THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DECREASING PROCESS MEMORY REQUIREMENTS BY OVERLAPPING RUN-TIME STACK DATA

By

EMILY J. RATLIFF

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 1997

The members of the Committee approve the thesis of Emily J. Ratliff defended on February 23, 1997.

David B. Whalley
Professor Directing Thesis

Theodore P. Baker
Committee Member

Susan I. Hruska
Committee Member

Approved:

R. C. Lacher, Chair, Department of Computer Science

CHAPTER 1

INTRODUCTION

Most of the time, faced with a time/space trade-off, a compiler writer will choose to optimize time, even at the cost of space. This was not always the case. Early in the history of computers, programmers would try everything they could think of to reduce the size of their code to get it to fit in the computer's constrained space. As memory and disk space became cheaper and larger, the focus of optimizations shifted to saving time. Lately, program bloat has become normal. Programs are growing larger because people are more willing to buy programs that have extra functionality, even if they don't need the functionality. In addition, some time saving optimizations can greatly increase the space needed for a program. These optimizations include loop unrolling [1], inlining [2], scalar expansion [3], avoiding jumps [4], avoiding branches [5], and many others. This results in programs with very large executables.

Some of the early techniques to reduce process memory requirements were EQUIVALENCE statements in FORTRAN, variant records in Pascal, and unions in C. FORTRAN EQUIVALENCE statements allow the programmer to specify that two or more variables should be assigned the same address in memory. Variant records in Pascal and unions in C provide a way for the programmer to declare a variable that can have different types at different times in the program. The alignment requirements and space allocation are handled automatically by the compiler. However, there is a drawback; the programmer must keep track of the field type in the union each time it is referenced in the program. If the programmer stores the union as one field type and then loads it as a different field type, the results may be unpredictable and are machine dependent [6]. For each of the methods mentioned above, strange and subtle errors can be introduced if the programmer does not track the live ranges of the variable accurately. For the programmer maintaining this code, it adds a whole new level of complexity.

Another method often used to squeeze programs into a small space was overlaying. One portion of the program was always present in memory and would control the loading of other programs. Programmers spent much of their time dividing their programs into overlays, which were portions that never needed to be active simultaneously [7].

As processor speeds continue to increase faster than main memory and disk access times, the performance of a memory hierarchy is becoming more significant [8]. Reducing the size of a program on a machine with virtual memory can enhance paging performance. A page fault can easily require 700,000 to 6,000,000 cycles to resolve [8]. Thus, avoiding a single page fault by overlapping run-time stack data can result in a significant performance improvement. Decreasing the memory used by a process by overlapping run-time stack variables may improve data caching when the size requirements for data are diminished.

Processors are now being used in an increasing number of applications that are embedded within some other type of system. These systems frequently have no virtual memory so programs must be able to completely reside within main memory. Even the small improvement shown by non-inlined programs may tilt the balance in favor of the program fitting within the mandated space.

This thesis describes a technique for reducing the amount of memory required for a process by overlapping run-time stack data. As a quick explanation of what this optimization does, consider the simple but unrealistic program in Figure 1.

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
void main()
{
  int x, y, a, b;

  srand((int) time());                            /* Line 1 */
  y = rand() % 100;                               /* Line 2 */
  x = y * y;                                      /* Line 3 */
  printf("The square of %d is %d.\n", y, x);    /* Line 4 */

  a = rand() % 500;                               /* Line 5 */
  b = a * a;                                      /* Line 6 */
  printf("The square of %d is %d.\n", a, b);    /* Line 7 */

