Register Promotion in C Programs

Keith D. Cooper and John Lu

Department of Computer Science
Rice University
Houston, Texas 77005

Abstract

The combination of pointers and pointer arithmetic in C makes the task of improving C programs somewhat more difficult than improving programs written in simpler languages like Fortran. While much work has been published that focuses on the analysis of pointers, little has appeared that uses the results of such analysis to improve the code compiled for C. This paper examines the problem of register promotion in C and presents experimental results showing that it can have dramatic effects on memory traffic.

1 Introduction

The presence of pointer-valued variables in C has long been recognized as an impediment to effective compile-time optimization. Pointers introduce a degree of uncertainty into the results of static analysis. Pointer assignments create multiple names for storage locations, with the result that the compiler must avoid reordering stores to memory. Pointer arithmetic introduces further ambiguity; understanding the results of `*(p+8)` requires a detailed knowledge, at compile-time, of the run-time storage layout. In the case of values stored in the heap, such knowledge is tenuous at best. This uncertainty, coupled with separate compilation, forces compilers to treat variables with potentially exposed addresses quite conservatively. As a result, C compilers often produce code that would appear naive to a good assembly-level programmer.

Recent attempts to address this problem have focused on the problem of analyzing the values that can be taken on by pointer variables. The literature on pointer analysis has expanded greatly in recent years [18, 8, 22, 5, 14, 9, 12, 6, 13]. This paper presents a technique for using the results of pointer analysis to make the code generated by a C compiler run faster. We call the technique *register promotion*; quite simply, it uses the results of pointer analysis to determine which scalar variables can be safely kept in registers and rewrites the code to reflect those facts.

To study the problems introduced by pointer variables, we have built a pointer analyzer for our experimental compiler. It performs static analysis to derive information about which variables might be addressed by a specific pointer. To assess the value of register promotion, we implemented it and tested it using two forms of pointer analysis. The experimental results suggest that register promotion can be quite effective at reducing memory traffic; they also show that these results are relatively insensitive to the precision of the pointer analysis.

2 Background

During translation, the compiler makes many decisions that determine the "shape" of the code that is eventually generated. A particularly important decision relates to the storage of values; the compiler must determine, for each value, where it will reside at run-time. To simplify matters, we will assume that only two choices are possible: in memory and in a register.[1] Since registers are faster to read and write than memory, it is generally desirable to keep values in registers. Thus, modern RISC-style compilers try to keep as many values in registers as possible. This decision gets encoded into the shape of the intermediate language (*IL*) generated for each statement, usually as an explicit assignment of a "virtual register" to each distinct value. Definitions target this register and uses refer to it directly.

This material is based in part upon work supported by the Texas Advanced Technology Program under grant number 003604-015 and by DARPA through Army Contract DABT63-95-C-0115.

[1] Of course, different locations in memory, such as the heap and the stack, may have different properties and different costs.

Loads	*Stores*	*Purpose*
`iLoad`	–	*immediate load* a known constant value
`cLoad`	–	*constant load* an invariant, but unknown value
`sLoad`	`sStore`	*scalar load/store* a value known to be scalar
`Load`	`Store`	*general load/store* general form

Table 1: Hierarchy of memory operations

Situations can arise that prevent retention of a value in a register across statement boundaries. For example, if multiple names exist for a value, it must be stored to memory after every definition and loaded from memory before each use. The presence of pointers introduces precisely this problem; in the absence of specific knowledge about the set of variables that can be referenced by each pointer, the compiler is forced to treat references to *any storage* that the pointer might possibly address in this conservative fashion. In our compiler, like many other RISC-style compilers, this conservative treatment is enforced by inclusion in the intermediate code of explicit stores and loads for the values that cannot be enregistered safely. To improve this situation, the compiler can analyze the code to improve its knowledge, and use this information to "promote" some values from memory into registers.

The IL for our research compiler contains several features that encode information to facilitate analysis and optimization of memory-based values. Each memory operation has an associated list of "tags"; these are textual names that identify the memory locations that can be used by the operation. Procedure calls have lists of modified tags and referenced tags to record their summary side effects [7]. Finally, the IL contains a hierarchy of memory instructions that denote increasingly more specific knowledge (see Table 1.) When it emits the IL, the front end encodes the best information it has into the tag field and the opcode. In many cases, however, it must behave conservatively and assume that an operation may reference *any* memory location. The information encoded in the tags is part of the IL representation of the code and is available to every subsequent phase of compilation.

