An Overview of the PL.8 Compiler

Marc Auslander and Martin Hopkins

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

Abstract

The PL.8 compiler accepts multiple source languages and produces high quality object code for several different machines. The strategy used is to first do a simple translation of the source program to a low level intermediate language. Global optimization and register allocation are then used to improve code rather than relying on special case code selection.

Introduction

The PL.8 Compiler was developed as part of an exploration of the interaction of computer architecture, system design, programming language, and compiler techniques. It currently supports two source languages, Pascal and PL.8, a PL/I variant for systems programming. Object code is produced for for the System/370, MC68000, 801 (10) and two other experimental machines. The role of the compiler is to support high level language programming in a style which does not require attention to detailed issues of performance, and yet to provide very "good" code. The previous experience with using optimizing compilers in this way has been discouraging. Programmers have had to avoid particular language or styles for performance reasons. Furthermore, the need to cover the many cases for which good code was required caused the compiler to grow until its size, execution cost, or lack of reliability limited further progress. In order to avoid this difficulty, and still be free to add many new features to the language and its compiler, the PL.8 effort followed a strategy of dividing compilation into a series of simpler, independent problems.

The PL.8 compiler has four major components. These are:

Translation - the conversion of the source program to an intermediate language (IL). This can be thought of as the instruction set of a simple abstract machine.

Optimization - the transformation of the intermediate language program to an equivalent program with reduced running time or size.

Register allocation - the reduction of the register space of the program to that of the machine and the assignment of specific machine registers.

Final assembly - the selection of actual target machine instructions, and the formatting of the compiler output.

The compiler further partitions optimization into as many independent operations as possible to make them more reliable and easier to implement. This approach has lead to a compiler which reliably produces object code which is generally superior to that possible with previous techniques.(7)

In what follows, we describe the overall structure of the PL.8 compiler, with emphasis on the theme of separation. We also describe some of the optimization and register allocation techniques needed to make the approach workable, and with the use of an extended example, attempt to provide some insight into why the technique works. The bibliography indicates some more detailed reports of the specific algorithms used.(3,4,5,6,8,9)

The Intermediate Language

The IL can be thought of as the assembly language for a simple abstract computer. Its semantics closely match the computational semantics of the target machines. Its integer arithmetic is 32 bit binary twos complement, and storage is byte addressed in a

32 bit address space. The IL is represented and processed as a linear string of such primitive operations. It is this code that will be optimized. For maximum effect it is necessary to expose all instructions that will be executed on the target machine. The IL model is thus at a lower level than some of the target CPUs. All computation is done in registers. Code is generated as if there were no storage to register add or similar instructions. Thus the source statement:

```
x=y+z;
```

might result in:

```
L   RY,Y
L   RZ,Z
A   R100,RY,RZ
ST  R100,X
```

This has the advantage of permitting commoning or code motion of any or all of the instructions. For example the load of y might be moved out of the loop, while the load of z might be eliminated by another load of z in the loop. (For a short description of IL, see the Appendix.)

While the IL is low level, it is more general than many computers. For example there are no destructive instructions. Add has two operands and a result even when compiling to the System/370 or MC68000 on which one of the operands of add is destroyed. On the other hand there are subtle ways in which the target CPU shows through, an example being instruction displacements. On the MC68000 the range is between -32K and +32K, while on the System/370 it is zero through 4K. It is important to expose the arithmetic needed to deal with these limits to optimization. Thus a load of a variable at displacement 5000 on the MC68000 would result in a single load instruction, while the System/370 requires:

```
AI  R100,R.X,4096      Base of x + 4k
L   RX,X-4096(R100)    Load x
```

Notice that the fact that the IL reflects some aspects of the target machine in no way affects optimization. We do machine independent optimization on an IL that is partly machine dependent.

In one sense the IL is like no existing machine. There are an unlimited number of symbolic registers. By convention, instructions with the same operands always produce their results in the same symbolic register. This is done to assist in finding common subexpressions. The format of the IL provides for only one result of a computation, a symbolic register. Condition codes are ignored at this time. We assume that compares produce a symbolic register, which contains less than, equal and greater than bits. IL branches test these symbolic register bits. The IL also contains a set of string operators which may produce string symbolic register results. These operations are subject to the same optimizations as the rest of the IL.

