translation, in that they both operate on a similar amount of semantic content. Link-time optimizations focus on interprocedural opportunities that were not able to be addressed during the compilation of individual functions or modules. Alto is a link-time optimizer targeting the Compaq Alpha that features optimizations such as *register allocation*, *inlining*, *instruction scheduling*, and *profile-directed code layout* [16]. In contrast, the de-optimization and re-optimization components of VISTA are designed to work with the function as a basic unit.

De-optimizations have been applied in several areas ranging from the debugging of optimized code to the reverse engineering of executables. Due to the use of optimizing compilers and the development of complicated optimizations, debuggers have a need for additional information to provide proper interactive feedback when working with optimized executables. Hennessy attempted to address the problem of noncurrent variables when symbolically debugging optimized executables [9]. These are variables that may not contain their correct *current* value when walking through the optimized executable, perhaps due to *common subexpression elimination* or *dead assignment elimination.* Hölzle, Chambers and Ungar present another approach for debugging optimized code using dynamic de-optimization [11]. Their system however was limited to only debug the SELF object-oriented language and also requires the compiler to instrument the executable with necessary de-optimization information at various *interrupt points* within the program.

Reverse engineering is the technique of extracting high-level source information from binary executable files. The reconstruction of control flow graphs (CFGs) is a primary step in the reverse engineering process, allowing input assembly instructions to be better represented as the common loop and test constructs seen in a high-level language. CFGs for binary code tend to be more complicated than corresponding CFGs for the initial high-level source code. This increased complexity is only further exacerbated by optimizations that modify control flow such as *predication*, *speculation* and *instruction scheduling.* Snavely, Debray and Andrews have experimented with performing de-optimizations on CFGs generated by reverse engineering Itanium executables [17]. Results indicate that de-optimizations allow for reductions in both complexity and size of the generated CFGs.

3. EXPERIMENTAL SETTING

This section covers the details of the compiler framework, benchmarks, and optimizations used for the evaluation of de-optimizing and re-optimizing assembly code. All experiments were carried out on an Intel StrongARM SA-110 processor running Netwinder Linux.

3.1 VISTA Framework

The proposed modifications are added to the ARM port of the VISTA (VPO Interactive System for Tuning Applications) framework [20, 14]. VISTA is an interactive compiler that allows a knowledgeable programmer to finely tune the optimization phase order performed on a given function. Recent VISTA research prioritizes non-interactive automatic tuning of code. VPO (Very Portable Optimizer) is used as the backend to VISTA [2]. Code sequences are represented in VPO as Register Transfer Lists (RTLs) to enhance the portability of the compiler and its associated code-improving transformations.

Using the VISTA GUI, a programmer can tailor the phase sequence based on immediate performance feedback information generated using EASE (Environment for Architecture Study and Experimentation) [6]. EASE provides the ability to instrument assembly files to collect both static information (code size) and dynamic information (instructions executed, memory references). VISTA also provides an intelligent search for good phase orderings through the use of a genetic algorithm that allows the programmer to select which optimization phases are available, the maximum number of phases to perform, as well as the fitness criteria to be used for evaluating sequences. When selecting the fitness criteria, the programmer can choose between optimizing for code size, instructions executed, or a mixture of both. Optimization phase sequences are then tested systematically by the genetic algorithm to locate the most beneficial ordering it can find based on the fitness criteria selected.

3.2 Benchmarks

Due to the proprietary nature of most embedded applications, it is a challenge to find hand-tuned assembly programs that are representative of typical workloads. To the best of our knowledge, there are no currently available embedded benchmark suites written in assembly code. Rather than devising a new set of hand-tuned assembly benchmarks for embedded systems, an optimizing compiler could be used instead with an existing high-level source code benchmark suite. Although using actual hand-tuned assembly would be better, the use of optimization techniques with a known benchmark suite provides a legitimate framework for evaluating the effects of de-optimization when re-optimizing code.

For our experiments, we chose the MiBench embedded applications benchmark suite [8]. These applications are representative of common programs used in embedded systems. The suite consists of programs from six categories. Due to the long run-time of performing a thorough genetic algorithm search, one benchmark from each category is selected for evaluation as shown in Table 1. These are the same benchmarks used in our prior studies [14, 12, 13].

Table 1: Tested MiBench Benchmarks

Category	Program
Automotive/Industrial	bitcount
Network	dijkstra
Telecommunications	fft
Consumer	jpeg
Security	sha
Office	stringsearch

3.3 Optimization with GCC

It is necessary to run each program through an optimizing compiler in order to test the proposed de-optimization strategy. Each of the benchmark programs is optimized using the ARM port of the GNU Compiler Collection's C compiler version 3.3 [7]. The unmodified GCC compiler provides a fair baseline for evaluation. The command line used to compile each of the C source files is as follows:

```
gcc -O2 -S -c -fno-optimize-sibling-calls \
-ffixed-lr -ffixed-fp filename.c
```

Level 2 optimization allows a fair comparison to hand-tuned assembly since it does not invoke additional phases that will contribute to increased space requirements such as loop unrolling. Translating from object code is possible, but adds an additional complexity to the translation process.