The key to improving the quality of code generated for pointer-based operations is to improve the precision of these tag sets. To achieve this, we implemented a pointer analyzer. It performs interprocedural dataflow analysis to discern better information about the addresses that each pointer variable can contain. This knowledge can improve the compiled code in two ways. First, it lets the compiler shrink the tag sets of many loads and stores. This produces better results from several of the optimizations. Second, specific transformations can capitalize directly on the pointer information to rewrite the code in ways that classical techniques cannot. This paper describes one such technique, register promotion.

3 Register Promotion

Register promotion improves code by allowing a value that normally resides in memory to reside in a register for some portions of the code. This is done by identifying sections of the code in which it is safe to place the value in a register. Before entering such a section, the value is "promoted" (*i.e.* loaded) from its memory location to a register. Within the section, references to this value are rewritten to refer to the register. Upon exit from the section, the value is "demoted" (*i.e.* stored) to a memory location.

The compiler performs promotion in the early phases of optimization (see § 5). It rewrites the IL to keep additional values in a register. However, subsequent actions by the register allocator can "undo" a promotion. At the time that promotion occurs, the compiler cannot accurately predict the availability of a register to hold the promoted value. If the register allocator discovers that demand for registers exceeds supply, it must "spill" some values back to memory. The promoted values compete for registers on an equal footing with other values; nonetheless, some of them may get spilled.

3.1 The Algorithm

The algorithm that we have developed is relatively simple. It proceeds as follows:

1. *interprocedural analysis*—The compiler performs an interprocedural analysis to disambiguate memory references. The results are used to shrink the tag sets for references and procedure calls.
2. *gather initial information*—For each block b, the compiler computes two sets. $\texttt{B\_EXPLICIT}_b$ contains all tags referenced by an explicit memory operation in b. $\texttt{B\_AMBIGUOUS}_b$ contains all tags referenced ambiguously in b, through procedure calls or pointer-based memory operations where the pointer contains multiple tags.
3. *find loop structure*—The compiler computes dominator information to identify loop nests using an algorithm due to Lengauer and Tarjan [15].
4. *analyze loop nests*—For each loop l, the compiler solves the equations shown in Figure 1. The set

$$\texttt{L\_EXPLICIT}_l = \bigcup_{b \in l} \texttt{B\_EXPLICIT}_b \quad (1)$$

$$\texttt{L\_AMBIGUOUS}_l = \bigcup_{b \in l} \texttt{B\_AMBIGUOUS}_b \quad (2)$$

$$\texttt{L\_PROMOTABLE}_l = \texttt{L\_EXPLICIT}_l - \texttt{L\_AMBIGUOUS}_l \quad (3)$$

$$\texttt{L\_LIFT}_l = \begin{cases} \texttt{L\_PROMOTABLE}_l & \text{if } l \text{ is an outermost loop} \\ \texttt{L\_PROMOTABLE}_l - \texttt{L\_PROMOTABLE}_{surrounding\_loop(l)} & \text{otherwise} \end{cases} \quad (4)$$

Figure 1: Equations for Register Promotion

$\texttt{L\_PROMOTABLE}_l$ contains the tags that may safely be promoted inside loop l.

5. *rewrite the code*—For each tag that is in some $\texttt{L\_PROMOTABLE}_l$, a virtual register v is created. All references to the tag in loops for which the tag is promotable are converted to a copy involving v.[2]

6. *promote the tags*—A tag that has had its accesses rewritten to use a virtual register must be loaded into its virtual register before entering the outermost loop in which it is promotable. It also must be stored to at the loop exits. The set of tags that needs to be loaded and stored around a loop l is in $\texttt{L\_LIFT}_l$.

The equations from Figure 1 merit some additional explanation. $\texttt{B\_EXPLICIT}_b$ and $\texttt{B\_AMBIGUOUS}_b$ are computed in a simple linear pass over each block. The pass must examine each operation and its tag set. Equations (1) and (2) simply aggregate together the information for all the blocks in a loop. Equation (3) is solved once per loop; it computes the set of values that are only referenced explicitly in the loop. If a tag `t` is in $\texttt{L\_PROMOTABLE}_l$ for loop l, the loop can be rewritten safely to keep the value associated with `t` in a register. Finally, equation (4) ensures that a tag `t` is only loaded and stored around the outermost loop where it may be promoted.

What have we accomplished? As presented, the algorithm promotes references to a scalar variable in a loop if all the references to the scalar variable in the loop are explicit. It does not promote references based on pointers that may point to multiple objects; neither does it promote array references. The promoted variables are scalars that the compiler did not enregister because it lacked the information to show that enregistering them was safe. Section 3.3 discusses one technique for extending the domain of promotion to include some array and more pointer-based values. The algorithm only examines references inside loops; our implementation of partial redundancy elimination[17] uses memory tag information to achieve most of the effects of promotion in straight-line code.

