Using De-optimization to Re-optimize Code

Stephen Hines[1], Prasad Kulkarni[1],
David Whalley[1], Jack Davidson[2]

Computer Science Dept.[1]
Florida State University

Computer Science Dept.[2]
University of Virginia

September 20, 2005

❶ Introduction

- **Phase Ordering Problem**
  - No sequence of optimization phases will produce optimal code for all functions in all applications on all architectures
  - Long standing problem for compiler writers
  - **Register pressure** is a critical factor
- **Embedded Systems Development**
  - Greater tolerance for longer, more complex compile processes
    - ⋆ Large number of devices produced → even small savings add up
    - ⋆ Tighter constraints (code size, power, real-time)
    - ⋆ Fewer registers and features than modern CPUs
    - ⋆ Hand-tuned assembly code can suffer from an analogous problem to phase ordering

◆ Reducing Phase Ordering Effects

- Methods to Diminish Problems with Phase Ordering
  - Iteration of optimization phases (VPO)
  - Test combinations of optimization phases for best sequence (VISTA)
- Problems with Current Methodology
  - Current solutions work with higher-level languages (not assembly)
  - Not able to take into account previously applied optimizations, due to hand-tuning or another compiler (e.g. no spare registers for allocation)

◆ Proposed Approach

- **Translate** assembly code back to intermediate languages for input to an optimizer.
- **Undo** the effects of various optimization phases to allow for different phase ordering decisions (**De-optimization**).
- **Re-optimize** the code using new phase orderings to improve performance.