}
```

Figure 1: A Simple Program

A variable is said to be live from the point that it is first assigned a value to the point where it is last used. The extent of the program that the variable is live is called the live range of that variable. The live ranges of the variables depicted by the source lines[1] in the program in Figure 1 are as follows:

1. `x` Lines 3-4
2. `y` Lines 2-4
3. `a` Lines 5-7
4. `b` Lines 6-7

One can see that the variables `x` and `y` and the variables `a` and `b` have conflicting live ranges, but neither `x` nor `y` conflicts with `a` and `b`. This can be graphically depicted as in

[1] Live ranges are actually ranges of machine instructions in our compiler.

Figure 2, where each edge represents a conflict.

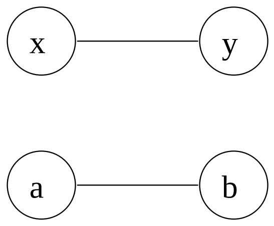


Figure 2: Conflict Graph

Since the live ranges of x and a, x and b, y and a, y and b do not conflict (that is, the variables are not live at the same time), any of these variables could be given overlapping locations on the stack. In the simple case with no register allocation and no overlapping, the variables would have been assigned offsets as follows:

x = 92

y = 96

a = 100

b = 104

With overlapping turned on and register allocation still off, the variables would have been assigned the following offsets:

x = 92

a = 92

y = 96

b = 96

In the first case the activation record would be 112 bytes, whereas in the second case the activation record would be 104 bytes, for a savings of 8 bytes.[1]

In reality, in the simple program of Figure 1, the programmer would have reused the first two variables, rather than declaring two new variables. Also, all four of the variables would have been assigned to registers, leaving no room for improvement via overlapping run-time stack data. However, the example is a simple way to understand what the optimization can accomplish. Since programs are not often as simple as in Figure 1, it is easier to automate the reusing of space rather than to rely on the programmer to recognize every opportunity to reuse stack space. It is especially better to automate the process than to rely on the programmer to correctly identify and implement the overlapping opportunities. The programmer may fail to take advantage of opportunities or fail to implement the opportunities properly. For example, programmers rarely make use of unions. Relying on the programmer to force two or more variables into one may make the program less readable. Automating the overlapping of variables supports the software engineering design principle of using descriptive variable names. When overlapping was applied with standard optimizations, not much improvement was noted. However, when applied with inlining a good improvement was seen. This is because inlining greatly expanded the opportunities for overlapping.

[1] The size of a SPARC activation record must be an integer multiple of 8. The first 92 bytes are required to support register windows and other static information for the SPARC calling sequence. More details about the structure of a SPARC activation record will be given later.

CHAPTER 2

RELATED WORK

While most optimizations focus on saving time, there are some that concentrate on saving space. The optimization described in this thesis is the first work that contains a general algorithm for reducing run-time stack data.

Several algorithms exist for reducing instruction space. Code hoisting moves identical instructions from multiple blocks in different paths to a single dominating block [9]. Cross jumping moves identical instructions from multiple blocks in different paths to a single post-dominating block [10]. Fraser *et. al.* [11] achieve a 7% decrease in the number of static instructions by applying a general text compression algorithm to assembly code. However, the number of instructions executed typically increased. Liao *et. al.* [12] developed techniques to decrease the number of instructions in programs compiled for DSP architectures that only allow auto-increment and auto-decrement modes for accessing memory. The main goal was to decrease the dynamic number of instructions required to access memory, but had the side effect of compressing code. Bowman [13] describe a method over overlapping instructions using algorithms to implement cross-jumping and abstracting relocatable code portions. Using this approach, minimal or no dynamic instruction increases were observed.

Wolfe [14] describes techniques for contracting arrays into scalar variables. This optimization is typically only possible after previously expanding a scalar variable into an array to support vectorizing optimizations. Ramsey [15] reduced the size of object-code files by abstracting common relocation information to support more efficient and machine-independent linking. The process memory requirements of the compiled programs were not affected. In the technical report mentioned above, Bowman *et. al.* achieved a 7% reduction in process size by overlapping static data with other static data using a method similar to the one described in this thesis. Bowman *et. al.* further report a 23% savings in code space when overlapping static data with program instructions.

CHAPTER 3

OVERVIEW AND ENVIRONMENT

Overlapping run-time stack data was implemented as an extension to a optimizing compiler called VPO (Very Portable Optimizer) [16] that compiles programs for the SPARC architecture. C programs are fed into the front end, VPCC (Very Portable C Compiler) [17]. Figure 3 contains an overview of the interaction between VPCC and VPO.

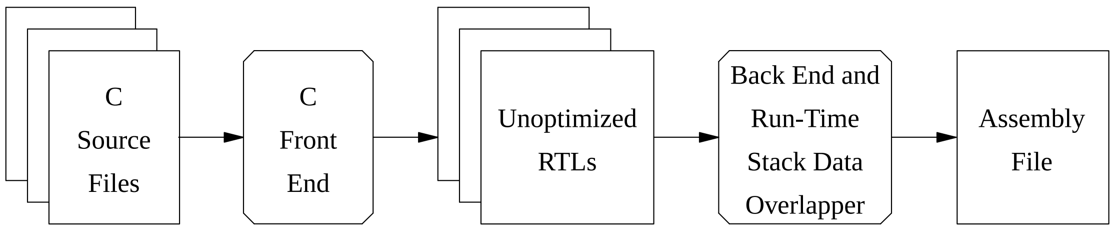


Figure 3: Environment

VPCC performs no optimizations. It translates the C code into machine independent intermediate code. This intermediate code is then used by VPO to perform the optimizations and produce the executable machine code. The code generator in VPCC translates the intermediate code into register transfer lists (RTLs) [18]. All optimizations occur on RTLs. The format of the RTLs is machine independent, but the RTLs can be checked after each transformation to ensure that they represent a valid operation on the machine for which the code is being compiled. Each RTL has the following format:

```
{dst = src;} dead register list
```

where there can be one or more `dst = src;` transfers depending upon the number of effects per machine instruction for the particular machine. The `dst` can be a register or memory reference, while the `src` can be any expression which is permissible for that particular machine.

VPO optimizes programs one function at a time. Figure 4 contains the order of optimizations for VPO. The optimization described in this thesis takes place during the