What does this algorithm cost? The cost of the interprocedural analysis used to support register promotion varies with both the algorithm used and the desired precision of the information (See Sections 4 and 5). The promotion algorithm itself runs efficiently. Its complexity is expressed as a function of the following variables that characterize a program.

C	code size
T	number of tags
L	number of loops
X	maximum number of exits in a loop
B	number of basic blocks
E	number of edges in CFG

Computing `B_EXPLICIT` and `B_AMBIGUOUS` takes a simple pass over the code. In each block, it examines each statement and, possibly, each tag set. This requires $\mathbf{O}(CT)$ time, worst case. The dominator algorithm used to find the loop structure can be implemented to require $\mathbf{O}(E\alpha(E,B))$ time, where $\alpha(E,B)$ is related to a functional inverse of Ackermann's function [15]. Computing `L_EXPLICIT` and `L_AMBIGUOUS` requires $\mathbf{O}(LBT)$ time, while `L_PROMOTABLE` and `L_LIFT` require $\mathbf{O}(LT)$ time. Rewriting the code requires $\mathbf{O}(C)$ time to convert memory operations to copies, plus $\mathbf{O}(TLX)$ time to insert loads and stores at loop landing pads and loop exits. Thus, the overall time bound is

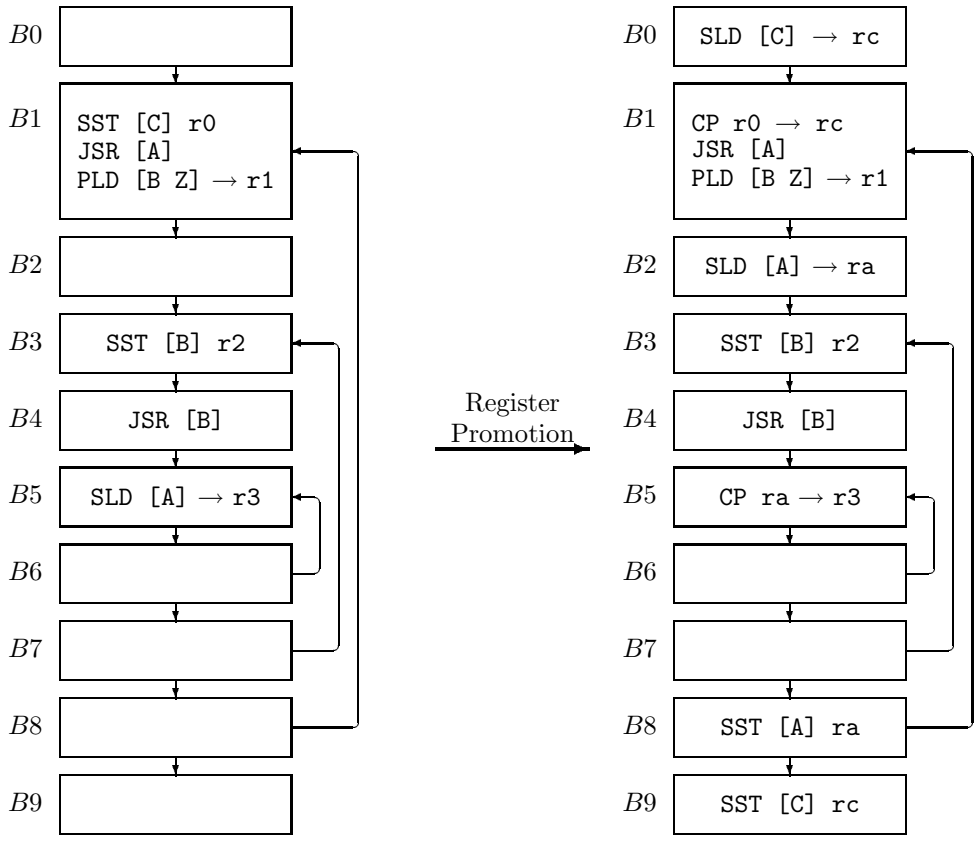
$$\mathbf{O}(CT + E\alpha(E,B) + LBT + LT + C + TLX),$$

which simplifies to

$$\mathbf{O}(E\alpha(E,B) + T \cdot (C + LB + LX)).$$

In practice, it runs quite quickly.

[2]The copies are subject to coalescing by the register allocator [1]. It is quite effective at eliminating copies like these.