Translation

PL.8 translation is accomplished by conventional techniques. It is done "bottom up", with the IL code for each fragment being emitted independent of context. Translation avoids special cases, but is concerned with the overall computational strategy of implementing its source language. For example, the PL.8 translator translates the iterative do statement

```
do i = m to n;
...
end;
```

into the IL equivalent of:

```
    L       R100,M
    ST      R100,I
    L       R102,I
    L       R105,N
    C       R104,R102,R105
    BT      R104,GT,L4
L3:
    (loop body)

    L       R102,I
    AI      R103,R102,1
    ST      R103,I
    L       R102,I
    L       R105,N
    C       R104,R102,R105
    BF      R104,GT,L3 14:
```

The strategy of replicating the loop test is thus implemented by translation. The analysis needed to take into account what is known about m and n (which might be constants or expressions) or modifications to i in the loop is left for optimization and register allocation. For example, if the code at the loop head is found to contain only constant operands optimization will eliminate the compare and branch, leaving the correct special case for loops with constant bounds.

The example above also indicates that the control operations of the IL are simple conditional and unconditional branches. The control structure of the program is discovered from the IL, source program control flow clues having been eliminated. This approach simplifies translation and makes optimization

independent of particular source language control constructs.

In addition to avoiding analysis better left to optimization, the translator does no register allocation. Rather, the translator assumes only that the current stack frame is addressable via a register. All other address computations are expressed as complete follows of the appropriate addressing path starting at in the stack frame. For example, to fetch a variable x which has been declared to be static external requires:

```
L  RS,/.STATIC(RAUTO)   address of static
L  R.X,.X(RS)           address of X
L  RX,X(R.X)            X
```

PL.8 does not preassign registers to specific values or addresses when the program is translated. Rather, if any address or value should be assigned to a register to get the best performance, this will be discovered by the normal mechanisms of optimization and register allocation.

Optimization

The optimization section of the compiler performs a number of independent transformations. Each of these may be performed several times. Optimizations are repeated because one may provide new opportunities for another, or may introduce new code requiring optimization. Dealing with these interactions by iteration reduces the complexity of the individual transformations much as optimization reduces the complexity of translation. This approach does require that the format of IL remain invariant during all of optimization so that the result of any transformation can be the input to any other.

The most important transformations in the PL.8 optimizer are:

Dead Code Elimination - the elimination of computations whose results are unused

Common subexpression elimination - the elimination of a computation whose result is still available for use because of a previous execution of the same or an equivalent computation (sometimes called "commoning" in what follows.)

Code motion - the moving of computations to a place in the program that results in faster execution.

Constant expression evaluation - the compile time evaluation of operations whose operands become known.

Strength reduction - the recognition of iterative variables which appear in products, and the replacement of the products with other iterative variables. (Note that translation has expanded all addressing, including array indexing, into the implied arithmetic computations. Thus array references indexed by do indexes will be processed by this optimization.)

Value numbering - a more powerful form of common subexpression elimination which can take into account the effects of assignment. Value numbering works over extended basic blocks, while common subexpression elimination operates globally. For example:

```
Source      IL

x=a+b;      L   RA,A
            L   RB,B
            A   RAB,RA,RB
            ST  RAB,X
y=a         ST  RA,Y
a=w         L   RW,W
            ST  RW,A
z=y+b       ST  RAB,Z
```

The computation y+b is the same as a+b before the variable a was altered.

Dead store elimination - much like dead code elimination, but applied to stored values rather than the results of expressions.

Straightening - the elimination of unnecessary branches by combining successive basic blocks which are unconditionally connected into a single block.

Trap elimination - the recognition of certain cases in which tests for out of bounds values can be moved from loops by appropriately modifying the loop termination test(9).

Reassociation - the rearrangement of expressions within loops to gather together all the loop constant values. For example, if an expression has the form:

(loop varying+loop constant1)+loop constant2

none of the computation can be moved out of the loop. By reassociating the expression as:

(loop constant1+loop constant2)+loop varying,

an add can be moved from the loop.