The selected *-f* flags disable specific phases of the *-O2* optimization process. Optimizing sibling calls is a transformation that allows leaf or sibling functions to omit save and restore instructions. Allowing GCC to perform these optimizations makes translation more difficult. Additionally VISTA will automatically re-perform this transformation both for the GCC baseline code as well as the experimental code. The two *-ffixed* flags force GCC to disallow the use of these special-purpose registers as general-purpose registers. These flags are necessary to perform a fair comparison since VISTA currently does not support using either the Frame Pointer or the Link Register in a different manner than that for which they were originally intended.

4. ASSEMBLY TRANSLATION

The construction of a translator from native ARM assembly to VISTA RTLs is necessary for studying the effects of de-optimization and re-optimization. This translator is part of a larger suite of ASM2RTL tools, a group of translators in which each converts instructions from a given assembly language to RTLs. The current version of ASM2RTL supports assembly instructions from the Sun Ultra SPARC III, the Texas Instruments TMS320c54x and the Intel StrongARM. Translation appears to be mostly straightforward, but it does contain several potential pitfalls.

Almost every problem that can occur is due to the loss of semantic content needed for correct interpretation. However, there are other potential problems, such as the lack of support from the VPO compiler for a very context-specific instruction (e.g. predicated return). If a compiler cannot produce an instruction, it must be replaced by an equivalent sequence that preserves all aspects of the current program state. As a program progresses through various intermediate representations, each lower-level form carries less information than preceding higher-level forms. Thus it becomes increasingly difficult (if not impossible) to extract the entire original program representation from a low-level form.

4.1 Local Variables

Local variables are those which are allocated on the run-time stack. Modern ISAs support memory accesses of fixed-size increments. The StrongARM supports memory access sizes of 1 byte, 2 bytes, 4 bytes, and 8 bytes. Consequently, data that is larger than these sizes can only be accessed by breaking it into pieces that conform with these sizes. Arrays and structures are typically larger than these fixed sizes and are handled by moving the necessary pieces into and out of registers using these standard instructions. From the low-level assembly representation of a program, it is often difficult to distinguish between accesses to scalar and array data.

This inability to distinguish can be a problem when translating, since the VISTA RTL format (like most intermediate languages) handles local variables symbolically. Original numeric offset information is lost during the translation, allowing the stack ordering of local variables to change. Reordering can cause the code to become incorrect, since local variables larger than 4 bytes may be split apart and spread out on the stack. This is especially true for local structures where certain fields may be manipulated while others are ignored. Calling a function using a pointer to such a split structure causes other local variables to be incorrectly read and written in memory, since the called function mistakenly assumes a different structure layout. The same argument holds for arrays, since they are merely a constrained structure where each field is of the same data type.

To protect against these types of translation errors, additional information concerning the locations and sizes of local variables may be required. ASM2RTL can be supplied with necessary annotations including the actual memory layout of objects on the local stack. Fixing the structure information when translating the function then relies on coalescing known structure members into a single reference with additional offset information for each particular member.

4.2 Calling Conventions

Maintaining calling conventions is another important factor in properly translating assembly code to RTLs. It is necessary to detect incoming register arguments, incoming stack-placed arguments, outgoing register arguments, outgoing stack-placed arguments, as well as register return values in order to properly produce meta-information RTLs. Various analyses could be miscalculated without this information and necessary instructions could be eliminated.

Live register and variable analysis can be used to detect some incoming parameters and outgoing parameters in a function. To perform live register and variable analysis, the entire CFG of the program needs to be constructed. Even at this point, some of the information, such as the size of incoming stack arguments, is still unreliable as it was lost during compilation. Since performing such inter-procedural analysis is time-consuming and may not even yield entirely correct information in these cases, ASM2RTL was set up to strictly perform line-by-line translation. Information about parameters and return values for functions is supplied along with the original input assembly file.

Simple text files are parsed by ASM2RTL for information about function return types and incoming argument sizes in 32-bit words (the smallest unit of allocation for the StrongARM). Function data is split into two files, one for globally accessible functions (library or system calls), and one for application-specific functions. This data is then used during the translation process to reconstruct the appropriate meta-information RTLs. Knowing the return type of a function allows ASM2RTL to generate appropriate RTLs for maintaining this data as live when exiting the function, so that an RTL setting the return value register is not inappropriately deleted as a dead assignment. Local configuration files can be created by either inspecting the assembly code and interpreting the necessary information, or extracted from the original high-level source code (if available). The global function configuration file is easily created using library and system call documentation. ASM2RTL is able to handle variable length argument functions such as *printf* and function pointers using similar annotations.