Loop Nest and Relevant Code

Block Information										
Set	0	B1	B2	B3	B4	B5	B6	B7	B8	B9
B_EXPLICIT		C		B		A				
B_AMBIGUOUS		A B Z			B					

Loop Information							
Loop	Blocks	Landing Pad	Exit	L_EXPLICIT	L_AMBIGUOUS	L_PROMOTABLE	L_LIFT
B1	B1-B8	B0	B9	A B C	A B Z	C	C
B3	B3-B7	B2	B8	A B	B	A	A
B5	B5-B6	B4	B7	A		A	

Figure 2: An Example

```
for (i=0; i<DIM_X; i++) {
  B[i]=0;
  for (j=0; j<DIM_Y; j++) {
    B[i]+=A[i][j];
  }
}
```

Original code

```
for (i=0; i<DIM_X; i++) {
  rb=0;
  for (j=0; j<DIM_Y; j++) {
    rb+=A[i][j];
  }
  B[i]=rb;
}
```

Transformed code

Figure 3: Promoting Array References

3.2 An Example

To make this discussion more concrete, consider the example shown in Figure 2. It shows a triply nested loop, along with some of the code that populates the loop. (The remaining code is assumed to have no impact on the example.) The instructions are presented in an abstracted form; each shows its tag list followed by any relevant registers. The mnemonics have simple meanings.

SST	Scalar store
SLD	Scalar load
CP	Register copy
PST	Pointer-based store
PLD	Pointer-based load
JSR	Jump to subroutine

The version on the left shows the code before register promotion. The version on the right shows the results of register promotion. Notice that each loop has an explicit landing pad before its header and an explicit exit block. Our compiler automatically inserts landing pads and exits as part of constructing the control-flow graph; empty blocks are automatically removed after optimization. Loops are referred to by the block number of their headers.

The tables at the bottom of the figure show the local information computed for the example, B_EXPLICIT and B_AMBIGUOUS, as well as the sets computed for the loops. The L_PROMOTABLE and L_LIFT sets concisely summarize the situation. The value associated with tag A is promotable in the two inner loops but not the outer loop. The JSR instruction in block B1 references A ambiguously, so it cannot be promoted in that loop. The value associated with tag B is referenced ambiguously in loop B3; since it is not referenced in loop B5, no opportunity for promoting it exists. Finally, the value associated with tag C is never referenced ambiguously. Since it is referenced in the outer loop (B1), it is promotable in that loop. The L_LIFT set correctly shows that A should be promoted in B3 rather than B5 since loop B3 contains loop B5.

When the compiler rewrites the code, it promotes C in loop B1 and A in loop B3. Thus, it inserts a scalar load of C into rc in loop B1's landing pad (block B0) and a scalar store into loop B1's exit block (B9). The store in block B1 becomes a copy into rc. To promote A, it inserts a scalar load of A into ra in loop B3's landing pad (B2), and a scalar store into loop B3's exit block (B8). The load in B5 becomes a copy out of ra. The other instructions remain unchanged.

The net result is to replace the scalar load in the innermost loop with a copy operation and a load/store pair two loops farther out. The scalar store in the outer loop is replaced with a copy operation and a load/store pair outside that loop. In many cases, the register allocator can coalesce away these copies.

3.3 Handling Pointer-based References

The algorithm from Section 3.1 only promotes scalar variables that are only explicitly referenced. Pointer-based loads and stores that may point to multiple locations cannot be modified. Consider, for example, the code shown on the left hand side of Figure 3. The inner loop uses a single value of B[i] per iteration of the outer loop; since i does not change, the address of B[i] is invariant. Thus, the compiler should rewrite the code as shown on the right by promoting B[i] into a register rb. This eliminates a load before the reference to B[i] in the inner loop and a store after it. To achieve this, however, the compiler must recognize that B[i] refers to the same location in each iteration of the inner-loop and that only one way to reference the code is possible in the inner loop. The analysis described earlier cannot do that.

We developed another algorithm to promote some pointer-based references to multiple locations. In particular, it finds memory references, r, where the base

register, b, is invariant in a loop and the only accesses in the loop to the tags accessed by r are through the invariant base register b. This algorithm relies on loop-invariant code motion to identify the loop-invariant base registers and place the computation of these registers outside a loop. When it finds memory references satisfying these conditions, it promotes the reference into a register using the same rewriting scheme as before—a load before each loop entry, a store at each loop exit, and a copy at each reference. These conditions include the example from Figure 3.

Anecdotally, the pointer-based promotion scheme is a success. When its conditions apply, it produces the code that might be expected of a good assembly programmer. For example, it produces a loop equivalent to the transformed code shown in Figure 3, after coalescing removes the copy operations. In our suite of test programs, however, the measured improvements were not overwhelming when compared with scalar promotion. For total operations executed, pointer-based promotion hurt performance for one program and had no effect on nine others. The improvements in three of the other four programs were less than 1% of the improvement due to scalar promotion. In **fft**, the only significant success, pointer-based promotion was able to remove 48.3% more operations, 48.3% more stores, and 48.4% more loads than scalar promotion was able to remove. This accounted for 0.41% of the stores and 0.34% of the loads in the execution of **fft**. The reason for this disappointing performance may be that the restrictions are too strict; it may be that the promotable pointer-based references in our collection of programs are relatively unimportant to performance. We intend to continue to investigate this set of problems.