All these transformations (and others) operate pervasively on an IL program in which all computations are exposed. Arithmetic leading to an address value is treated no differently from that leading to a program result. Conditional expressions introduced to check array bounds violations are treated no differently from those that implement source language loop constructs or those written by the programmer. Thus, each transformation is in practice well tested and is applied often to every program that is compiled. There is little in optimization which can long go untried because the special case it represents is unusual.

In addition, the independence of the transformations simplifies and at the same time improves each of them. As an example, code motion is in fact a complex operation. A computation in a loop must be found which can be done outside the loop instead. Then, the computation must be removed and placed outside. This analysis is subject to many errors which could damage the program. Our approach is to replace motion with insertion. Once analysis determines that code should be moved, the code is in fact just copied to the target of the move. This insertion itself can do no harm to the program. Later, common subexpression elimination will eliminate the in loop version of the computation. Thus, the correctness of code motion is almost assured by the correctness of common subexpression elimination. Many transformations are realized as simple modifications followed by "bread and butter" reoptimization.

Operation Expansion

In some instances it is better to optimize complex operations and then expand them. After the expansion, optimization is needed again to deal with the newly exposed operations. The max and min functions are examples of this. Max and min are implemented by compare and branch logic on most architectures. Our analysis is incapable of recognizing that such a computation is in fact a pure function which can be eliminated if redundant. Thus, max and min operators are introduced into the IL. These operators can be eliminated or moved. Part way through optimization these higher level operations are expanded. Thus:

```
MAX     RX,RA,RB
```

becomes:

```
MR      RX,RA  move
C       R100,RA,RB
BF      R100,LT,L
MR      RX,RB  move
L:
```

Appropriate boolean expressions are also expanded into compares and branches at this point. This permits boolean expressions or their components to be eliminated and moved out of loops, while still performing short circuit evaluation (anchor pointing).

During the expansion portion of optimization, the specific machine register requirements of subroutine and library linkage are introduced. These have been suppressed because, like control flow, they make it difficult to recognize opportunities for optimization. For example, on an experimental machine that has no multiply instruction x*y is initially expressed by:

```
MULT    R100,RX,RY
```

Optimization can move and common this instruction, while the use of real registers after expansion makes it harder to optimize. After expansion, the operation becomes:

```
MR      R2,RX
MR      R3,RY
CALL    R2,MULT(R2,R3)
MR      R100,R2
```

(R2 and R3 denote those actual machine registers in the IL.) In the spirit of separation of analysis, register requirements are introduced locally. Note that values are moved into the specific registers just before use, and out of the specific registers immediately when produced. No attempt to propagate these register constraints through the program is made until register allocation is performed, at which time all such requirements are treated uniformly, regardless of origin.

Analysis for Optimization

So far, we have viewed optimization as a collection of program transformations. In fact, data gathering and global analysis is necessary to support these transformations (3). PL.8's strategy for maintaining correct global information is to recompute the information when a transformation places it in doubt. Since much of the cost of optimization is in analysis, our design represents a conscious choice of simplicity and robustness over potential performance improvement. However, our experience with optimization is that its cost is not so great that we should reconsider this decision.

Raising the Level of the IL

Some architectures contain single machine operations which are equivalent to a sequence of IL operations. To use these instructions, several IL operations must be replaced by a single operation.

The PL.8 compiler does only a limited amount of such analysis.

For example, storage to register ops replace sequences of the form:

```
L   RX,X
A   R1OO,RY,RX     Register to register
```

with:

```
A   R1OO,RY,X      Storage to register
```

This is only done if the load and add are in the same basic block and there are no more uses of RX and RY. By waiting until after optimization to insert such ops optimization has a chance to eliminate the load entirely, or to move it out of a loop even though the add must stay in the loop. A similar process inserts register to storage instructions on MC68000.