```
branch chaining
useless jump elimination
dead code elimination
eliminate unconditional jumps by reordering code
instruction selection
evaluation order determination
global instruction selection
register assignment
jump minimization
instruction selection
eliminating jumps in loops
do {
   register allocation
   instruction selection
   common subexpression elimination
   dead variable elimination
   code motion
   recurrences
   strength reduction
   induction variable elimination
   useless jump elimination
   cheaper instruction replacement
   instruction selection
   }
while change;
setup entry and exit
instruction scheduling
fill slots
useless jumps
```

Figure 4: Order of Compiler Optimizations in VPO

setup entry and exit phase. This phase has the responsibility of adjusting the entry and exits of a function to conform to the calling sequence conventions of the machine. The actions in this phase include generating code to allocate/deallocate space on the run-time stack for the activation record, saving/restoring registers, and assigning offsets of local variables.

The stack locations for the run-time stack data are assigned during the setup entry and exit phase. As one can see, almost all of the optimizations have been performed before the stack location of the run-time stack data is considered. VPO previously assigned the stack location to the run-time stack data by assigning locations in decreasing order of size (largest data first). This had the simple appeal of minimizing the amount of padding needed for correct alignment. Otherwise, the order of local variables and temporaries in an activation record were considered unimportant.

A SPARC activation record has the format illustrated in Figure 5. Activation records

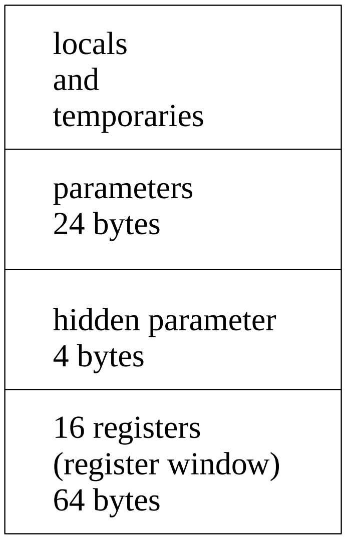


Figure 5: Structure of a SPARC Activation Record

are aligned on an eight byte boundary. The size of an activation record must be an integer multiple of eight. This preserves alignment on an 8-byte boundary and allows the compiler to ensure that local variables are properly aligned to prevent misalignment exceptions.

CHAPTER 4

OVERLAPPING RUN-TIME STACK VARIABLES

Overlapping run-time stack data requires the following steps: 1) determine if the variables were used indirectly, 2) if a variable is used indirectly, determine where it is referenced, 3) determine the live range of each variable, 4) build an interference graph based on the live range of each variable, 5) establish the order in which the live ranges of the variables will be assigned offsets to achieve maximum overlap with minimum alignment padding penalty, and 6) assign an offset for each live range of a variable in such a way that it does not overlap with any variable that has a conflicting live range. Figure 6 contains the C source code for the example that will be used to illustrate the process of overlapping run-time stack data. Figure 7 displays the resulting RTLs with no register allocation performed. The control flow of the program is also represented.

If the address of a variable is taken but not immediately used, the variable is said to have been used indirectly. Arrays are by definition indirectly referenced. Detection of accurate live ranges of non-scalar variables is much more difficult since the range of elements accessed and the loops driving the induction variable(s) associated with the memory reference must be known [19]. To determine which variables are being referenced indirectly, each RTL in the function is examined to detect 1) if it references a variable, and 2) whether the reference is inside a memory reference. If a variable is being referenced outside of a memory reference, it is being used indirectly and is

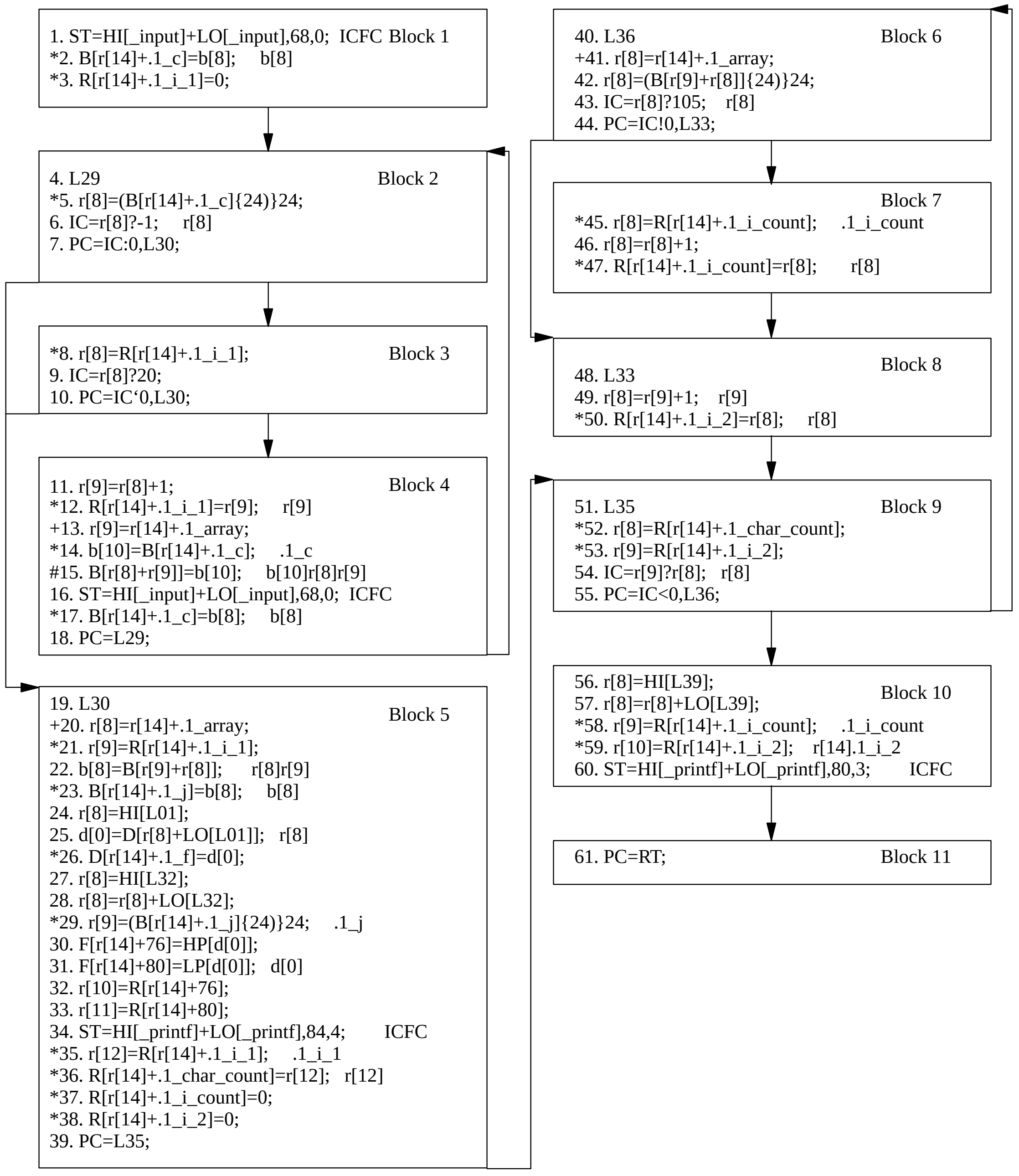
```
void main()
{
   char array[20], c, j;
   int char_count, i, i_count;
   double f;