3.4 Planned Improvements

Our current register promoter misses some opportunities. We are interested in extending this work to increase its coverage of real programs.

- The loop-based approach to analysis and transformation causes the promoter to overlook situations that occur outside loop nests. There should be many cases where it is profitable to promote values to registers in straight-line code.

  In our compiler, partial redundancy elimination catches many of these cases in straight-line code. It uses the tag fields to eliminate redundant loads. It must treat stores more conservatively. Extending the promoter could improve the behavior for these stores.

- The array references handled by the scheme described in Section 3.3 catches a set of relatively simple cases. Some of the more complex examples require detailed dependence analysis or an equivalent technique to reason about conflicts with other references to the same array inside the loop [11, 16]. For example, Carr used dependence analysis to detect consistent patterns of cross-iteration reuse in Fortran and to promote the corresponding values into scalar temporaries that ended up in registers [2].

  We are interested in expanding the set of array references promoted by the compiler. Our work to date has focussed on poor code that results from lack of information about the behavior of other procedures and pointer-based memory operations. As we delve deeper into array promotion, we will need to improve our analysis of subscripts.

As in any experimental study, examining the code that comes out of the compiler suggests additional areas of improvement. Further cases for improvement will suggest themselves as we continue this work.

However, we must sound a note of caution. Register promotion increases the demand for registers—often called *register pressure*. As we improve the promoter, we increase its ability to generate an intermediate code program that requires spilling in the register allocator. Carr discovered this effect in his work on scalar replacement in Fortran [3]; beyond some point, the memory accesses removed by the transformation were balanced by the spills added during register allocation. He adopted a bin-packing discipline to "throttle" the promotion process. As we extend our work, we will undoubtedly encounter the same problem and need a similar solution to moderate register pressure.

4 Our Approach to Analysis

To test our ideas, we implemented two forms of interprocedural analysis: interprocedural MOD/REF analysis and a "points-to" analysis. The results of analysis are used to limit the size of the tag sets for pointer-based memory operations and function calls.

The MOD/REF analyzer starts by limiting the tag sets of pointer-based memory operations in two ways. First, only tags that have had their address taken are placed in the tag sets of pointer-based memory operations. The front end identifies these tags. To further limit the tag sets, it only places the tag of a local variable into the tag sets of memory operations that appear in descendants of the function that creates the local variable. Indirect calls are conservatively assumed to target any addressed function. Once pointer-based memory operations are limited, the tag sets of function calls can be limited. A function call receives the tag set of the called function. A function's tag set is the union of the sets of tags that it uses or its descendants in the call graph use. To compute function tag sets, the

Program	*Lines*	*Description*
tsp	760	a traveling salesman problem
mlink	9264	genetic linkage analysis
fft	1037	fast-fourier transform
clean	11191	basic-block cleaning pass
cachesim	2849	cache simulator
dhrystone	534	classic benchmark code
water	1345	from SPLASH benchmark
indent	5955	prettyprinter for C programs
allroots	215	polynomial root-finder
bc	7583	calculator language from GNU
go	28553	game program from SPEC benchmarks
bison	10179	LR(1) parser generator
jpeg	19842	graphics compression code from SPEC
gzip	7331	file compression program

Figure 4: Program Descriptions

algorithm identifies the strongly-connected components (SCC) of the call-graph, and calculates the tag set of each SCC. Inside an SCC, all the functions have identical tag sets. Processing the SCCs in reverse topological order ensures that the tag set of any called function not in the current SCC has already been calculated.[3]

Our approach to pointer analysis is similar to Ruf's work [18]. We analyze the entire program at once. Each function is converted into SSA form. For each SSA name, the analyzer determines the set of tags to which it may point. This is done by initializing SSA names with the pointer values they may initially have. Pointer values are propagated through the program using a worklist algorithm. Non-local memory is modeled with explicit names rather than representative names. Heap memory is modeled with a single name for each call-site that can generate a new heap address. The analysis is context-insensitive. The effects of recursion are approximated. Addressed locals of recursive functions are represented with a single name. Since this one name represents multiple locations, strong updates are not possible.

Once the analyzer has found a set of possible pointers for each pointer-based memory operation, it can calculate a more restrictive set of tags for each pointer-based memory operation. MOD/REF analysis is then repeated, using the new tag sets for the pointer based-memory operations.