Register Allocation

One of the important simplifications of the PL.8 compiler structure is its dependence on a global register allocator. In summary, the register allocator starts with an IL program which uses an arbitrarily large number-of registers. Included in this set are the actual machine registers which where introduced in IL expansion. The register allocator collects information about the ability of pairs of these "symbolic" registers to reside in the same machine register. Two symbolic registers (or one and a machine register) interfere if at any point in the program they cannot coincide. Eventually, the graph whose nodes are registers and whose edges are interferences is colored so that nodes connected by an edge do not have the same color. Consequently, all symbolic registers (nodes) with the same color can share the machine register which was assigned that color. (6) deals with the treatment of programs which do not "color" and thus must "spill", as well as describing this process in detail.

Once the interference graph has been computed but prior to assigning registers, it is easy to perform certain heuristics or take into account machine constraints. In particular, the program is searched for indications that allocating two symbolic registers to the same real register would be profitable. For example, if one register is moved to another, that move would disappear if the two values had the same assignment. Similarly, if the target machine has destructive operations, it is profitable to assign the source and target of such operations to the same machine register. Given:

```
OP  RT,RA,RB
```

if OP only has a two address form in the target architecture and RT and RA do not interfere they should be coalesced. An attempt is also made to coalesce RB and RT if there is an interference between RT and RA and OP is commutative. (Final assembly will deal with cases which do not disappear by introducing additional register copies.) Coalescing can be done without actually preassigning RA and RB. Rather, if two registers do not interfere, one can be renamed to the other at every appearance without changing the program. If one member of the coalesce pair is a machine register, the coalesce becomes a pre-assignment. Coalescing with machine registers causes computations to use and produce values in the specific machine registers required by linkage conventions. Interference information allows simple checks for the possibility of such pre-assignment. In practice, they turn out to work extremely well.

Some machine peculiarities can also be handled by manipulating the interference graph used by register allocation. For architectures like System/370, if a symbolic register is ever used as a base or index such a computation is made to interfere with R0, which cannot be used as a base or index on that architecture. This insures it will never be assigned to R0. In architectures with typed registers, such as the MC68000, which distinguishes address and data registers, if an operation can only be performed or must be available in one type of register it is made to interfere with the other set. There are instances in which a computation must be in both sets of registers. This is discovered and move registers introduced.

Scheduling

On many machines it is profitable to move loads of data away from uses, set the condition register several instructions before a branch etc. Such rearrangement of the order of evaluation is called scheduling. By doing scheduling before register allocation the accidental constraints that register allocation would introduce are avoided. After register allocation spill code may have been inserted which should be scheduled. Therefore scheduling is done both before and after register allocation.

Final Assembly

The result of register allocation is a program still in IL format. However, every register reference is now to a machine register. Final assembly is a two pass assembler. It computes the actual values of all branch locations on the first pass, and assembles the final code on the second. Most machine dependence is isolated in tables and in final assembly. This includes instruction and data formats and such matters as the extra code needed to deal with destructive operations and program addressability. Final assembly produces the object representation of initialized static storage, and the external reference information of the

program. Optionally, it produces an assembly listing and a variable cross reference map.

Final assembly includes in the object program tables which allow run time debugging support. These tables provide a map from program location to statement number without any executable code to record the current statement number. They also contain dictionary information for symbolic debugging. Final assembly also produces a coded representation of the type of every external variable used or defined by the program. This type information supports link edit and load time type enforcement.

After register allocation some local fix up is possible. This work is also left to final assembly. A typical example is the replacement of full with half word arithmetic on the MC68000.

Final assembly must also deal with condition codes. No acceptable way to deal with machine condition codes during optimization has been found. Thus, the IL does not have condition codes. Rather, compare operations are viewed as producing values which are used as the operands of other instructions, including conditional branches. Register allocation assigns these values to the condition register when possible. Final assembly performs two local fix ups of that code. Whenever such a value is computed or used, and its assigned register is not the condition register, extra code is generated to capture the condition code in the assigned general purpose register, or to use the condition code value in the general purpose register. In addition, some compares with zero are eliminated by a basic block peephole when a previous condition code setting operation (e.g. subtract) can be seen to make the compare redundant.