◆ Outline

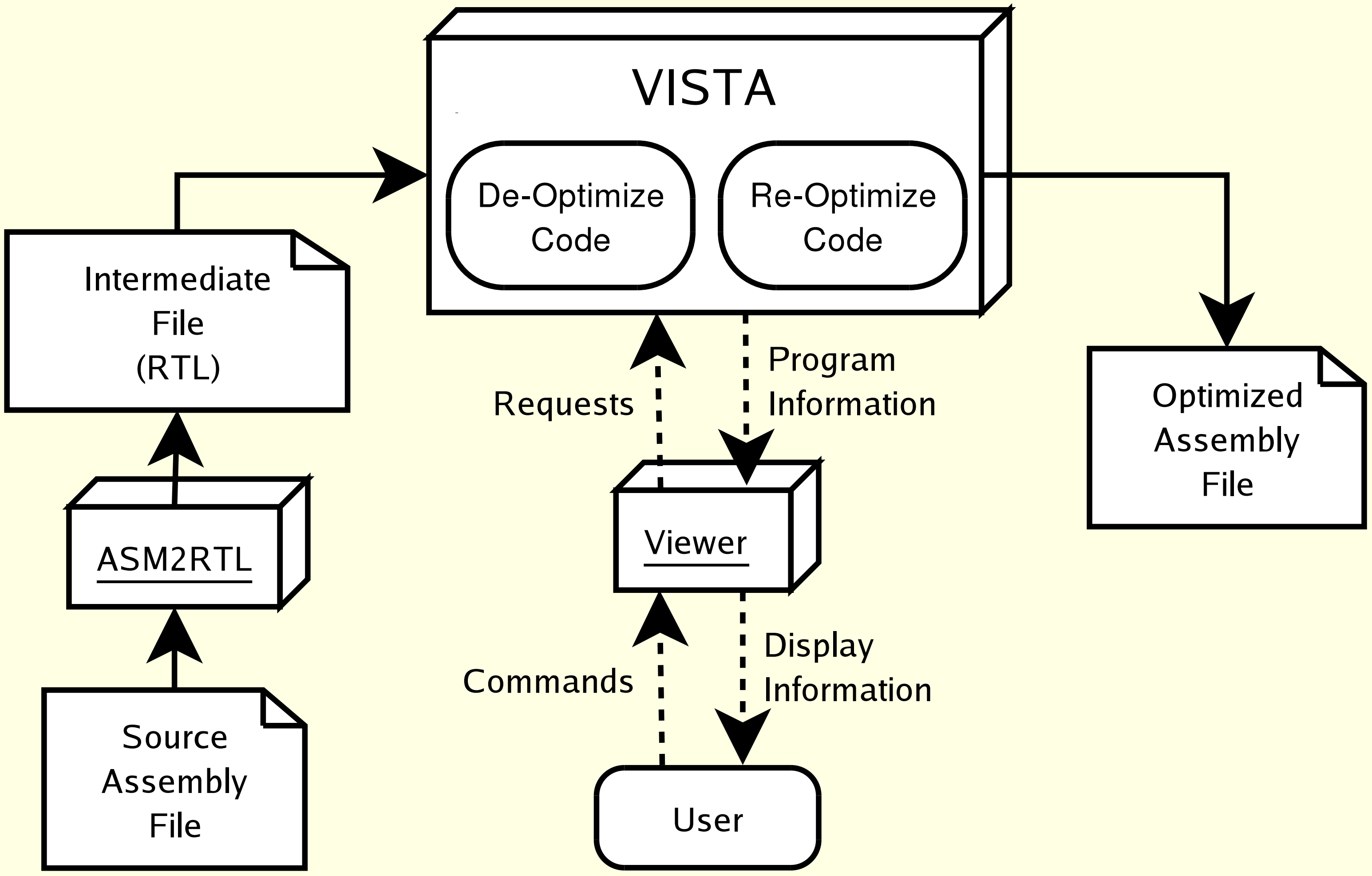
❷ Related Work

- Binary translation
  - Executable Editing Library (EEL)
  - University of Queensland Binary Translator (UQBT)
- Link-time optimizations – ALTO
- De-optimization
  - Debugging optimized executables
  - Reverse engineering

❸ VISTA Framework

- **V**PO **I**nteractive **S**ystem for **T**uning **A**pplications
- Graphical viewer connected to VPO (Very Portable Optimizer) backend
- Interactive approach to tuning code (arbitrary phase orderings permitted, along with hand modification of code)
- Transformations performed on **RTLs** (Register Transfer Lists) – machine-independent representations of instruction semantics
- Automatic tuning of code via a genetic algorithm search for effective phase sequences

◆ Overview of Modified Framework



❹ Assembly Translation

- Converting optimized assembly code to VISTA intermediate language (RTLs)
- Preserving semantics
  - Information Loss – high-level languages have more semantic content than low-level representations
  - Local Variable Confusion – local stack variable start and end points, as well as actual data types
  - Maintaining Calling Conventions – recognizing function parameters and return values

◆ Implementation Strategy

- **ASM2RTL** – translate assembly code $\longrightarrow$ VISTA RTL format
- Split into machine-dependent and machine-independent portions:
  - Sun SPARC
  - Texas Instruments TMS320c54x
  - Intel StrongARM $\longleftarrow$ used for these experiments
- Translate each line individually and perform a pass to patch things up.
- VISTA reconstructs additional information from contextual clues.
- Simplify problems with memory consistency and calling conventions.

◆ Memory Consistency

- VISTA reorganizes local variables during **Fix Entry Exit**
- Cannot allow splitting of arrays, structures or large data types $\longrightarrow$ other functions will not be able to interface with them
- Fixed by supplying translator with annotations regarding functions and corresponding stack information for local structures and arrays

◆ Following Calling Conventions

- VISTA can reconstruct some but not all information regarding registers and stack locations used for special purposes (e.g. arguments, return values):
  - No mechanism for knowing how many registers are used as arguments and thus need to be preserved across a call
  - No distinguishing between stack local variables and arguments
- Knowing the number of parameters and return types of each function (signature), we can recreate the proper environment.
- Variable length argument functions are pre-processed with a tool to detect actual arguments used.
- Function pointers are handled conservatively.

◆ Translation Tradeoffs

- Could assume worst case scenarios and not require annotations
  - Stack layout $\rightarrow$ one large array/structure that is unable to be split
    - ⋆ Most optimizations ignore arrays/structures since they are difficult to manipulate while guaranteeing correctness.
    - ⋆ Decreases chance that re-optimization will be beneficial
  - All argument registers and all stack locations may be parameters.
    - ⋆ Stack variables are already unable to be adjusted (as above).
    - ⋆ Optimizations such as **Dead Assignment Elimination** will be less effective since we will have undetectable dead registers.
- Luckily, a simple code inspection is usually all that is needed to extract the necessary information.

