

USING DE-OPTIMIZATION TO RE-OPTIMIZE CODE

STEPHEN R. HINES

APRIL 30, 2004

COMMITTEE:

- DAVID WHALLEY
- GARY TYSON
- XIN YUAN

◆ OUTLINE OF PRESENTATION

- ① Background and Brief Motivation
- ② VISTA Framework
- ③ Assembly Translation
- ④ De-optimization
- ⑤ Experimental Plan and Results
- ⑥ Related Work
- ⑦ Additional Benefits
- ⑧ Future Work
- ⑨ Conclusions

1 BACKGROUND AND BRIEF MOTIVATION

- **Phase Ordering Problem**

- There is no sequence of optimization phases that 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
- Interaction between various phases is hard to characterize

- **Embedded Systems Development**

- Increasing in popularity and complexity
- Greater tolerance for longer and more complex compile processes
 - ★ Large number of devices produced → even small savings add up
 - ★ Tighter constraints (code size, power, real-time)
- Fewer registers and other fancier features from modern CPUs
- Hand-tuned assembly code can suffer from an analogous problem to phase ordering

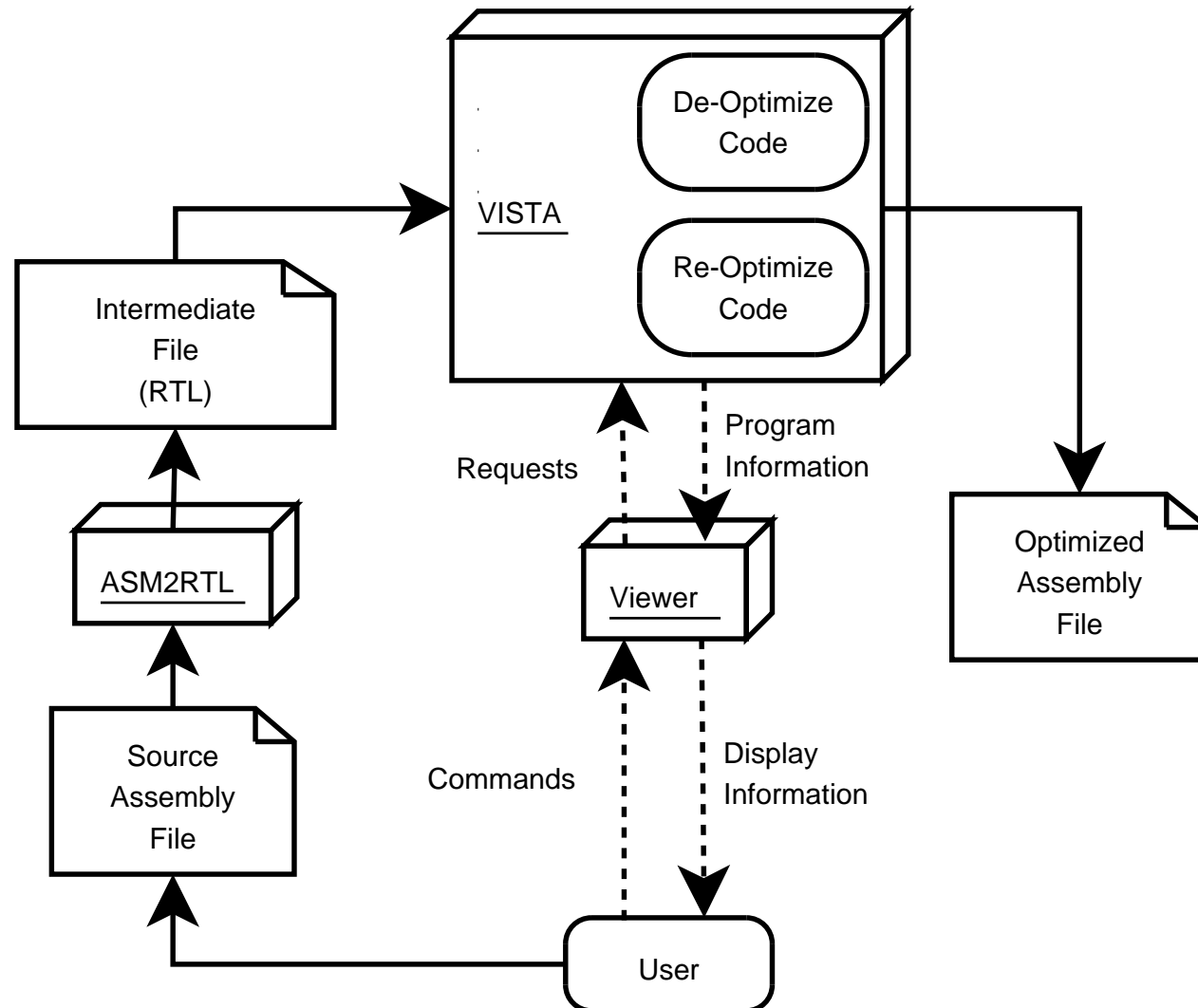
◆ REDUCING PHASE ORDERING EFFECTS

- Methods to Diminish Problems With Phase Ordering
 - Iteration of optimization phases (VPO)
 - Test combinations of optimization phases for best sequence (VISTA)
- Problems With Current Methods
 - Current solutions work with higher level languages (not assembly)
 - Not able to take into account previously performed 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 obtain better performance

◆ OVERVIEW OF MODIFIED FRAMEWORK



② VISTA FRAMEWORK

- **V**P**O** Interactive **S**ystem for **T**uning **A**pplications
- Graphical viewer connected to VPO (Very Portable Optimizer) backend
- Interactive approach to tuning code
- Support for hand-tuning of assembly code
- Extraction of static and dynamic performance information using EASE (Environment for Architecture Study and Experimentation)
- Automatic tuning of code via a genetic algorithm search for effective phase sequences

◆ VPO - VERY PORTABLE OPTIMIZER

- Operates on **RTLs** (Register Transfer Lists), which are machine-independent representations of the effects of machine instructions
- Optimization phases are able to be iterated since representation for all optimizations is always the RTL format