Assembly must also cope with all the remaining machine peculiarities. On the System/370 a program must be divided into 4K blocks as only one register is dedicated to addressing instructions and literals. Interblock branches are indirect through a transfer vector at the end of the block. Such branches require three rather than one instruction, but in practice they occur very rarely. On machines with short displacement branches, such as the MC68000, final assembly determines whether a short branch form can be used.

Assembly inserts appropriate prolog and epilog code. This allows a choice of prolog and epilog code based on information gathered by optimization and register allocation. If all references to the local stack frame have been eliminated it is not necessary to allocate a frame. This is the case with the final System/370 code in Fig. 4. Also the compiler arranges to save and restore only those registers that linkage conventions require to be saved and that were actually altered in the procedure.

Range Checking

One of the goals of the PL.8 compiler effort was to make the enforcement of language addressing rules efficient enough to make such checking the normal case. In particular, the PL.8 translator introduces checks to guarantee that every array or offset reference is within its target array or area. To support this checking the final object code includes sequences which will trap if a bounds error occurs. However, the IL models these conditional traps as operations which produce results which are then used as operands of storage operations. In Fig. 2, the instruction TGTI, trap greater than immediate, is used. Whenever a subscripted storage reference is made such an instruction or instructions is used to guarantee that the subscript is within range. TGTI is a logical compare. As X in the example has a low bound of zero TGTI verifies that R111 is neither negative nor too large. (Negative values in twos complement are large positive numbers when a logical comparison is made.) Because traps are modeled as result producing instructions, optimization can move and eliminate traps in the same way it does other computations. The use of the trap "results" as operands of storage reference operations expresses exactly the right dependence. The trap must be evaluated before the storage reference can occur. If more than one trap is required, as would be the case for a reference to a two dimensional array, they are joined with a LIST pseudo operation, eg.

```
TGTI   R100,RI,10
TGTI   R101,RJ,10
LIST   R102,R100,R101
L      RX,X(RA)   TRAP=R102
```

Global optimization processes TRAP and LIST operations. Register allocation ignores the result of TRAP and LIST instructions, and final assembly ignores the latter entirely. It is possible to eliminate checking code. The code generator zeros the TRAP=field and the dead code elimination process eliminates all trap code as well as any computations that are only required to compute traps such as loads of bound fields from descriptors. The 801 has instructions which implement IL traps. However, System/370 has no trap instructions and the single MC68000 trap instruction is limited to a half word and does not correspond to the IL trap. By expanding traps on System/370 and MC68000 to a compare followed by a branch to the location of the compare +1, a trap is implemented in two instructions, and can be easily recognized in the interrupt handler for diagnostic purposes. (Branches to an odd address cause an addressing exception when the branch is taken.)

The result of this effort is that checked code from the PL.8 compiler is normally 5 to 10 percent slower that unchecked code. This cost allows the use of checked code in production as well as during testing. Because almost all the work of making checking code efficient comes from applying the same processes which operate on useful computation confidence in the correctness of checking code is high.

Target Machines

The PL.8 compiler produces object code for five CPU architectures. All are twos complement, byte addressable 32 bit machines, and thus implement IL computations reasonably well. All have enough registers to support the separation of register allocation from code generation and optimization. The PL.8 compiler is written in PL.8. A little over half its programs need spill code when compiled for the sixteen register System/370. If 32 registers are available as in the 801, over 95% of the compiler routines can register allocate without spill code.

There is an obvious tension between optimization, which increases the time when computations should be available in registers, and register allocation which must pack these computations into a minimal set of registers. If a target machine had many fewer than sixteen registers we suspect our compiler would over optimize and produce unacceptable amounts of spill code. Some evidence of this is given by our experience with the MC68000, which partitions its registers into eight address and eight data registers. In practice the register allocator runs out of one set while there are still available registers in the other. A few complex programs on the MC68000 take 1.5 times the code space of System/370 because of excessive register spill code. Thus our ability to efficiently support several machines is subject to some architectural restrictions. Our method of compilation biases us in favor of regular and simple register-register architectures such as the 801.

Compiler Reliability

One perennial fear about optimization is its correctness. As we have shown above, careful organization of an optimizing compiler greatly simplifies optimization. This organization also tends to encourage most bugs in the optimizing algorithms to produce catastrophic results which can thus be found by simple testing procedures. Our basic procedure is to compile and execute a test bucket of 150 self checking programs every night. The test run produces code which is executed on all computers and at various levels of optimization.