[3]This algorithm seems quite simple. The equations for MOD and REF are drastically simplified by C's lack of call-by-reference parameters [7].

5 Experimental Results

To understand the impact of register promotion, we compiled 14 C programs using our laboratory compiler (see Figure 4). Four versions of each program were prepared, using the combinations of scalar promotion, no scalar promotion, MOD/REF analysis, and pointer analysis. Each version was optimized with value numbering, partial redundancy elimination, constant propagation, loop invariant code motion, dead code elimination, register allocation, and a basic block cleaning pass. Each version was instrumented to record the total number of operations executed, stores executed, and loads executed. These results are shown in Figures 5, 6, and 7. Note that these numbers are for whole programs, rather than individual procedures.

The principal effect of register promotion is the removal of memory operations–stores and loads. The figures for total operations executed show small improvements in some applications. Figures 6 and 7 provide a more precise picture of the impact of promotion. In several of the applications, promotion removed a large fraction of the stores and many of the loads. In other applications, it found few, if any, opportunities. When it found opportunities, the promoter often made significant improvements. If memory operations take more cycles than other operations, as in many modern machines, the positive impact of promotion will be greater.

In some cases, the net effect of promotion was a minor performance degradation. Degradation was caused by two effects, promoting rarely used or conditionally used values and increasing register pressure. For example, in **dhrystone**, values were promoted in a loop that always executed once, and in **bison**, values were promoted that were only accessed on an error condition. In **water**, register promotion was able to promote twenty-eight values for one loop nest. Unfortunately, this caused the register allocator to spill values which resulted in a performance loss compared to no register promotion.[4] Most of the improvements were the result of global variables which are normally placed in memory being promoted to registers.

The results also show that the improved information derived from pointer analysis does not greatly improve the results of register promotion. This does not warrant a conclusion that pointer analysis is unprofitable; it does suggest that MOD/REF analysis is a good basis for evaluating the benefits of improved analysis.

For example, register promotion removed 2.8 million loads from one function in **mlink**. This particular improvement did not require the extra precision provided

[4]It might be expected that the allocator would simply spill some subset of the twenty-eight promoted values and avoid the actual performance degradation. Our compiler uses a graph-coloring allocator [1]. These allocators are known to "over-spill" in tight situations.

Program	*analysis*	*without*	*with*	*difference*	*% removed*
`tsp`	modref	657067	657067	0	0.00
	pointer	651283	651283	0	0.00
`mlink`	modref	132386726	126902038	5484688	4.14
	pointer	130108670	124562634	5546036	4.26
`fft`	modref	12636489	12635955	534	0.00
	pointer	12575809	12558866	16943	0.13
`clean`	modref	1106502	1087295	19207	1.74
	pointer	1112754	1097759	14995	1.35
`cachesim`	modref	11590290	11590303	-13	0.00
	pointer	11588693	11588707	-14	0.00
`dhrystone`	modref	588194	596194	-8000	-1.36
	pointer	560195	568195	-8000	-1.43
`water`	modref	13736948	13737985	-1037	-0.01
	pointer	12934445	12935195	-750	-0.01
`indent`	modref	869492	865766	3726	0.43
	pointer	869609	866171	3438	0.40
`allroots`	modref	1011	1011	0	0.00
	pointer	1001	1001	0	0.00
`bc`	modref	5630210	5605846	24364	0.43
	pointer	5642432	5618569	23863	0.42
`go`	modref	13136359	13118131	18228	0.14
	pointer	13013054	12995062	17992	0.14
`bison`	modref	3347717	3348166	-449	-0.01
	pointer	3344839	3345288	-449	-0.01
`jpeg`	modref	37071266	37071326	-60	0.00
	pointer	37070488	37070548	-60	0.00
`gzip(enc)`	modref	5813661	5712142	101519	1.75
	pointer	5804903	5679946	124957	2.15
`gzip(dec)`	modref	984570	984770	-200	-0.02
	pointer	984202	984257	-55	-0.01

Figure 5: Total Operations

by pointer analysis; note, however, that pointer analysis did enable other improvements in **mlink**. An example where pointer analysis was required to promote a value arose in **fft**.

```
for (I = begin; I < end; I++)
  for (J = 0; J < N3; J++)
    for (K = 0; K < N1; K++)
      {
        index3 = (I*N3+J)*N1+K;
        index1 = (I*N3+J)*N1*2+K;
        T1 = pow(X3[index3], (double) KT) ;
        X2[index1] = T1 * X1[index1];
        X2[index1+N1] = T1 * X1[index1+N1];
      }
```

`T1`'s address is taken elsewhere in this code. `X2` is a pointer so the stores through it may modify `T1`. Thus `T1` is not promotable with just MOD/REF analysis. Pointer analysis can discover that the stores through `X2` cannot modify `T1`, and thus `T1` can be promoted.