- Various analyses are performed as needed on the RTL representation between phases

◆ SEARCHING FOR EFFECTIVE PHASE SEQUENCES

- Evolutionary algorithms are most effective when:
 - Dealing with large search spaces
 - Interactions between parameters are not well understood
- The phase ordering problem fits both of these criteria, so a genetic algorithm can provide an efficient mechanism for finding good phase sequences
- Generate and test phase sequences in a process similar to biological natural selection (survival of the fittest)
- Fitness criteria can be tailored to application requirements (e.g. power)

◆ VISTA CANDIDATE OPTIMIZATION PHASES

Branch Chaining	Common Subexpression Elimination	Remove Unreachable Code	Remove Useless Blocks
Dead Assignment Elimination	Block Reordering	Minimize Loop Jumps	Register Allocation
Loop Transformations	Merge Basic Blocks	Strength Reduction	Reverse Jumps
Instruction Selection	Remove Useless Jumps	Register Assignment	Fix Entry Exit

- **Green** items are required phases
- All others are selectable during the genetic algorithm search
- Loop Transformations includes loop-invariant code motion, induction variable elimination, and loop strength reduction

③ ASSEMBLY TRANSLATION

- Translation is necessary for converting the optimized assembly code to the VISTA intermediate language (RTLs)
- Preserving semantics
 - Information Loss
 - ★ High-level languages possess a greater semantic content than a corresponding lower-level representation.
 - ★ As code is compiled/optimized/assembled, it becomes increasingly harder to reconstruct the original form
 - Local Variable Confusion
 - ★ Where do local variables on the stack start and end?
 - ★ What about data types and conversions?
 - Maintaining Calling Conventions
 - ★ Recognizing parameters (registers, stack)
 - ★ Recognizing return values

◆ LOCAL VARIABLE CONFUSION

Double placed in two integer registers		
Line	Instruction	RTLs and Comments
	...	
1	add r2, sp, #8	r[2]=r[11]+LOC[8];
2	ldmia r2, {r2-r3}	r[2]=R[r[2]];r[3]=R[r[2]+4];
3	str r2, [r5, #16]	R[r[5]+16]=r[2];
4	str r3, [r5, #20]	R[r[5]+20]=r[3];
	...	

Two integer loads coalesced in same manner		
Line	Instruction	RTLs and Comments
	...	
1	add r2, sp, #40	r[2]=r[11]+LOC[40];
2	ldmia r2, {r2, r3}	r[2]=R[r[2]];r[3]=R[r[2]+4];
3	add r3, r3, r2	r[3]=r[3]+r[2];
	...	