In addition, the PL.8 effort has lead to a powerful compiler testing strategy. The PL.8 compiler is written in PL.8 and compiles itself. Periodically a "fixed point" is compiled. The current source is completely recompiled using a compiler whose operation is believed to correspond to that source. The object and listing output of this compilation is saved and also used to build a new compiler. This new compiler is then used to compile the same source again. A bit for bit comparison of the object and listing files from both compiles is then made (ignoring dates, times, and other system noise). If they are identical the new version is made available to our users and as a fall back for compiler development. The fixed point process guarantees that the official compiler will be a reasonable development tool for itself. This technique seems to be a very effective check of the compiler, especially when coupled with a recompile of the runtime library for each machine. PL.8 code is highly portable from machine to machine and it is rare to find a bug on one machine that does not occur on all.

Conclusion

Global optimization and register allocation are useful tools for simplifying compiler design. They greatly reduce the need for ad hoc methods to obtain good object code and produce higher quality code than special casing. Compiler quality is enhanced because reliance is placed on methods that are separable and whose correctness can be studied in general terms. Furthermore the PL.8 compiler has demonstrated that the optimizing algorithms can be applied to a compiler that accepts multiple source languages and produces code for several different architectures. By applying global optimization to checking code, encouragement is given to the use of high level, checked languages and reliability is further enhanced.

Acknowledgments

Many people contributed to the PL.8 compiler. Greg Chaitin, Dick Goldberg, Pete Markstein, Vicky Markstein, Victor Miller, Peter Oden, Phil Owens and Hank Warren all made major contributions to its design and development. Fran Allen has provided us valuable advice and counsel over the years. John Cocke contributed many ideas, but most importantly taught us how to think about compilation.

Appendix

An Example of PL.8 Compilation

In order to view the effects of optimization on an actual program, consider the simple sort in Fig. 1. Fig. 2 shows the result of source language translation for the the exchange of values in statements 18 through 20. (Fig 3 is the translation of the rest of the source program.) The listings can be read as follows:

1) The source line that gave rise to the code is on the left.

2) Register 15 (ROF) is the base for the stack frame and is the only implicit addressability.

3) The procedure begins by storing a parameter, which is by linkage convention the location of the called program's static storage, in the (stack) variable named /.static. Static storage contains PL.8 static internal variables (Algol own). It also contains address constants to access static external data and controlled descriptors.

4) Code generation uses as many "symbolic" registers as it needs. Hence R98, R99 etc. These registers are not reused except in the case of formally identical computations. (Computations that apply the same operator to the same operands are formally identical.)

5) This example uses the following operations:

```
AI    Add immediate value
BF    Branch if condition bit specified = 0
BT    Branch if condition bit specified = 1
C     Compare two registers, result is
      a symbolic register
CI    Compare a register to an (immediate)
      literal value
L     Load from storage to a register
LI    Load immediate value
MPYI  Multiply immediate value
MR    Move register
RET   Return to caller
ST    Store
TGTI  Trap if register is logically greater
      than immediate value
```

6) .X is the address constant in static internal to access the external variable X.

7) Trap instructions produce results that are used.

Note the repetitive nature of the code. It has been produced from source language fragments without considering context.

After code motion and commoning much of the "housekeeping" code is out of loops and repetition is eliminated. (Source language computations will also be optimized but they tend to be of lesser importance.) Considering only the code to do the exchange (lines 18-20), the original 19 lines of IL code are reduced to four:

```
ST  R119,TEMP(ROF)     temp=x(j);
ST  R118,X(R99, R117)  x(j)=x(i);
L   R132,TEMP(ROF)
ST  R132,X(R99, R115)  x(i)=temp;
```

Improvement is still possible but commoning and code motion alone have reduced the number of instructions executed by more than a factor of four. Dead store elimination, load anticipation, and move register (MR) coalescing will further improve the code. After every store of the form:

```
ST  RX,A
```