4.3 Translation Tradeoffs

ASM2RTL adopts various strategies for coping with problems that can affect proper translation. The strategies we have chosen for ASM2RTL are not without drawbacks. They require the programmer to inspect the supplied input assembly code and extract necessary information from it. There is a tradeoff involved since it is possible to assume a worst case scenario for each of the problems. In this way, no additional information is needed from the programmer, but code improvability is sacrificed.

With the local variable layout problem, it can be assumed that all stack elements belong to one large structure. In this case none of the elements are replaceable or reorderable. Doing this will inhibit any further optimizations concerning these variables since arrays and structures are often ignored by the majority of code improving transformations.

There are two requirements for guaranteeing consistency without additional information for calling conventions. The function can be assumed to be using its entire stack for argument space, and thus the stack is not able to be reorganized. This is merely the same requirement as for local variables. Additionally, all argument registers and return registers can be marked as live using the meta-information RTLs in appropriate places. However, this can inhibit many transformations from improving the code. One example of this would be the inability to detect dead assignments to argument or return registers.

5. DE-OPTIMIZATION

This section presents the motivation and implementation details of performing de-optimization on previously optimized assembly code. The de-optimization of both *loop-invariant code motion* and *register allocation* is covered. Additional tradeoffs and difficulties are also described.

5.1 Loop-invariant Code Motion

Loop-invariant code motion (LICM) focuses on moving instructions that do not change the program state out of loop bodies. Instructions that are considered loop-invariant can be moved to the loop preheader, a basic block that precedes the loop header. In addition to attempting to move loop-invariant assignments to loop preheaders, VPO and other optimizing compilers attempt to place any loop-invariant expressions or memory references into registers. These operations can then be performed cheaply in the loop preheader, prior to executing the various iterations of the loop.

LICM requires the use of additional registers in order to provide the greatest benefit. These registers are used to

```
for loop ∈ loops sorted outermost to innermost do
    perform loop_invariant_analysis() on loop
    for rtl ∈ loop→preheader→rtls sorted last to first do
        if rtl is invariant then
            for blk ∈ loop→blocks do
                for trtl ∈ blk→rtls do
                    if trtl uses a register set by rtl then
                        insert a copy of rtl before trtl
        update loop_invariant_analysis() data
```

Figure 2: De-optimize LICM

RTLs Before De-Optimization

#	RTLs	Comments
	...	
a1	+r[10]=R[L44]	# Load LI global
a2	+r[6]=0	# Initialize loop ctr
a3	L11:	# Label L11
a4	+r[2]=r[10]+(r[6]{2)	# Calc array address
a5	+r[5]=r[5]+R[r[2]]	# Add array value
a6	+r[6]=r[6]+1	# Loop ctr increment
a7	+c[0]=r[6]-79:0	# Set CC
a8	+PC=c[0]'0,L11	# Perform loop 80X
	...	

RTLs After De-Optimization

#	RTLs	Comments
	...	
b1	**+r[10]=R[L44]**	# Load LI global
b2	+r[6]=0	# Initialize loop ctr
b3	L11:	# Label L11
b4	**+r[10]=R[L44]**	# LI load (in loop)
b5	+r[2]=**r[10]**+(r[6]{2)	# Calc array address
b6	+r[5]=r[5]+R[r[2]]	# Add array value
b7	+r[6]=r[6]+1	# Loop ctr increment
b8	+c[0]=r[6]-79:0	# Set CC
b9	+PC=c[0]'0,L11	# Perform loop 80X
	...	

Figure 3: De-optimizing LICM

hold values such as loop-invariant variable loads or complex arithmetic calculations that cannot be further simplified using traditional optimizations. Increased register pressure inhibits additional code-improving transformations from being as beneficial as possible. Undoing LICM provides VISTA with the possibility of applying additional code-improving transformations before potentially re-applying LICM.

The algorithm for performing de-optimization of LICM is shown in Figure 2. It attempts to place all loop-invariant RTLs before RTLs where a register they set is used. The algorithm operates on loops sorted from outermost to innermost, moving invariant RTLs as far inward as possible in the CFG.

The top of Figure 3 depicts a group of RTLs corresponding to a loop that has had LICM performed on it. The load of a global variable containing the starting address of an array (Line a1) is a loop-invariant instruction that was moved out of the loop. The loop (Lines a3-a8) is performed 80 times using an induction variable set prior to beginning the loop (Line a2). The loop-invariant register `r[10]` is used as part of an address calculation with the loop counter `r[6]`.