Code from *fft_float()* in FFT benchmark

◆ IMPLEMENTATION STRATEGY

- ASM2RTL - Translate assembly code → VISTA RTL format
- Split into machine-dependent and machine-independent parts to make additional translators easier to construct
 - Sun SPARC
 - Texas Instruments TMS320c54x
 - Intel StrongARM ← used for these experiments
- Process each assembly line individually
- Perform an additional pass on the RTL output at the end of each function if necessary
- Allow VISTA to reconstruct additional information from contextual clues
- Simplify handling of difficulties with memory consistency (local variable confusion) and calling conventions

◆ ASM2RTL PROGRAM STRUCTURE

Input: Assembly source file, [function information file], [local structure information]

Output: Register Transfer List (RTL) representation of file

- 1 read configuration data
- 2 **foreach** *function* \in *file* **do**
- 3 **foreach** *line* \in *function* **do**
- 4 parse line and pass to appropriate handler
- 5 add generated RTLs to function list
- 6 generate appropriate local variable information for RTLs
- 7 output RTL style declarations of globals
- 8 output all RTLs from function lists

◆ MEMORY CONSISTENCY

- VISTA will reorganize all local variables during *Fix Entry Exit*
- Cannot allow splitting of arrays, structures or large data types → other functions will not be able to interface correctly with them
- Problems occur when accessing the internal members of a structure or array (e.g. `a[3]` or `s.total`)
- Can be fixed by supplying the translator with a list of functions and corresponding stack information for local structure/array information

◆ LOCAL VARIABLE RECONSTRUCTION

- 1 fix all function arguments on the stack as unmodifiable
- 2 **if** *fix_structs_flag* **then**
- 3 **foreach** *struct* \in *structs_list* **do**
- 4 **foreach** *var* \in *local_vars* **do**
- 5 **if** *struct.loc* < *var* < (*struct.loc* + *struct.size*) **then**
- 6 replace *var* in RTLs with *struct.loc* + $\#(\text{var} - \text{struct.loc})$
- 7 remove *var* from *local_vars*
- 8 extract locs from local symbol table
- 9 sort locs by offset in increasing order
- 10 **foreach** *var* \in *locs* **do**
- 11 └ *var.size* = *nextvar.offset* - *var.offset*
- 12 prepend local variable declaration RTLs to function list

◆ 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
 - Can we distinguish between stack local variables and arguments?
- Knowing the number of parameters (size in words) and return types for each function, we can actually 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

- We could always assume worst case scenarios and not require additional information about the assembly code
 - Stack layout is one large array or structure, which is unable to be split
 - ★ Most optimizations ignore arrays and structures since they are hard to manipulate while guaranteeing correctness
 - ★ This decreases the chance that re-optimization will provide benefits
 - All functions use all argument registers and all stack locations may be parameters
 - ★ Stack variables are unable to be adjusted in any way (as above)
 - ★ Optimizations such as *Dead Assignment Elimination* will be less effective since we will have dead registers that are not detectable
- Luckily, most of the additional information is easy to extract by just simple inspection of the code

4 DE-OPTIMIZATION

- **Undo** the effects of previous transformations on the code
- Enables VISTA to reapply these optimization phases in a 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 to be considered loop-invariant:
 - ① All source operands must be loop-invariant
 - ② Must dominate all exits of loop
 - 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 LOOP-INVARIANT CODE MOTION

```

1  foreach loop ∈ loops sorted outermost to innermost do
2    perform loop_invariant_analysis() on loop
3    foreach rtl ∈ loop→preheader sorted last to first do
4      if rtl is invariant then
5        foreach blk ∈ loop→blocks do
6          foreach trtl ∈ blk do
7            if trtl uses a register set by rtl then
8              └ insert a copy of rtl before trtl
9    └ update loop_invariant_analysis() data

```

◆ BEFORE DE-OPTIMIZATION

Line	Label	RTLs	Comments
		...	
1		+r[10]=R[L44]	▷ Load a global variable (loop-invariant)
2		+r[6]=0	▷ Initialize loop counter
3	L11		
4		+r[2]=r[10]+(r[6]{2)	▷ Calculate array address (global + counter)
5		+r[5]=r[5]+R[r[2]]	▷ Add array value from memory to register
6		+r[6]=r[6]+1	▷ Loop counter increment
7		+c[0]=r[6]-79:0	▷ Set condition codes register
8		+PC=c[0]'0,L11	▷ Perform loop 80 times
		...	