A move register is inserted as follows:

```
ST  RX,A
MR  RA,RX
```

RA is the symbolic register that would have been loaded if a load from A were generated, i.e.

```
L   RA,A
```

After inserting the MR ops, commoning will eliminate any loads of RA that are reached by the move register. Dead code elimination removes the MR ops that are superfluous. Store elimination may then eliminate the store itself and those move registers that remain may be eliminated by coalescing registers. The following shows the code for the exchange after move registers have been inserted. Each instruction which can be eliminated is labelled with the optimization which acts on it.

```
ST  R119,temp(ROF)     store elimination
MR  R132,R119          register coalescing
ST  R118,X(R99,R117)
MR  R119,R118          dead code
L   R132,temp(ROF)     commoning
ST  R132,X(R99,R115)
MR  R118,R132
```

Fig. 4 shows the final source code for the entire procedure. Note that the exchange code from source lines 18-20 which was first 19 lines and then four has been reduced to:

```
ST  R01,X(R05,R0C)
ST  R02,X(R05,R0B)
MR  R01,R02
```

The final move register, which was inserted, has not been eliminated as it makes it possible to move a load of x(i) on a high frequency path out of the loop even though x(i) is altered in the loop. Thus, many simple steps, operating in an independent but pervasive manner can do as well as the best special case analysis.

Fig. 1. Source Code Example

```
 1 |
 2 |
 3 |bsort: procedure;
 4 |
 5 |   declare
 6 |      x(0:100)      fixed(31)bin static ext,
 7 |      temp          fixed(31)bin,
 8 |      (i, j)        fixed(31)bin,
 9 |      ;
10 |
11 | /* bubble sort in static external storage */
12 |
13 |   do i = 0 to hbound(x)-1;
14 |
15 |      do j = i to hbound(x);
16 |
17 |         if x(i)>x(j)  then do;
18 |            temp = x(j);     /*  exchange  */
19 |            x(j) = x(i);     /*  x(i) and  */
20 |            x(i) = temp;     /*  x(j).     */
21 |         end do;
22 |
23 |      end do j;
24 |
25 |   end do i;
26 |
27 |end proc bsort;
```

Fig. 2. Generated code to do exchange

```
18 (1)     L       R98,/.STATIC(ROF)           (2,3)
18         L       R99,.X  (R98)                 (6)
18         L       R111,J  (ROF)                 (4)
18         TGTI    R116,R111,100                 (7)
18         MPYI    R117,R111,4
18         L       R119,X(R99,R117)   TRAP=R116 (7)
18         ST      R119,TEMP(ROF)
19         L       R98,/.STATIC(ROF)
19         L       R99,.X(R98)
19         L       R102,I(ROF)
19         TGTI    R114,R102,100
19         MPYI    R115,R102,4
19         L       R98,/.STATIC(ROF)
19         L       R99,.X(R98)
19         L       R111,J(ROF)                   (4)
19         TGTI    R116,R111,100
19         MPYI    R117,R111,4
19         L       R118,X(R99,R115)   TRAP=R114
19         ST      R118,X(R99,R117)   TRAP=R116
20         L       R98,/.STATIC(ROF)
20         L       R99,.X(R98)
20         L       R102,I(ROF)
20         TGTI    R114,R102,100
20         MPYI    R115,R102,4
20         L       R132,TEMP(ROF)
20         ST      R132,X(R99,R115)   TRAP=R114
```