Finally, some of the improvement due to register promotion was hidden because other passes in the optimizer achieve similar results. For example, loop invariant code motion can remove a load of a constant value out of a loop. Register promotion's main benefit seems to be transforming multiple stores of a promoted variable in a loop to a single store at the loop's exit, an effect that other optimization passes cannot achieve.

6 Related Work

The literature contains prior work that relates to our work in two broad categories: techniques that use improved analysis to enregister more values and methods

Program	*analysis*	*without*	*with*	*difference*	*% removed*
`tsp`	modref	51049	51049	0	0.00
	pointer	51049	51049	0	0.00
`mlink`	modref	5885109	2506412	3378697	57.41
	pointer	5885454	2358048	3527406	59.93
`fft`	modref	1036669	1036401	268	0.03
	pointer	1016181	1007706	8475	0.83
`clean`	modref	86889	84035	2854	3.28
	pointer	86888	84034	2854	3.28
`cachesim`	modref	594474	594474	0	0.00
	pointer	594474	594474	0	0.00
`dhrystone`	modref	60012	60012	0	0.00
	pointer	56012	56012	0	0.00
`water`	modref	1080062	1080060	2	0.00
	pointer	1064672	1064605	67	0.01
`indent`	modref	71302	68462	2840	3.98
	pointer	71302	68462	2840	3.98
`allroots`	modref	11	11	0	0.00
	pointer	11	11	0	0.00
`bc`	modref	273922	249732	24190	8.83
	pointer	273916	249727	24189	8.83
`go`	modref	751252	432982	318270	42.37
	pointer	750242	432183	318059	42.39
`bison`	modref	540001	539666	335	0.06
	pointer	540001	539666	335	0.06
`jpeg`	modref	2566591	2566613	-22	0.00
	pointer	2566861	2566883	-22	0.00
`gzip(enc)`	modref	274719	212585	62134	22.62
	pointer	274718	199108	75610	27.52
`gzip(dec)`	modref	17575	17389	186	1.06
	pointer	17575	17243	332	1.89

Figure 6: Stores

for disambiguating memory references in the presence of pointers. The goal of our work is to enregister values that the compiler placed in memory because it lacked the information to safely place them in registers. When the algorithm discovers such a value, it allocates a new virtual register and rewrites the code accordingly. This work is similar to Carr's work on scalar replacement in Fortran [2, 4, 3]. Carr developed a source-to-source translator that used dependence information to promote array elements that are reused across different iterations of the loop. To force the value into a register, he rewrote the code using a scalar temporary, subjecting his results to the vagaries of subsequent optimization and register allocation. Our technique works from data-flow information rather than dependence and promotes a different set of references (scalar values and pointer references based on loop invariant base registers). 214z

The work on pointer analysis is somewhat peripheral to this paper. Our effort has focussed on discovering ways to profit from pointer disambiguation. However, since pointer analysis has received so much attention in the literature, some comparisons with previous work in that arena are warranted. Many papers have described techniques for discerning information about the side effects of memory operations through pointer variables. Our work is based on "points-to" analysis; earlier work, like Landi and Ryder [13];, Choi, Burke, and Cytron [6]; and Deutsch [8] cast the problem in an "aliasing" framework. Algorithms for computing "points-to" information have been described by Emami, Ghiya, and Hendren [12]; by Wilson and Lam [22]; aand by Ruf [18]. Steensgaard showed a linear-time algorithm for performing a flow-insensitive points-to analysis by casting it as a type-inference problem [20]. Other papers have

Program	*analysis*	*without*	*with*	*difference*	*% removed*
`tsp`	modref	113723	113723	0	0.00
	pointer	107940	107940	0	0.00
`mlink`	modref	30033837	27232067	2801770	9.33
	pointer	28738653	25717677	3020976	10.51
`fft`	modref	1253052	1252784	268	0.02
	pointer	1216700	1208230	8470	0.70
`clean`	modref	188518	184919	3599	1.91
	pointer	185694	182095	3599	1.94
`cachesim`	modref	1901621	1901622	-1	0.00
	pointer	1900700	1900701	-1	0.00
`dhrystone`	modref	62024	62024	0	0.00
	pointer	54024	54024	0	0.00
`water`	modref	2675456	2676497	-1041	-0.04
	pointer	1912295	1913113	-818	-0.04
`indent`	modref	161367	145378	15989	9.91
	pointer	161927	145992	15935	9.84
`allroots`	modref	145	145	0	0.00
	pointer	141	141	0	0.00
`bc`	modref	843078	823047	20031	2.38
	pointer	838366	818341	20025	2.39
`go`	modref	1954619	1650736	303883	15.55
	pointer	1877295	1573600	303695	16.18
`bison`	modref	553970	553728	242	0.04
	pointer	552830	552588	242	0.04
`jpeg`	modref	6659671	6659684	-13	0.00
	pointer	6658356	6658369	-13	0.00
`gzip(enc)`	modref	910746	854886	55860	6.13
	pointer	902023	836200	65823	7.30
`gzip(dec)`	modref	136740	136387	353	0.26
	pointer	136408	136055	353	0.26