◆ AFTER DE-OPTIMIZATION

Line	Label	RTLs	Comments
		...	
1		+r[10]=R[L44]	▷ Load a global variable (loop-invariant)
2		+r[6]=0	▷ Initialize loop counter
3	L11		
4		+r[10]=R[L44]	▷ Loop-invariant load (moved back into loop)
5		+r[2]= r[10] +(r[6]{2)	▷ Calculate array address (global + counter)
6		+r[5]=r[5]+R[r[2]]	▷ Add array value from memory to register
7		+r[6]=r[6]+1	▷ Loop counter increment
8		+c[0]=r[6]-79:0	▷ Set condition codes register
9		+PC=c[0]'0,L11	▷ Perform loop 80 times
		...	

◆ REGISTER ALLOCATION

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

◆ REGISTER ASSIGNMENT

- Frontends to VISTA and VPO generate references to pseudo-registers (which do not actually exist)
- Register assignment is the process of mapping these pseudo-registers to actual hardware registers
- Similar to *register allocation*, since conflicts must also be avoided between register live ranges
- Spill code - generated if not enough registers are available for all live ranges

◆ MOTIVATION FOR UNDOING REGISTER ALLOCATION

- Different phase orders can be tested with the reduced register pressure
- Allows for re-allocation of local variable live ranges according to VISTA's algorithm instead of the original code's choices
- Necessary analysis information also allows us to easily undo *register assignment* at the same time
- Re-assigning registers can potentially allow the code to be **squeezed** into fewer registers, thus freeing up registers for other optimizations

◆ REGISTER INTERFERENCE GRAPHS

- Register Interference Graphs (**RIGs**) are analogous to the **Interference Graphs** used for *register allocation*
- Difference is that RIGs look at the live ranges of registers and not local variables
- RIG is constructed by analyzing register usage in a basic block, then connecting predecessors and successors (sibling nodes in the graph)
- Each RIG node together with its siblings corresponds to the live range of a register in the function

```

1  foreach blk ∈ function basic blocks do
2      foreach reg live on entry to blk do
3          └ add new node to blk→blkins with rtl live register data
4      copy all blk→blkins into blk→blkouts
5      foreach rtl ∈ blk do
6          if rtl is a parameter line then
7              | mark all parameter registers as not replaceable in blk→blkouts
8          else
9              foreach reg_used ∈ rtl do
10                 if reg_used ∈ blk→blkouts then
11                     | update found node with rtl reg_used data
12                 else
13                     └ add new unreplaceable node to blk→blkouts with rtl reg_used data
14             if rtl is a reserve line then
15                 | add new node to blk→blkouts with return data
16             else
17                 foreach reg_set ∈ rtl do
18                     if reg_set ∈ blk→blkouts then
19                         | update found node with rtl reg_set data
20                     else
21                         └ add new node to blk→blkouts with rtl reg_set data
22                 foreach reg_dead ∈ rtl do
23                     | search for node using reg_dead in blk→blkouts
24                     | update found node with rtl reg_dead data
25                     └ remove node from blk→blkouts

```

```
26 foreach blk that can exit the function do mark all  $blk \rightarrow blkouts$  as not replaceable
27 foreach  $blk \in$  function basic blocks do
28   foreach  $pblk \in blk \rightarrow preds$  do
29     foreach  $node \in blk \rightarrow blkins$  do
30       foreach  $pnode \in pblk \rightarrow blkouts$  do
31         if node and pnode use the same register then
32           connect node and pnode as siblings
33         if node or pnode are not replaceable then
34           mark all siblings as not replaceable
```

◆ DE-OPTIMIZING REGISTER ALLOCATION

- Construct a **Register Interference Graph**
- Live range nodes in the graph can be:
 - **Intrablock** - live range contained in a single basic block
 - **Interblock** - live range spans multiple basic blocks
- For all replaceable nodes in the RIG, replace register references with non-conflicting pseudo-registers
- For interblock nodes:
 - Create a new local variable for the register reference
 - Insert a load of this new local before each use
 - Insert a store of this new local after each set

```

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 foreach node ∈ RIG do
6     if  $\neg node \rightarrow done \wedge node \rightarrow can\_replace$  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 all siblings with 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 foreach node ∈ RIG do
18   if  $\neg node \rightarrow done \wedge node \rightarrow can\_replace$  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           └ insert load of node → local into node → pseudo before use
28           └ replace use with node → pseudo
29         for set ∈ sib → sets do
30           └ replace set with node → pseudo
31           └ insert store of node → pseudo into node → local after set
32 re-perform register_assignment() to assign all pseudo-registers
33 perform instruction_selection() to clean up code