After de-optimizing LICM, the code shown in the bottom portion of Figure 3 is obtained. Notice that the set of register `r[10]` has been added before its use (Line b5). Previously this set existed only in the loop preheader (Line a1/b1). Notice that the set of `r[6]` in the preheader was not able to be moved into the loop (Line a2/b2). This is because the register `r[6]` is not loop invariant, as evidenced by its set within the loop (Line a6/b7).

5.2 Register Allocation and Assignment

Register allocation (ALLOC) is a code-improving transformation that attempts to place local variable live ranges into registers because memory accesses are more expensive than register accesses. Moving local variable live ranges into registers also enables additional code-improving transformations such as *instruction selection* and *common subexpression elimination*, which are more effective with register expressions. This process consumes available registers as any conflicts with existing register live ranges must be avoided.

ALLOC is traditionally treated as a graph coloring problem, which is defined as finding the minimum number of colors needed to color all vertices in a graph such that no two connected vertices share the same color. Graph coloring is NP-complete, and as such an optimal solution is computationally expensive. Thus approximation algorithms are used instead. The application of graph coloring to ALLOC treats registers as colors and live ranges as vertices. Live ranges that overlap or conflict are connected by edges in the graph.

The graph used for performing ALLOC is called an interference graph. Since the number of registers available to the compiler is finite, the entire graph may not be color-able. It is also true that the allocation of non-scratch registers can incur additional costs due to the necessity of saving and restoring their values on function entry and exit. Critical choices are made by the compiler as to which live ranges should be allocated and which potentially should never be allocated even if registers are available. Priority-based coloring is an approach that attempts to weight live ranges according to various heuristics so that a good solution can be obtained in a relatively short amount of time [3]. Similar approaches have been adopted by both GCC and VPO.

VPO typically receives RTL input from a frontend that does not make choices as to which registers should be used. Instead the RTLs refer to pseudo-registers which do not actually exist. Certain hardware registers are specified for function call and return semantics. *Register assignment* (ASSIGN) is the process by which pseudo-register references are converted to actual hardware registers. ASSIGN is similar to ALLOC in many ways, since conflicts of register live ranges must be avoided. Pseudo-registers that cannot be assigned must have appropriate spill code generated.

The undoing of ALLOC and ASSIGN will allow for fewer registers to be used in various code sequences, freeing up a greater number of registers for other transformations. It is possible that the choices made during the initial run of ASSIGN require additional stores and loads to preserve scratch registers around function calls. Changing the assignment could alleviate the need for such spill code entirely.

In order to de-optimize ALLOC and ASSIGN, we construct a register interference graph (RIG). The RIG is similar to the standard interference graph, except that nodes correspond to register live ranges, and not variable live ranges. The process of constructing a RIG analyzes basic blocks first, then later connects corresponding incoming and outgoing register live ranges as sibling nodes in the graph. Hardware-specified registers for argument usage and return values cannot be replaced, and such nodes are appropri-

```
1  calculate live variable information
2  calculate dead register information
3  RIG = construct_register_interference_graph()
4  mark all nodes in RIG as not done
5  for node ∈ RIG→nodes do
6      if ¬node→done ∧ node→can_replace then
7          node→done = TRUE
8          if node is an intrablock live range then
9              node→pseudo = new_pseudoregister()

10         else
11             # node is an interblock live range
12             node→local = new_local_variable()
13             node→pseudo = new_pseudoregister()
14             update siblings (local/pseudo) and mark them done

15 recalculate necessary analysis for pseudo-registers in VPO
16 mark all nodes in RIG as not done
17 for node ∈ RIG→nodes do
18     if ¬node→done ∧ node→can_replace then
19         node→done = TRUE
20         if node is an intrablock live range then
21             for ref ∈ node→sets ∪ node→uses do
22                 replace ref with node→pseudo

23         else
24             # node is an interblock live range
25             for sib ∈ node ∪ node→sibs do
26                 for use ∈ sib→uses do
27                     load node→local in node→pseudo before use
28                     replace use with node→pseudo
29                 for set ∈ sib→sets do
30                     replace set with node→pseudo
31                     store node→pseudo in node→local after set

32 re-perform register_assignment() to assign all pseudo-registers
33 perform instruction_selection() to clean up code
```

Figure 4: De-optimize ALLOC

ately marked during construction. Further details on the construction and use of RIGs can be found in [10].

Figure 4 shows the algorithm employed for de-optimizing ALLOC and ASSIGN. Nodes with no siblings are referred to as intrablock live ranges, meaning that they do not span basic block boundaries. Nodes with one or more siblings are referred to as interblock live ranges. These nodes do span basic block boundaries, and are the primary target.