   c = input();
   i = 0;
   while ((c != EOF) && (i < 20))
     {
      array[i++] = c;
      c = input();
     }
   j = array[i];
   f = 1.0;
   printf("The last character is %c. f is %f\n", j, f);
   char_count = i;
   i_count = 0;
   for (i = 0; i < char_count; i++)
      if (array[i] == ’i’)
        i_count++;
   printf("There are %d i’s out of %d chars\n", i_count, i);
}
```

Figure 6: An Example Function

marked. An example of an RTL where a variable (`array`) is being used indirectly is RTL number 15 in Figure 7.

Scalar variables that are not used indirectly can have more than one live range. Each live range of a scalar variable is from the set(s) of the variable to the last use(s) before the next set(s) or the end of the function. Similar analysis to determine the live ranges of scalar variables is performed during register allocation.

Because of the difficulty in determining the exact live ranges of indirectly used variables, one large live range was calculated for these variables. This one live range is the extent from its first reference(s) to its last reference(s). This live range was calculated by intersecting the basic blocks that precede the references to the variable with the blocks that can follow the references. All of the blocks containing the references are



```
1. ST=HI[_input]+LO[_input],68,0;  ICFC Block 1
*2. B[r[14]+.1_c]=b[8];     b[8]
*3. R[r[14]+.1_i_1]=0;
```

```
4. L29                                    Block 2
*5. r[8]=(B[r[14]+.1_c]{24)}24;
6. IC=r[8]?-1;     r[8]
7. PC=IC:0,L30;
```

```
*8. r[8]=R[r[14]+.1_i_1];                 Block 3
9. IC=r[8]?20;
10. PC=IC‘0,L30;
```

```
11. r[9]=r[8]+1;                          Block 4
*12. R[r[14]+.1_i_1]=r[9];     r[9]
+13. r[9]=r[14]+.1_array;
*14. b[10]=B[r[14]+.1_c];     .1_c
#15. B[r[8]+r[9]]=b[10];     b[10]r[8]r[9]
16. ST=HI[_input]+LO[_input],68,0;  ICFC
*17. B[r[14]+.1_c]=b[8];     b[8]
18. PC=L29;
```