```

◆ DE-OPTIMIZE REGISTER ALLOCATION EXAMPLE

Function *dequeue()* from *dijkstra* benchmark

```
void dequeue (int *piNode, int *piDist, int *piPrev)
```

```
begin
```

```
1   static QITEM *qKill;  
2   qKill = qHead;  
3   if qHead then  
4       *piNode = qHead→iNode;  
5       *piDist = qHead→iDist;  
6       *piPrev = qHead→iPrev;  
7       qHead = qHead→qNext;  
8       free(qKill);  
9       g_qCount-;
```

```
end
```

RTLs Before De-Optimization			
Line	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;		▷ Check for NULL pointer
5	R[r[3]+0]=r[12];	r[3]	
6	r[4]=r[1];	r[1]	▷ Saving argument r[1]
7	r[3]=r[0];	r[0]	▷ Saving argument r[0]
8	r[5]=r[2];	r[2]	▷ Saving argument r[2]
9	r[0]=r[12];		
10	PC=c[0]:0,L0001;	c[0]	▷ If NULL then goto 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];		▷ Call free() with 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

RTLs After Deallocation			
Line	RTLs	Deads	Comments
1a	r[32] =R[L21];		▷ r[6] → r[32]
1b	R[r[13]+ __dequeue_0]= r[32] ;	r[32]	▷ Store set of pseudo-register r[32]
2a	r[32]=R[r[13]+ __dequeue_0];		▷ Load use of pseudo-register r[32]
2b	r[33]=R[r[32]+0];	r[32]	▷ Perform actual operation
2c	R[r[13]+ __dequeue_1]=r[33];	r[33]	▷ Store set of pseudo-register r[33]
3	r[34] =R[L21+4];		▷ Intrablock live range so replace ▷ with only pseudo-register r[34]
4a	r[33]=R[r[13]+ __dequeue_1];		
4b	c[0] =r[33]-0:0;	r[33]	▷ c[0] is not replaceable
5a	r[33]=R[r[13]+ __dequeue_1];		
5b	R[r[34]+0]=r[33];	r[33] r[34]	▷ Death of Intrablock r[34]
6a	r[35]=r[1];	r[1]	▷ r[1] is incoming argument
6b	R[r[13]+ __dequeue_2]=r[35];	r[35]	▷ so not replaceable
7a	r[36]=r[0];	r[0]	▷ r[0] is incoming argument
7b	R[r[13]+ __dequeue_3]=r[36];	r[36]	▷ so not replaceable
8a	r[37]=r[2];	r[2]	▷ r[2] is incoming argument
8b	R[r[13]+ __dequeue_4]=r[37];	r[37]	▷ so not replaceable
9a	r[33]=R[r[13]+ __dequeue_1];		▷ r[0] in this case is
9b	r[0] =r[33];	r[33]	▷ outgoing argument to free()
10	PC=c[0]:0,L0001;	c[0]	▷ Conditional branch uses only
	...		▷ c[0] so no replacements at all