❺ De-optimization

- **Undo** the effects of previous transformations on the code.
- Enable VISTA to reapply those phases in a potentially different order.
- Focus on optimizations that are likely to affect **register pressure**:
  - **Loop-invariant Code Motion**
  - **Register Allocation**

◆ Loop-invariant Code Motion

- Attempts to decrease unnecessary computations by moving RTLs that are not loop-dependent to the **loop preheader**
- Loops are handled from most deeply nested to least deeply nested
- For an RTL/instruction to be considered loop-invariant:

  ① All source operands must be loop-invariant
  ② Must dominate all loop exits
  ③ No set register can be set by another RTL in the loop
  ④ No set register can be used prior to being set by this RTL

◆ De-optimizing LICM

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

◆ Performing the De-optimization

Comments	RTLs Before
	...
Load LI global	+r[10]=R[L44]
Init loop ctr	+r[6]=0
Label L11	L11:
Calc array address	+r[2]=r[10]+(r[6]{2)
Add array value	+r[5]=r[5]+R[r[2]]
Loop ctr increment	+r[6]=r[6]+1
Set CC	+c[0]=r[6]-79:0
Perform loop 80X	+PC=c[0]'0,L11
	...

◆ Performing the De-optimization

Comments	RTLs Before	RTLs After
	...	...
Load LI global	+r[10]=R[L44]	+r[10]=R[L44]
Init loop ctr	+r[6]=0	+r[6]=0
Label L11	L11:	L11:
Load LI global		**+r[10]=R[L44]**
Calc array address	+r[2]=r[10]+(r[6]{2)	+r[2]=r[10]+(r[6]{2)
Add array value	+r[5]=r[5]+R[r[2]]	+r[5]=r[5]+R[r[2]]
Loop ctr increment	+r[6]=r[6]+1	+r[6]=r[6]+1
Set CC	+c[0]=r[6]-79:0	+c[0]=r[6]-79:0
Perform loop 80X	+PC=c[0]'0,L11	+PC=c[0]'0,L11
	...	...

◆ Register Allocation

- Attempts to place local variables live ranges into registers $\longrightarrow$ save on memory access overhead costs
- Traditionally treated as a graph coloring problem, which is NP-complete
- Register allocation algorithms work with **interference graphs**
  - Vertices $\longleftarrow$ variable live ranges
  - Edges $\longleftarrow$ connect live ranges that overlap or conflict
  - Colors $\longleftarrow$ available registers
- **Priority-based coloring** weights live ranges according to various heuristics to find a good solution if graph cannot be completely colored

◆ De-optimizing Register Allocation

- Construct a **register interference graph** (RIG)
- Replace register live ranges from RIG depending on their span
  - Intrablock live ranges just get remapped to pseudo-registers
  - Interblock live ranges get remapped to pseudo-registers as well as a new local variable for storage
- Insert stores of new local variables after sets of these registers
- Insert loads of new local variables before uses of these registers

◆ Prior to De-optimization

#	RTLs	Deads
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;	
5	R[r[3]+0]=r[12];	r[3]
6	r[4]=**r[1]**;	**r[1]**
7	r[3]=**r[0]**;	**r[0]**
8	r[5]=**r[2]**;	**r[2]**
9	r[0]=r[12];	
10	PC=c[0]:0,L0001;	c[0]
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];	

#	RTLs	Deads
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]**;	
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;	
25	L0001:	
26	PC=RT;	

◆ After De-optimizing Register Allocation

#	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

◆ After Register Re-assignment

#	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)

◆ After Re-optimization

#	RTLs	Deads
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	~~**r[4]=r[1]**~~	
7	r[8]=r[0];	r[0]
8	~~**r[5]=r[2]**~~	
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];	

#	RTLs	Deads
14	R[**r[1]**]=r[12];	**r[1]**r[12]
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]
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;	

❻ Experimental Results

- Hand-tuned assembly code is usually proprietary, so we will focus on optimized C code from a different compiler:
  - O2 optimized benchmarks with GCC 3.3 for the ARM
  - MiBench: bitcount, dijkstra, fft, jpeg, sha, stringsearch

- Evaluate benefit of re-optimization using VISTA's genetic algorithm search against de-optimization plus re-optimization

- Fitness criteria tested include static code size, dynamic instruction count and a 50%/50% mix

◆ Benefit of De-optimization

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

❼ Conclusions & Future Work

- Embedded applications have rigid constraints regarding code size, power consumption and/or execution time.
- De-optimization can be used to **roll back** some existing phase orderings from other compilers/hand tuning.
- Improvements can be achieved when combining de-optimization with a compiler framework that can evaluate multiple phase orderings using a genetic algorithm search.
- Further de-optimizations can be developed for phases that impact register pressure, including common subexpression elimination.
- ASM2RTL + de-optimizations may also prove beneficial when working with actual hand-tuned assembly code.

◆ The End

Thank you!

Questions ???

See us for a demo of VISTA!