In the first pass, each intrablock live range is assigned a new pseudo-register, while each interblock live range is assigned a new pseudo-register as well as a new local variable of appropriate type (Lines 5-14). The second pass actually performs the de-optimization. Both intrablock and interblock live ranges have register references replaced with their new pseudo-registers. Additionally, interblock live ranges have any set of their pseudo-register followed immediately by a store of that pseudo-register to the new local variable memory location. Any use of an interblock pseudo-register must first have that pseudo-register value loaded from the appropriate local variable memory location. Finally, we re-perform ASSIGN, such that the minimal number of hardware registers are employed for function correctness.

Figures 5–8 depict an example showing the benefit of de-optimizing ALLOC and ASSIGN. Figure 5 shows the original optimized RTLs corresponding to a dequeue routine. Horizontal lines show the basic block divisions, as viewed by VISTA. This routine takes three arguments from registers `r[0]`, `r[1]`, and `r[2]`, and saves them for later use.

#	RTLs	Deads	Comments
1	`r[6]=R[L21];`		
2	`r[12]=R[r[6]+0];`		
3	`r[3]=R[L21+4];`		
4	`c[0]=r[12]-0:0;`		`# NULL ptr?`
5	`R[r[3]+0]=r[12];`	`r[3]`	
6	`r[4]=`**`r[1]`**`;`	**`r[1]`**	`# Saving` **`r[1]`**
7	`r[3]=`**`r[0]`**`;`	**`r[0]`**	`# Saving` **`r[0]`**
8	`r[5]=`**`r[2]`**`;`	**`r[2]`**	`# Saving` **`r[2]`**
9	`r[0]=r[12];`		
10	`PC=c[0]:0,L0001;`	`c[0]`	`# beqz L0001`
11	`r[2]=R[r[12]+0];`		
12	`R[r[3]+0]=r[2];`	`r[2]r[3]`	
13	`r[3]=R[r[12]+4];`		
14	`R[r[4]+0]=r[3];`	`r[3]r[4]`	
15	`r[2]=R[r[12]+8];`		
16	`r[1]=R[r[12]+12];`	`r[12]`	
17	`R[r[5]+0]=r[2];`	`r[2]r[5]`	
18	`R[r[6]+0]=r[1];`	`r[1]r[6]`	
19	`ST=free; =`**`r[0]`**`;`		`# free(`**`r[0]`**`)`
20	`r[2]=R[L21+8];`		
21	`r[3]=R[r[2]+0];`		
22	`r[3]=r[3]-1;`		
23	`R[r[2]+0]=r[3];`	`r[2]r[3]`	
24	`PC=RT;`		`# Return`
25	`L0001:`		`# Label`
26	`PC=RT;`		`# Return`

Figure 5: Dequeue prior to De-optimizing ALLOC

Figure 6 shows the code for the first basic block after de-optimizing ALLOC and ASSIGN. Each register that is not hardware-specific is remapped as a pseudo-register. Additional loads and stores of the newly added local variables are represented with each particular RTL line using letter suffixes with the appropriate line number. ASSIGN has not been re-performed at this point in time.

Figure 7 shows the code after re-performing ASSIGN. This code uses a minimal set of registers for all of the appropriate operations. Many operations can reuse the same hardware register (`r[12]`), since it is only needed for very short live ranges. When live ranges overlap, additional registers are selected. There is no possibility of exhausting the register supply, since the original code was able to map completely to hardware registers, and we have only shortened live ranges by using temporary local variables with de-optimization.

Figure 8 shows the RTLs after performing further optimizations including *dead code elimination*, *strength reduction*, *instruction selection*, *register allocation*, *dead variable elimination*, *common subexpression elimination*, and *fix entry exit*. All de-optimization added local variable references are eliminated. Other optimization phases now have additional opportunities to improve the code layout, resulting in the removal of two register move operations that were previously used to save incoming arguments (Lines 6,8).

5.3 Difficulties with De-optimization

De-optimization of optimized code is complicated by the same problems with calling conventions that affected assembly translation. Incoming register arguments, outgoing return values, local array/structure layouts and hardware-specific registers can all limit the effectiveness of the pro-