RTLs After Register Assignment			
Line	RTLs	Deads	Comments
1a	<code>r[12]=R[L21];</code>		▷ <code>r[12]</code> is first non-argument
1b	<code>R[r[13]+__dequeue_0]=r[12];</code>	<code>r[12]</code>	▷ scratch register
2a	<code>r[12]=R[r[13]+__dequeue_0];</code>		▷ Note the use of <code>r[12]</code> to
2b	<code>r[12]=R[r[12]+0];</code>		▷ combine two distinct live
2c	<code>R[r[13]+__dequeue_1]=r[12];</code>	<code>r[12]</code>	▷ ranges in these 3 lines
3	<code>r[12]=R[L21+4];</code>		
4a	<code>r[3]=R[r[13]+__dequeue_1];</code>		▷ First appearance of <code>r[3]</code> since
4b	<code>c[0]=r[3]-0:0;</code>	<code>r[3]</code>	▷ there are currently 2 live ranges
5a	<code>r[3]=R[r[13]+__dequeue_1];</code>		
5b	<code>R[r[12]+0]=r[3];</code>	<code>r[3]r[12]</code>	
6a	<code>r[12]=r[1];</code>	<code>r[1]</code>	▷ Save argument <code>r[1]</code>
6b	<code>R[r[13]+__dequeue_2]=r[12];</code>	<code>r[12]</code>	
7a	<code>r[12]=r[0];</code>	<code>r[0]</code>	▷ Save argument <code>r[0]</code>
7b	<code>R[r[13]+__dequeue_3]=r[12];</code>	<code>r[12]</code>	
8a	<code>r[12]=r[2];</code>	<code>r[2]</code>	▷ Save argument <code>r[2]</code>
8b	<code>R[r[13]+__dequeue_4]=r[12];</code>	<code>r[12]</code>	
9a	<code>r[12]=R[r[13]+__dequeue_1];</code>		
9b	<code>r[0]=r[12];</code>	<code>r[12]</code>	
10	<code>PC=c[0]:0,L0001;</code>	<code>c[0]</code>	▷ Live registers leaving block are
	<code>...</code>		▷ <code>r[0]</code> and <code>r[13]</code> (stack pointer)

RTLs After Additional Optimizations			
Line	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;		

◆ DE-OPTIMIZATION DIFFICULTIES

- Calling conventions
 - Cannot de-optimize hardware register references used for function call arguments and return values
 - Cannot de-optimize special purpose registers like the stack pointer
 - Solved by RIG algorithm as well as a machine-dependent routine for notifying the de-optimize routine of special purpose registers
- Code expansion
 - Global variable offsets are 12 bit offsets from the program counter
 - Code that is de-optimized may be expanded so that globals are no longer reachable
 - Solved by having genetic algorithm reject phase sequences that produce code that breaks global offsets

5 EXPERIMENTAL PLAN AND RESULTS

- Problem: Hand-coded assembly applications are hard to find
 - No embedded applications benchmark suites written in assembly code
 - ★ Many embedded processors on the market, but few with exactly the same instruction set
 - ★ How to ensure fairness when looking at tuned assembly for each architecture
 - Most real-world embedded applications are proprietary
- Solution: Use an existing embedded applications benchmark suite with a suitable optimizing compiler to produce “tuned” assembly
 - **MiBench** embedded applications benchmark suite
 - **GCC** version 3.3 C compiler for the StrongARM

◆ MiBENCH EMBEDDED BENCHMARK SUITE

- 6 categories representing common embedded application domains
- 1 application chosen for study from each area
- All applications are written in C

Category	Program	Description
Automotive/Industrial	bitcount	Bit manipulation tests
Network	dijkstra	Dijkstra's shortest path algorithm
Telecomm	fft	Computes a Fast Fourier Transform
Consumer	jpeg	Creates a jpeg image from a ppm
Security	sha	NIST Secure Hash Algorithm
Office	stringsearch	String pattern matcher

◆ EVALUATION CRITERIA

- Assembly optimization strategies to compare
 - Translated GCC assembly (baseline for comparison)
 - Translated GCC assembly + re-optimization
 - Translated GCC assembly + **de-optimization** + re-optimization
- Re-optimization fitness criteria for the genetic algorithm
 - Opt. for Space - 100% static code size
 - Opt. for Both - 50% static code size/50% dynamic instruction count
 - Opt. for Speed - 100% dynamic instruction count