```
19. L30
+20. r[8]=r[14]+.1_array;                 Block 5
*21. r[9]=R[r[14]+.1_i_1];
22. b[8]=B[r[9]+r[8]];     r[8]r[9]
*23. B[r[14]+.1_j]=b[8];     b[8]
24. r[8]=HI[L01];
25. d[0]=D[r[8]+LO[L01]];   r[8]
*26. D[r[14]+.1_f]=d[0];
27. r[8]=HI[L32];
28. r[8]=r[8]+LO[L32];
*29. r[9]=(B[r[14]+.1_j]{24)}24;     .1_j
30. F[r[14]+76]=HP[d[0]];
31. F[r[14]+80]=LP[d[0]];   d[0]
32. r[10]=R[r[14]+76];
33. r[11]=R[r[14]+80];
34. ST=HI[_printf]+LO[_printf],84,4;        ICFC
*35. r[12]=R[r[14]+.1_i_1];     .1_i_1
*36. R[r[14]+.1_char_count]=r[12];   r[12]
*37. R[r[14]+.1_i_count]=0;
*38. R[r[14]+.1_i_2]=0;
39. PC=L35;
```

```
40. L36                                   Block 6
+41. r[8]=r[14]+.1_array;
42. r[8]=(B[r[9]+r[8]]{24)}24;
43. IC=r[8]?105;    r[8]
44. PC=IC!0,L33;
```

```
                                          Block 7
*45. r[8]=R[r[14]+.1_i_count];     .1_i_count
46. r[8]=r[8]+1;
*47. R[r[14]+.1_i_count]=r[8];       r[8]
```

```
                                          Block 8
48. L33
49. r[8]=r[9]+1;    r[9]
*50. R[r[14]+.1_i_2]=r[8];     r[8]
```

```
51. L35                                   Block 9
*52. r[8]=R[r[14]+.1_char_count];
*53. r[9]=R[r[14]+.1_i_2];
54. IC=r[9]?r[8];   r[8]
55. PC=IC<0,L36;
```

```
56. r[8]=HI[L39];
57. r[8]=r[8]+LO[L39];                    Block 10
*58. r[9]=R[r[14]+.1_i_count];     .1_i_count
*59. r[10]=R[r[14]+.1_i_2];    r[14].1_i_2
60. ST=HI[_printf]+LO[_printf],80,3;       ICFC
```