#	RTLs	Deads	Comments
1a	**r[32]**=R[L21];		# **r[6]** → **r[32]**
1b	R[r[13]+**__dequeue_0**]=**r[32]**;	**r[32]**	# Store pseudo **r[32]**
2a	r[32]=R[r[13]+__dequeue_0];		# Load pseudo **r[32]**
2b	r[33]=R[r[32]+0];	r[32]	# Perform actual op
2c	R[r[13]+__dequeue_1]=r[33];	r[33]	# Store pseudo **r[33]**
3	**r[34]**=R[L21+4];		# Intrablock live range
			# Use pseudo **r[34]**
4a	r[33]=R[r[13]+__dequeue_1];		
4b	**c[0]**=r[33]-0:0;	r[33]	# **c[0]** not replaceable
5a	r[33]=R[r[13]+__dequeue_1];		
5b	R[r[34]+0]=r[33];	r[33]**r[34]**	# Intrablock **r[34]** dies
6a	r[35]=r[1];	r[1]	# Incoming argument **r[1]**
6b	R[r[13]+__dequeue_2]=r[35];	r[35]	# is not replaceable
7a	r[36]=r[0];	r[0]	# Incoming argument **r[0]**
7b	R[r[13]+__dequeue_3]=r[36];	r[36]	# is not replaceable
8a	r[37]=r[2];	r[2]	# Incoming argument **r[2]**
8b	R[r[13]+__dequeue_4]=r[37];	r[37]	# is not replaceable
9a	r[33]=R[r[13]+__dequeue_1];		# **r[0]** is outgoing
9b	**r[0]**=r[33];	r[33]	# argument to free()
10	PC=c[0]:0,L0001;	c[0]	# Branch uses only **c[0]**
	...		# so no replacements

Figure 6: Dequeue after De-optimization of ALLOC

#	RTLs	Deads	Comments
1a	r[12]=R[L21];		# **r[12]** is first non-arg
1b	R[r[13]+__dequeue_0]=r[12];	r[12]	# scratch register
2a	**r[12]**=R[r[13]+__dequeue_0];		# Note use of **r[12]** to
2b	**r[12]**=R[**r[12]**+0];		# combine 2 distinct live
2c	R[r[13]+__dequeue_1]=**r[12]**;	**r[12]**	# ranges in these lines
3	r[12]=R[L21+4];		
4a	**r[3]**=R[r[13]+__dequeue_1];		# First appearance of
4b	c[0]=**r[3]**-0:0;	**r[3]**	# **r[3]** since there are
			# currently 2 live ranges
5a	r[3]=R[r[13]+__dequeue_1];		
5b	R[r[12]+0]=r[3];	r[3]r[12]	
6a	r[12]=r[1];	r[1]	# Save argument **r[1]**
6b	R[r[13]+__dequeue_2]=r[12];	r[12]	
7a	r[12]=r[0];	r[0]	# Save argument **r[0]**
7b	R[r[13]+__dequeue_3]=r[12];	r[12]	
8a	r[12]=r[2];	r[2]	# Save argument **r[2]**
8b	R[r[13]+__dequeue_4]=r[12];	r[12]	
9a	r[12]=R[r[13]+__dequeue_1];		
9b	r[0]=r[12];	r[12]	
10	PC=c[0]:0,L0001;	c[0]	# Live regs leaving block
	...		# are **r[0]** and **r[13]** (SP)

Figure 7: Dequeue after Re-performing ASSIGN

posed de-optimizations. However, such restrictions are necessary to retain inter-operability between function modules.

Additionally, the ARM uses PC-relative 12-bit offsets for global address references. These de-optimization algorithms expand code, causing some of these references to fall out of reach of their original symbolic targets. This results in the assembler not being able to generate appropriate instructions when the offset is out of reach. In order to prevent such expansion problems during the re-optimization process, code containing out of reach symbolic offsets caused VISTA to reject the corresponding phase ordering. The resulting code would be eliminated as a poor solution in any case.

6. RESULTS

This section presents the results of running experiments comparing de-optimization and re-optimization strategies with previously tuned code. First, the GCC-optimized code for each benchmark is translated to the RTL format and both static and dynamic counts are collected as baseline measures after performing a simple compilation pass with VISTA. This pass includes performing *instruction selection* and *predication* during the *fix entry exit* phase. The GCC-optimized baseline code can potentially obtain a slight benefit, since VISTA may detect additional sequences that are able to be predicated, as well as other instruction sequences that can be combined into fewer RTLs. Each benchmark program is instrumented during compilation using the EASE framework to obtain both static code size and dynamic instruction execution counts. Similar measures were obtained in previous studies for selecting phase sequences [5, 14].