◆ EXPERIMENTAL RESULTS

Comparison of De-optimization and Re-optimization with GCC baseline

Benchmark	Compiler Strategy	Opt. for Space		Opt. for Both		Opt. for Speed	
		static count	dynamic count	static count	dynamic count	static count	dynamic count
bitcount	opt	-2.32 %	0.00 %	-2.32 %	0.00 %	-2.32 %	0.00 %
	de-opt	-2.32 %	0.00 %	-2.32 %	0.00 %	-2.32 %	0.00 %
dijkstra	opt	-2.16 %	-2.70 %	-1.73 %	-2.70 %	-1.73 %	-2.70 %
	de-opt	-2.60 %	-2.73 %	-2.16 %	-2.73 %	-2.16 %	-2.73 %
fft	opt	-0.75 %	-1.75 %	-1.88 %	-1.22 %	-0.94 %	-1.75 %
	de-opt	-0.19 %	-1.05 %	-0.19 %	-1.05 %	-0.19 %	-1.05 %
jpeg	opt	-3.44 %	-1.15 %	-3.37 %	-1.15 %	-3.34 %	-3.56 %
	de-opt	-2.64 %	-0.51 %	-2.45 %	-0.50 %	-2.42 %	-2.91 %
sha	opt	-5.09 %	-4.39 %	-5.09 %	-4.39 %	-2.99 %	-6.27 %
	de-opt	-2.40 %	-6.17 %	-5.09 %	-4.36 %	-2.40 %	-6.17 %
stringsearch	opt	-0.46 %	0.00 %	-0.46 %	0.00 %	-0.46 %	0.00 %
	de-opt	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
average	opt	-2.37 %	-1.66 %	-2.48 %	-1.58 %	-1.96 %	-2.38 %
	de-opt	-0.91 %	-1.74 %	-1.25 %	-1.44 %	-0.80 %	-2.14 %

◆ ANALYSIS OF RESULTS

- De-optimization turns out to be less beneficial than previously anticipated
 - Only *dijkstra* shows an overall benefit from de-optimization before re-optimization (**Bold Green** in table)
 - *Register assignment* is the critical phase when dealing with de-optimized code
- Inherent randomness of genetic algorithm appears to be more pronounced when re-optimizing code (*Italic Purple* in table)
- Dynamic instruction counts
 - Some optimization benefits may go unnoticed
 - Memory hierarchy can affect overall results

◆ REGISTER RE-ASSIGNMENT

- Re-assigning registers after de-optimization is critical
 - Re-assignment is responsible for the improvement with *dijkstra* and several other functions in the benchmarks
 - Poor re-assignment is responsible for a majority of the negative results
- VPO and VISTA choose register assignments based on which registers are available for a given live range
 - After de-optimization of *register allocation*, the insertion of loads/stores makes pseudo-register live ranges short
 - VISTA then assigns scratch registers for many of these short ranges
 - Loads/stores are later removed by *register allocation*, leaving long spanning references to scratch registers, particularly across calls
 - Register moves saving these registers are then unable to be removed

◆ REGISTER RE-ASSIGNMENT EXAMPLE

Function *keymatch()* from *jpeg* benchmark

Translated Input RTLs			
Line	RTLs	Deads	Comments
a1	<code>r[6]=r[0];</code>	<code>r[0]</code>	Copy/save parameter <code>r[0]</code>
a2	<code>r[4]=B[r[6]]&255;r[6]=r[6]+1;</code> ...		Initial load of value <code>r[6]</code> live this entire time
a3	<code>r[4]=B[r[6]]&255;r[6]=r[6]+1;</code> ...		Load inside loop

De-optimized RTLs After Register Re-assignment			
Line	RTLs	Deads	Comments
b1	<code>r[12]=r[0];</code>	<code>r[0]</code>	Copy/save parameter <code>r[0]</code>
b2	<code>R[r[13]+__keymatch_0]=r[12];</code>	<code>r[12]</code>	Interblock live range
b3	<code>r[12]=R[r[13]+__keymatch_0];</code>		so use loads/stores
b4	<code>r[1]=B[r[12]]&255;r[12]=r[12]+1;</code>		
b5	<code>R[r[13]+__keymatch_0]=r[12];</code>	<code>r[12]</code>	
b6	<code>R[r[13]+__keymatch_1]=r[1];</code> ...	<code>r[1]</code>	

RTLs After Register Re-allocation			
Line	RTLs	Deads	Comments
c1	<code>r[12]=r[0];</code>	<code>r[0]</code>	Copy/save parameter <code>r[0]</code>
c2	<code>r[8]=r[12];</code>	<code>r[12]</code>	<code>r[8]</code> allocated for <code>__keymatch_0</code>
c3	<code>r[12]=r[8];</code>	<code>r[8]</code>	
c4	<code>r[1]=B[r[12]]&255;r[12]=r[12]+1;</code>		
c5	<code>r[8]=r[12];</code>	<code>r[12]</code>	
c6	<code>r[6]=r[1];</code>	<code>r[1]</code>	<code>r[6]</code> allocated for <code>__keymatch_1</code>
	...		