```
61. PC=RT;                                Block 11
```

* - variable referenced directly
+ - variable’s address taken indirectly
# - variable referenced indirectly

Figure 7: RTLs for the Example Function

included in this one live range. In the example function, the variable `array` is the only variable that has its live range calculated in this fashion. The variable `array` initially has blocks 4 (RTL 13), 5 (RTL 20), and 6 (RTL 41) marked as referencing it. The dataflow is traced forward (resulting in the marking of blocks 2-12) and back (resulting in the marking of blocks 1-10) and the results are intersected. This results in a live range that spans from block 2 through block 10.

Before calculating the extent of an indirectly referenced variable, the compiler first must determine where the variable's address is actually referenced. This is accomplished by using a demand-driven approach rather than an exhaustive solution. At each point where the address of run-time stack data is taken indirectly, the compiler recursively searches forward marking all memory references that use the address. This is an effective approach because we found that in general the distance between taking the address of a local variable and the points where the address is dereferenced are typically close.

If the compiler detects that the address itself is stored into memory, then the point of the store and the return points in the function are marked as references since pointer analysis is not performed. The same action occurs if the address is passed to a function since inter-procedural analysis is not performed. In both cases this action causes the live range of the indirectly referenced variable to span from the point of the store or the call to the end of the function. Note that such a variable may still be overlapped with another variable that is only referenced before the store or call.

Once the live ranges of the variables are determined, an interference graph is built. This is similar to the interference graph that VPO uses for register allocation. The range of RTLs for each live range is compared with every other live range to determine if they conflict (that is, if they have RTLs in common). By definition, different live ranges of the same variable cannot overlap. Each live range of a variable is numbered, in effect creating a different variable name for each live range. The interference graph for the example function is depicted in Figure 8.

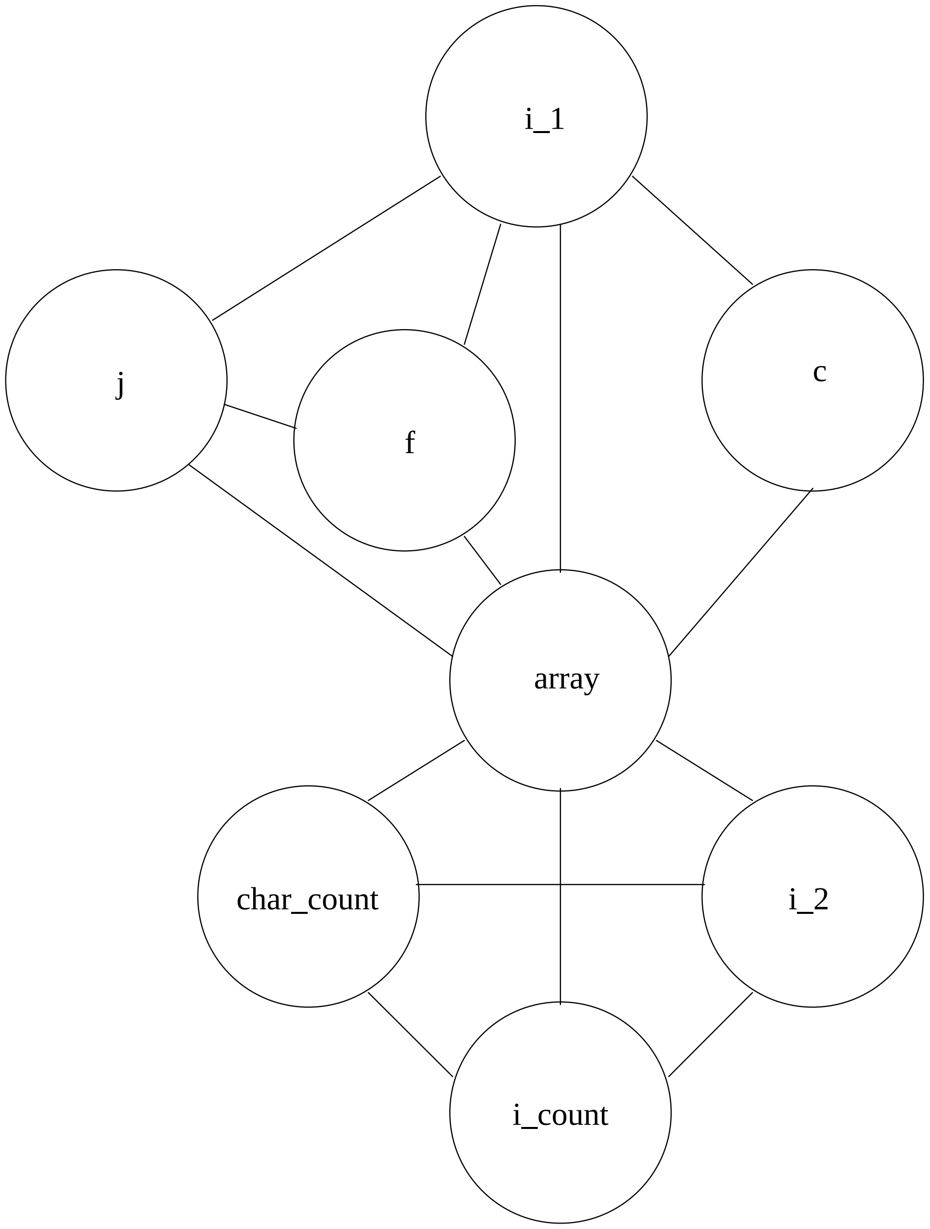


Figure 8: Conflict Graph for Example Program

The next step is to sort the live ranges to allow for the best overlap while reducing the alignment padding penalty. Several versions of a similar scheme were tried. Chaitin *et. al.*, found that assigning live ranges with the greatest conflict level first was the best heuristic to use when allocating live ranges to registers [20]. The rationale for using this scheme for overlapping run-time stack data is that if a live range conflicts with most of the other live ranges in the function, then assigning its offset early within the activation will give it the best chance of being overlapped with the few live ranges with which it does not have any conflicts. So, this scheme was tried first. However, padding is introduced to ensure that alignment requirements are met. (For the SPARC, floats and doubles must be aligned on an 8-byte boundary. In VPO, structures and arrays are aligned on 8-byte boundaries for simplicity.) When there was little opportunity for overlapping data this heuristic often caused the size of the activation record to be larger than it would have been if this optimization were not enabled, because of the extra padding required. Thus, this heuristic was rejected. The next approach was to sort the live ranges by size of the variable, with the live ranges of variables of the same size sorted by conflict level. This completely solved the problem of some activation records being larger after the optimization was applied to the programs in the test set. Upon further consideration, it was apparent that size was less important than alignment requirements. Now, the live ranges are sorted first by their alignment requirement, then by conflict level.

The final step is to assign the offset. For ease, offsets of live ranges were first assigned as relative offsets to each other and then the existing procedure for assigning exact offsets was used. Variables with the same offset are assigned the same address.

Figure 9 shows the algorithm used to assign live ranges of run-time stack data to