Fig. 3. Generated for rest of program

```
28 (1)     ST      R134,/.STATIC(ROF)
13         L       R98,/.STATIC(ROF)           (2,3)
13         L       R99,.X(R98)                   (6)
13         LI      R100,0
13         ST      R100,I(ROF)
13         L       R102,I(ROF)                   (4)
13         CI      R104,R102,99
13         BT      R104,27/GT,L4
13 L3:
15         L       R102,I(ROF)
15         MR      R108,R102
15         L       R98,/.STATIC(ROF)
15         L       R99,.X(R98)
15         ST      R108,J(ROF)
15         L       R111,J(ROF)
15         CI      R113,R111,100
15         BT      R113,27/GT,L7
15 L6:
17         L       R98,/.STATIC(ROF)
17         L       R99,.X(R98)
17         L       R102,I(ROF)                   (4)
17         TGTI    R114,R102,100                 (7)
17         MPYI    R115,R102,4
17         L       R118,X(R99,R115)   TRAP=R114 (7)
17         L       R98,/.STATIC(ROF)
17         L       R99,.X(R98)
17         L       R111,J(ROF)
17         TGTI    R116,R111,100
17         MPYI    R117,R111,4
17         L       R119,X(R99,R117)   TRAP=R116
17         C       R120,R118,R119
17         BF      R120,27/GT,L9
        (Code for lines 18-20)
21 L9:
15         L       R111,J(ROF)
15         AI      R112,R111,1
15         ST      R112,J(ROF)
15         L       R111,J(ROF)
15         CI      R113,R111,100
15         BF      R113,27/GT,L6
15 L7:
13         L       R102,I(ROF)
13         AI      R103,R102,1
13         ST      R103,I(ROF)
13         L       R102,I(ROF)
13         CI      R104,R102,99
13         BF      R104,27/GT,L3
13 L4:
28         RET
```

Fig. 4. Final System/370 object code

```
   |          ST      ROD,52(ROF)
   |          BALR    ROD,0
   |          L       R03,.X(R01)
 13|          LA      R01,0
   |          LR      R05,R01
 15|  L14:                              <---------
 15|          LR      R04,R01                     |
 15|          C       R05,=F'400'                 |
   |          BH      L7                          |
   |          CL      R01,=F'100'       PERFORMS  |
   |          BH      *-3               TGTI      |
   |          L       R02,X(R03,R05)              |
   |          SLL     R04,2                       |
   |  L13:                              <----     |
 17|          L       R00,X(R03,R04)         |    |
 17|          CR      R02,R00                |    |
 17|          BNH     L9                     |    |
 19|          ST      R02,X(R03,R04)         |    |
 20|          ST      R00,X(R03,R05)         |    |
 20|          LR      R02,R00                |    |
 21|  L9:                                    |    |
 15|          A       R04,=F'4'              |    |
 15|          C       R04,=F'400'            |    |
 15|          BNH     L13                -----    |
 15|  L7:                                        |
 13|          A       R05,=F'4'                  |
 13|          A       R01,=F'1'                  |
 13|          C       R05,=F'396'                |
 13|          BNH     L14                ---------
   |          L       ROD,52(ROF)
   |          BR      ROE
```

References

1. Allen, F.E., "Bibliography on Program Optimization," IBM Research Report RC5767, 1975.

2. Allen, F.E., et al, "The experimental compiling system", IBM Journal of Research and Development, Vol. 24, No. 6, Nov 1980, pp. 695-715

3. Allen, F.E. and Cocke, J., "A Program Data Flow Analysis Procedure," Communications of the ACM, March 1976.

4. J. L. Carter, "A case study of a new code generation technique for compilers", Communications of the ACM, Vol. 20, pp. 914-920, (1977)

5. Chaitin, G. J., et al, "Register Allocation via Coloring," Computer Languages, Vol. 6, pp. 45-57, 1981, Great Britain.

6. Chaitin, G. J., "Register Allocation and Spilling via Graph Coloring", SIGPLAN Symp. on Compiler Construction, June 23-25, 1982, Boston, Mass.

7. Cocke, J. and Markstein, P., "Measurement of Program Improvement Algorithms." Proc. IFIP Cong. '80, Tokyo, Japan Oct. 6 - 9, 1980, Melbourne, Australia Oct. 14 - 17 1980, 221-228.

8. William Harrison, "A New strategy for code generation - The general purpose optimizing compiler", Proc. Fourth ACM Symp. on Principles of Programming Languages, January, 1977, pp. 29-37

9. Markstein, V., Cocke, J., and Markstein, P., "Optimization of Range Checking," SIGPLAN Symp. on Compiler Construction, June 23-25, 1982, Boston, Mass.

10. Radin, G., "The 801 Minicomputer", Symp. on Architectural Support for Programming Languages and Operating Systems, March 1982, pp. 39-47