RTLs After Instruction Selection			
Line	RTLs	Deads	Comments
d1	<code>r[12]=r[0];</code>	<code>r[0]</code>	Copy/save parameter <code>r[0]</code>
d2	<code>r[6]=B[r[12]]&255;r[12]=r[12]+1;</code>		Initial load of value
d3	<code>r[8]=r[12];</code>	<code>r[12]</code>	Copy/save <code>r[12]</code>
	...		<code>r[8]</code> live this entire time
d4	<code>r[4]=B[r[8]]&255;r[8]=r[8]+1;</code>		Load inside loop
	...		

- Re-assignment leaves saves and restores of `r[12]` unable to be removed from the code
- A *reverse copy propagation* optimization could allow the prior instances of `r[12]` to be replaced by `r[8]`, thus removing the need for the saves

⑥ RELATED WORK

- Assembly translation
 - Binary translation
 - ★ UQBT: University of Queensland Binary Translator - Cifuentes, Van Emmerick, Lewis, Ramsey
 - ★ EEL: Executable Editing Library - Larus
 - Link-time optimizations
 - ★ Alto: A Link-Time Optimizer for the Compaq Alpha - Muth, Debray, Watterson, DeBosschere
- De-optimization
 - Debugging optimized executables
 - ★ Dynamic de-optimization of SELF code - Hölzle, Chambers, Ungar
 - Reverse engineering executables
 - ★ Undoing predication, speculation and instruction scheduling on the Itanium - Snavely, Debray, Andrews

7 ADDITIONAL BENEFITS

- ASM2RTL used with VISTA in DSP-driven labs at UIUC
- Detecting homomorphic code in VISTA
 - Linking and execution of code is costly for evaluating phase sequences, so avoid if possible
 - Detect structurally equivalent code with different register usage
 - Method
 - ① Construct the **Register Interference Graph** for the function
 - ② Traverse RTLs replacing each register and its associated live range with the new pseudo-register from the RIG
 - ③ Perform a CRC checksum on the new code and see if it matches any other CRC checksums

◆ HOMOMORPHIC CODE EXAMPLE

Register Allocation Before Code Motion			
Line	Label	RTLs	Re-mapped RTLs
a1		r[2]=0;	r[32]=0;
a2		r[0]=L21;	r[33]=L21;
a3		r[3]=4000+r[0];	r[34]=4000+r[33];
a4	L3		
a5		r[1]=R[r[0]];	r[35]=R[r[33]];
a6		r[2]=r[2]+r[1];	r[32]=r[32]+r[35];
a7		r[0]=r[0]+4;	r[33]=r[33]+4;
a8		c[0]=r[0]? r[3];	c[0]=r[33]? r[34];
a9		PC=c[0]<0,L3;	PC=c[0]<0,L3;

Code Motion Before Register Allocation			
Line	Label	RTLs	Re-mapped RTLs
b1		r[3]=0;	r[32]=0;
b2		r[0]=L21;	r[33]=L21;
b3		r[1]=4000+r[0];	r[34]=4000+r[33];
b4	L3		
b5		r[2]=R[r[0]];	r[35]=R[r[33]];
b6		r[3]=r[3]+r[2];	r[32]=r[32]+r[35];
b7		r[0]=r[0]+4;	r[33]=r[33]+4;
b8		c[0]=r[0]? r[1];	c[0]=r[33]? r[34];
b9		PC=c[0]<0,L3;	PC=c[0]<0,L3;

⑧ FUTURE WORK

- More extensive experiments
 - Work with actual hand-tuned assembly code
 - Interface VISTA with a cycle-accurate simulator (SimpleScalar)
- Improve de-optimization
 - Further de-optimizations (e.g. *common subexpression elimination*)
 - Genetic algorithm tweaks to support turning off de-optimizations that inhibit later improvements
 - New optimization phases to clean up and enhance de-optimized code (e.g. a *reverse copy propagation* as already discussed)

9 CONCLUSIONS

- Embedded applications can have strict requirements that must be met for code size, power consumption, and/or execution time
- Hand-tuning code assembly code is a widely used practice for maximizing the quality of the generated code, but it can still exhibit phase ordering problems
- Constructed ASM2RTL - a translator for converting assembly code into VISTA RTL format
- Implemented and evaluated de-optimization techniques for undoing previous optimizations to allow for different phase ordering choices to be made by VISTA