```
WHILE any live ranges left to assign DO
    curr_lr := live range not yet assigned with biggest alignment requirement and highest conflict level;
    curr_lr->offset := first offset where locals can be assigned;
    FOR lr := each live range in function DO
        IF (lr->status = assigned) AND (curr_lr IN lr->conflicts) AND
           does_overlap(lr, curr_lr) THEN
            curr_lr->offset := lr->offset + lr->size;
            curr_lr->offset := curr_lr->offset + necessary alignment padding
        curr_lr->status := assigned
```

Figure 9: Run-Time Stack Live Range Offset Assignment Algorithm

offsets. The current live range of a variable is assigned to the first offset that does not overlap with any previously assigned live ranges conflicting with the current live range.

Table 1 lists all the information needed to assign the offsets for the variables of the example function in Figure 6. It shows the resulting offsets with and without this optimization applied. The variables are listed in the order that offsets are assigned. The number of conflicts was difficult to determine quickly, due to the internal representation of the conflicts. Each variable had a structure for each conflict for each block that they conflicted. The total number of conflicts were counted. This discriminates against variables that are live across few blocks, as `j` in this example. The variables `f` and

Table 1: Live Range Information for Example Function

Number	Live Range Name	Live Range	Conflicts With	Size in Bytes	Offset	Offset Without Overlapping
1	array	2-10	2,3,4,5,6,7,8	20	96	96
2	f	5.26	1,6,8	8	120	136
3	i_count	5.37-10.58	1,4,5	4	92	132
4	i_2	5.38-10.59	1,3,5	4	116	128
5	char_count	5.36-9.52	1,3,4	4	120	124
6	i_1	1.2-5.39	1,2,7,8	4	92	128
7	c	1.2-4.14	1,6	1	116	116
8	j	5.23-5.39	1,2,6	1	116	120

`array` must be aligned on an 8-byte boundary; `c`, `j`, `char_count`, `i_count`, `i_1`, and `i_2` all must be aligned on a 4-byte boundary. Thus, `f` and `array` have the greatest alignment requirement. The variable `array` was the first to be assigned an offset and it must be aligned on an 8-byte boundary, so it received the offset of 96. The variable `f` conflicts with `array`, so it was assigned the first offset past the end of `array` that was on an 8-byte boundary. This turned out to be 120. The variable `i_count` needs only to be aligned on a 4-byte boundary and thus fits at offset 92. The variable `i_2` conflicts with `i_count` and `array`. It can be overlapped with `f`, but there is unallocated space between 116 and 120, so its offset is 116. The variable `char_count` conflicts with `i_count`, `array` and `i_2`, but not `f`, so it gets placed at 120. The variable `i_1` can overlap with `i_count` at 92. The variables `c` and `j` both conflict with `i_1` and `array`, but not each other, nor `i_2`, so they are assigned the offset of 116. The resulting space utilization can be seen graphically in Figure 10.

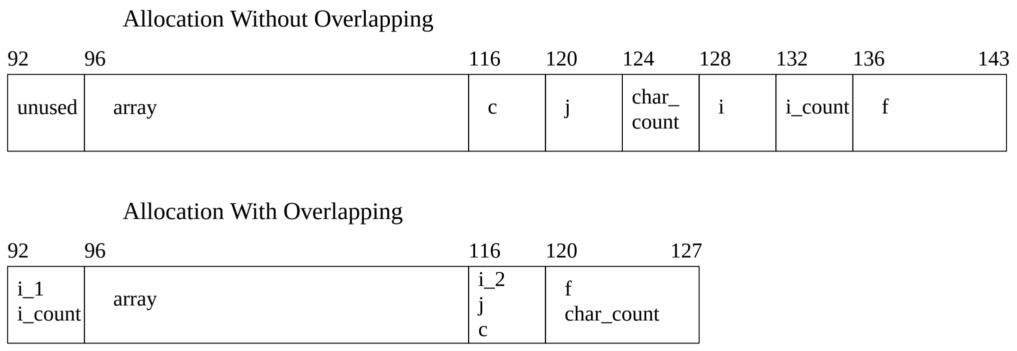


Figure 10: Offset Allocation

CHAPTER 5

RESULTS

Table 2 shows the results of overlapping run-time stack data on 19 test programs. The programs consist of various benchmarks, Unix utilities, and application programs. The number of bytes shown for the stack size is obtained by simply calculating the sum of the sizes of the different activation records as opposed to measuring the space used for the run-time stack at execution time. The number of bytes required for library functions is not included because library functions are dynamically linked.

The table also shows the results when overlapping run-time stack data was combined with inlining. Inlining was accomplished by modifying an existing inliner within VPCC, which only performed inlining within a single file of a compiled program. The new inliner processes all of the files of intermediate code produced from every source file in the program, resolves conflicting labels between the files, and removes functions that are no longer referenced. Notice that the size of the run-time stack data changed after inlining. Sometimes the size was decreased due to fewer activation records required. Sometimes it was increased due to multiple inlined copies of functions each requiring a copy of their variables.

The results show very little improvement when inlining is not invoked. This is because without inlining there were very few opportunities for improvement. There were

Table 2: Results of Overlapping Run-time Stack Data

Program	Description	Without Inlining			With Inlining		
		Stack Size	Bytes Saved	Percent Decrease	Stack Size	Bytes Saved	Percent Decrease
ackerman	Synthetic Benchmark	312	8	2.56%	232	8	3.45%
bubblesort	Synthetic Benchmark	568	8	1.41%	136	8	5.88%
cal	Calendar Generator	384	0	0.00%	96	0	0.00%
cmp	Comparison of Two Files	768	0	0.00%	192	0	0.00%
csplit	Split a File	1488	0	0.00%	728	0	0.00%
ctags	C Tags Generator	8144	0	0.00%	24544	88	0.36%
dhrystone	Synthetic Benchmark	664	0	0.00%	200	8	4.00%
grep	Pattern Search	592	0	0.00%	304	0	0.00%
join	Relational Join on Files	480	0	0.00%	96	0	0.00%
lex	Scanner Generator	9472	0	0.00%	7208	8	0.11%
linpack	Linear Algebra Routines	1504	48	3.19%	3312	112	3.38%
mincost	VLSI Circuit Partitioning	1216	0	0.00%	192	8	4.17%
prof	Profile a Program	1584	0	0.00%	400	40	10.00%
sdiff	Side-by-Side File Display	2536	0	0.00%	5784	16	0.28%
spline	Smooth Curves	560	8	1.43%	200	8	4.00%
tr	Translate Characters	192	0	0.00%	96	0	0.00%