Table 2 shows the results of running the experiments for the StrongARM architecture. Each of the six tested benchmarks (*bitcount*, *dijkstra*, *fft*, *jpeg*, *sha*, and *stringsearch*) is presented individually. The field labeled *Compiler Strategy* denotes whether just the genetic algorithm search is performed (`Pure Re-opt`) or if de-optimization phases are enabled prior to executing the genetic algorithm search (`De-opt + Re-opt`). When de-optimization is not bene-

#	RTLs	Deads	Comments
1	r[5]=R[L21];		
2	r[4]=R[r[5]];		
3	r[12]=R[L21+4];		
4	c[0]=r[4]:0;		
5	R[r[12]]=r[4];	r[12]	
6			# RTL **r[4]=r[1]** now unnecessary
7	r[8]=r[0];	r[0]	
8			# RTL **r[5]=r[2]** now unnecessary
9	r[0]=r[4];		
10	PC=c[0]:0,L0001;	c[0]	
11	r[12]=R[r[4]];		
12	R[r[8]]=r[12];	r[8]r[12]	
13	r[12]=R[r[4]+4];		
14	R[**r[1]**]=r[12];	**r[1]**r[12]	# **r[1]** live until here now
15	r[12]=R[r[4]+8];		
16	r[1]=R[r[4]+12];	r[4]	
17	R[**r[2]**]=r[12];	**r[2]**r[12]	# **r[2]** live until here now
18	R[r[5]]=r[1];	r[1]r[5]	
19	ST=free; =r[0];		
20	r[12]=R[L21+8];		
21	r[1]=R[r[12]];		
22	r[1]=r[1]-1;		
23	R[r[12]]=r[1];	r[1]r[12]	
24	PC=RT;		
25	L0001:		
26	PC=RT;		

Figure 8: Dequeue after Additional Optimizations

ficial, we use the result of performing re-optimization only. Measurements are taken using three different fitness criteria for VISTA's genetic algorithm search for effective phase sequences, varying the weight of potential tradeoffs such as code size and dynamic instruction count. The test criteria for these experiments includes optimizing for static code size (`Optimize for Space`), optimizing for dynamic instruction count (`Optimize for Speed`), and optimizing for a combination of the two, weighting each equally (`Optimize for Both`). The initial translated RTLs from the GCC-optimized code are used as a baseline measure to which all tested configurations for de-optimization and genetic algorithm searching can be compared. Results are calculated by comparing experimental static and dynamic instruction counts to the initial static and dynamic instruction counts for the GCC-optimized code for each benchmark. Improvements are expressed in the table as percentages.

Results show that performing de-optimizations before re-optimizing allows for some potential benefits. In both the *dijkstra* and *stringsearch* benchmarks, de-optimizing provides benefits for each of the three fitness criteria tested. De-optimizations are also beneficial when optimizing for space for both the *fft* and *jpeg* benchmarks. In the cases of *bitcount* and *sha*, de-optimizing provides no additional benefit over pure re-optimization.

Overall results show that by performing re-optimizations alone, VISTA is successful in decreasing static code size by an average of 2.50% and dynamic instructions by an average of 3.28% across all benchmarks when compared to the original GCC-optimized code. De-optimizing before re-optimizing yields even greater success in decreasing static code size with an average of 3.18%, winning against pure re-optimization in 4 out of 6 cases. Optimizing for speed with de-optimization provides no additional benefit over pure re-optimization. In all cases, re-optimization with or without de-optimization yields improvements to the original GCC-optimized code. A closer look shows that no primary or secondary fitness measure performs worse than the baseline GCC-optimized code for an entire benchmark although some functions did experience small decreases later canceled out by results from improved functions.

One confusing aspect of Table 2 is the comparison of dynamic counts for the *fft* benchmark when looking at code that is de-optimized. For the mixed static/dynamic and dynamic search criteria, there are no additional improvements, yet the static search results in a slight improvement in dynamic instruction count (0.35%). Other portions of the table exhibit similar results. This is explainable by the inherent randomness of evolutionary algorithms, particularly, the genetic algorithm search that VISTA employs in creating and evaluating different phase orderings. In each particular such case, the better sequences for a particular fitness criteria are just not uncovered during the search process.

7. CONCLUSIONS AND FUTURE WORK

Embedded systems development is dominated by requirements which can include rigid constraints on code size, power consumption, and time. It is common for applications to be written in assembly and hand-tuned to meet these expectations. Yet, just as traditional optimizing compilers are subject to the phase ordering problem, hand-tuned assembly code can experience an analogous problem depending on programmer choices during the tuning process. The undoing of prior optimizations, or de-optimization of assembly code is one potential method of alleviating the negative ef-