Figure 7: Loads

looked at the problem of deriving better understanding of the "shape" of objects in the heap; two of the most recent are by Ghiya and Hendren [10] and Sagiv, Reps, and Wilhelm [19].

Each of these techniques has strengths and weaknesses. We chose to compute points-to rather than aliasing because we felt that it more closely corresponds to the problem that we were addressing in the compiler. It also handles function pointers in a natural and useful way. Our work on transformations, to date, have not looked deeply into the heap; thus, our analyzer uses the simple expedient of splitting the heap by allocation site as opposed to the more precise techniques used in shape analysis. Within the literature on points-to analysis, variations occur in the choice of approximation techniques for modeling parameter binding, for modeling the heap, and for tracking path information during analysis. The following table summarizes these differences. For parameter binding, the methods either propagate representative names or explicit names. The heap is either modeled as a single name or it is split on some criteria, like the call path that reaches the allocation site or the specific call site causing the allocation. Finally, they differ in the amount of information about call paths that they associate with names.

	Parm's	*Heap*	*Path Info.*
Emami *et al.*	repr	1 name	arbitrary
Wilson & Lam	repr	call path	arbitrary
Ruf	explicit	call site	none/arbitrary
Our work	explicit	call site	none

The experimental results shown in Section 5 might be interpreted as showing the benefits that can be obtained from increasing the precision of program anal-

ysis. These numbers are *not* a general indictment of pointer analysis; they simply show that increased precision did not significantly change the results from *our* transformation. Little prior work compares the results obtained by using a fixed set of transformations with different precisions of program analysis. One notable exception is David Wall's work on available instruction level parallelism (ILP) [21]. Wall estimated available ILP under a set of five different assumptions for the quality of analysis available in the compiler. His results suggest that increased precision in the analysis of pointers can have a significant impact on the amount of ILP discovered by a compiler.

7 Summary and Future Directions

We have implemented a method for register promotion that is loop-based. This method converts references to scalar values in memory to references to a register. We have shown that this technique can have a substantial impact on memory traffic for whole programs even after register allocation. As memory references become more expensive this reduction in memory traffic will become more important.

Register promotion can increase register pressure. This, in turn, can cause the register allocator to spill some values by inserting new loads and stores. These spill operations hurt performance; in some cases, this effect can lead to slower code than that obtained without register promotion. To guard against this problem, we may need to extend our promotion algorithm with an explicit decision-making process that considers register pressure and frequency of use before promoting a value.

Acknowledgments

This work would not have been possible without the support of many people. The implementation was built as part of the Massively Scalar Compiler Project at Rice; it relies on the work of everyone who has contributed to that effort. Tim Harvey and Phillip Schielke played critical roles in extending the software to support this work. The exposition of this work was improved substantially by comments from members of the program committee for PLDI 97 and from Linda Torczon. Funding was provided by the State of Texas ATP program and DARPA ITO.

References

[1] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[2] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[3] Steve Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, Department of Computer Science, 1992.

[4] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. Technical Report TR92283, Rice University, CRPC, November 1992.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[6] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[7] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Notices*, 23(7):57–66, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[8] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[9] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994.

[10] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c? *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 1–15, January 1996.

[11] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. *SIGPLAN Notices*, 26(6):15–29, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[12] Joseph Hummel, Laurie J. Hendren, and Alexander Nicolau. A general data dependence test for dynamic, pointer-based data structures. *SIGPLAN Notices*, 29(6):218–229, June 1994.

[13] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, June 1992.

[14] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[15] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[16] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Notices*, 26(6):1–14, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[17] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[18] Erik Ruf. Context-insensitive alias analysis reconsidered. *SIGPLAN Notices*, 30(6):13–22, June 1995.

[19] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 16–31, January 1996.

[20] Bjarne Steensgaard. Points-to analysis in almost linear time. *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 31–41, January 1996.

[21] David W. Wall. Limits of instruction-level parallelism. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 176–189, Santa Clara, California, 1991.

[22] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, June 1995.