Table 2: Effect of De-optimization on Static and Dynamic Instruction Count

Benchmark	Compiler Strategy	Optimize for Space		Optimize for Speed		Optimize for Both		
		static count	dynamic count	static count	dynamic count	static count	dynamic count	average
bitcount	Pure Re-opt	2.32 %	0.00 %	2.32 %	0.00 %	2.32 %	0.00 %	1.16 %
	De-opt + Re-opt	2.32 %	0.00 %	2.32 %	0.00 %	2.32 %	0.00 %	1.16 %
dijkstra	Pure Re-opt	1.30 %	2.70 %	1.30 %	2.70 %	1.30 %	2.70 %	2.00 %
	De-opt + Re-opt	2.16 %	2.73 %	3.03 %	2.73 %	3.03 %	2.73 %	2.88 %
fft	Pure Re-opt	0.19 %	0.00 %	0.19 %	0.00 %	0.19 %	0.00 %	0.09 %
	De-opt + Re-opt	0.19 %	0.35 %	0.19 %	0.00 %	0.19 %	0.00 %	0.09 %
jpeg	Pure Re-opt	4.30 %	10.61 %	4.30 %	10.61 %	4.30 %	10.61 %	7.46 %
	De-opt + Re-opt	5.20 %	10.53 %	4.30 %	10.61 %	4.30 %	10.61 %	7.46 %
sha	Pure Re-opt	5.99 %	4.39 %	3.89 %	6.27 %	5.99 %	4.39 %	5.19 %
	De-opt + Re-opt	5.99 %	4.39 %	3.89 %	6.27 %	5.99 %	4.39 %	5.19 %
stringsearch	Pure Re-opt	0.92 %	0.09 %	0.92 %	0.09 %	0.92 %	0.09 %	0.51 %
	De-opt + Re-opt	3.23 %	0.09 %	3.23 %	0.09 %	3.23 %	0.09 %	1.66 %
average	Pure Re-opt	2.50 %	2.97 %	2.15 %	3.28 %	2.50 %	2.97 %	2.73 %
	De-opt + Re-opt	3.18 %	3.01 %	2.83 %	3.28 %	3.18 %	2.97 %	3.07 %

fects of the phase ordering problem in such cases. In this paper, we presented an extension of VISTA for performing de-optimization and re-optimization of tuned assembly code.

De-optimizations that reduce register pressure, such as *loop-invariant code motion* and *register allocation* were evaluated, since registers are typically a limited resource in embedded systems. Results showed that de-optimization could be very beneficial in the re-optimization process, although it can potentially be detrimental as well. Overall, we have shown that de-optimization can provide an additional opportunity for reordering optimization phases that may have already been performed on previously generated assembly code. It is possible to extend the de-optimization process with a greater variety of optimizations to undo. *Common subexpression elimination* is another phase that increases register pressure, so de-optimizations developed for it may prove beneficial as well.

De-optimization and re-optimization of GCC-optimized code using the genetic algorithm search features of VISTA provided decreases in static code size on average of 3.18% and decreases in dynamic instruction count on average of 3.28% when compiling for each individually. Hand-tuned assembly benchmarks can be further examined using this framework to evaluate the effectiveness of de-optimization and re-optimization. In the embedded devices arena, more complex and longer optimization processes are acceptable since a large number of units are typically produced and code requirements may be more stringent than with traditional software. The development of tools such as the ASM2RTL translator suite and VISTA with de-optimization opens up new possibilities for the further optimization of code, particularly for hand-tuned assembly and legacy applications.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-020889, ACI-0305144, and CCR-0312493.

9. REFERENCES

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

[2] M. E. Benitez and J. W. Davidson. Target-specific global code improvement: Principles and applications. Technical Report CS-94-42, 4, 1994.

[3] F. C. Chow and J. L. Hennessey. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, June 1984.

[4] C. Cifuentes, M. Van Emmerik, B. T. Lewis, and N. Ramsey. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical Report TR-2002-105, Sun Microsystems Laboratories, January 2002.

[5] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9. ACM Press, 1999.

[6] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[7] Free Software Foundation. GNU compiler collection 3.3. http://gcc.gnu.org/, 2004.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[9] J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[10] S. Hines. Using de-optimization to re-optimize code. Master's thesis, Florida State University, Tallahassee, Florida, April 2004.

[11] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992.

[12] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182, 2004.

[13] P. Kulkarni, S. Hines, D. Whalley, J. Hiser, J. Davidson, and D. Jones. Fast and efficient searches for effective optimization phase sequences. *Transactions on Architecture and Code Optimization*, pages 165–198, June 2005.

[14] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 12–23, 2003.

[15] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[16] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. ALTO: A link-time optimizer for the Compaq Alpha. *Software - Practice and Experience*, 31:67–101, January 2001.

[17] N. Snavely, S. Debray, and G. Andrews. Unscheduling, unpredication, unspeculation: Reverse engineering Itanium executables. In *Proceedings of the 2003 Working Conference on Reverse Engineering*, pages 4–13, November 2003.

[18] S. R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the fifteenth annual workshop on microprogramming*, pages 125–133, 1982.

[19] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, 1997.

[20] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones. VISTA: A system for interactive code improvement. In *Proceedings of the joint conference on Languages, Compilers, and Tools for Embedded Systems*, pages 155–164. ACM Press, 2002.