tsp	Traveling Sales Person	3008	8	0.27%	2216	56	2.53%
whetstone	Arithmetic Operations	568	0	0.00%	488	296	60.66%
yacc	Parser Generator	4232	0	0.00%	1360	8	0.59%
average		1989	4	0.47%	2510	34	5.23%

290 total functions in the 19 test programs. Of these functions, 51 were leaf functions, 186 required no stack space for run-time variables, and 10 required only 8 bytes for run-time stack data. This left only 43 functions where overlapping could be attempted. 12 of those 43 functions had a stack size of only 112, meaning that the maximum overlap potential for those functions is only 8 bytes. Unless the programmer uses many local variables, this optimization is not worth invoking when inlining is not also invoked.

After inlining, 69 functions remained. (Twenty of the functions were in lex). Of these, 46 provide no opportunity for overlapping. The remaining functions generally provided good opportunities for overlap. The reason that inlining increases the opportunities for overlap is because when a function is inlined, it brings its local variables with it. The greater the number of local variables, the more likely it is that some of them will be overlappable. Whether the inlined functions were nested or serial also impacts upon overlapping opportunities. Functions that were nested provide less opportunity for overlap, because there is a higher probability that the live ranges will conflict. Upon examining cases where little overlap occurred in large functions, such as main() in sdiff, it became clear that with more exact calculation of the live range of arrays, the size of the stack could be reduced by as much as 75%. In no case did overlapping run-time stack data ever result in a larger activation record size.

The fact that the optimization was implemented on a SPARC impacted the percentage of improvement. The SPARC typically requires 92 bytes for state information for each activation record. Most machines require less space overhead for activation records.

CHAPTER 6

FUTURE WORK

There are many avenues open for continuing investigation in overlapping run-time stack data. Calculating the exact live range of indirectly referenced variables could extract large gains, especially when combined with inlining. Inlining created the opportunities for greater overlapping. Combining overlapping with other optimizations that increase the data size of a program should be tested for potential benefits.

The ordering of live ranges was based solely on alignment requirements and conflict level. Many heuristics have been proposed for register-coloring algorithms. These heuristics should be tested for their applicability to overlapping run-time data. A different heuristic may provide greater gains.

The decrease in process memory requirements was measured; however, a more detailed analysis of the effect that overlapping run-time stack data has on performance is needed. Only static run-time stack measurements were obtained. Recursive programs may use much more stack space. It is also unclear how the reduction in stack size will impact data caching and paging performance. It is unlikely that the small size decreases reported will have a significant impact.

A technique to promote global variables to local variables was considered but rejected because it would interfere with combining overlapping run-time stack data with overlapping static data. Promoting global variables could be included as an option when the optimization to overlap run-time stack data is used but the optimization to overlap static data is not.

CHAPTER 7

CONCLUSION

The thesis presents a compiler optimization for reducing process memory requirements by overlapping run-time stack data. Each variable is examined to determine if it is ever used indirectly. The live ranges of each variable are then determined. Variables that have been used indirectly are assigned one large live range. Based on the live ranges, an interference graph is created. Each live range is assigned an offset in such a way that no conflicting live range overlaps with it, but non-conflicting live ranges share overlapping locations in memory. Analysis of this technique shows that there are very few opportunities for overlapping with traditional optimizations applied. Programmers who use many local variables will benefit, but the test suite showed that most programmers rely on global variables or only need a small enough number of variables that all fit within a SPARC register window. Inlining increased overlap opportunities greatly. A 5% decrease in run-time stack memory requirements was noted when inlining was combined with overlapping run-time stack data.

REFERENCES

[1] J. W. Davidson and S. Jinturkar, “Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler,” *Proceedings of Compiler Construction Conference,* pp. 59-73 (April 1996).

[2] J. Davidson and A. Holler, “A Study of a C Function Inliner,” *Software—Practice & Experience* **18**(8) pp. 775-790 (August 1988).

[3] D. A. Padua, D. J. Kuck, and D. Lawrie, “High-Speed Multiprocessors and Compilation Techniques,” *IEEE Transactions on Computers* **29**(9) pp. 763-776 (September 1980).

[4] F. Mueller and D. B. Whalley, “Avoiding Unconditional Jumps by Code Replication,” *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation,* pp. 322-330 (June 1992).

[5] F. Mueller and D. B. Whalley, “Avoiding Conditional Branches by Code Replication,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation,* pp. 56-66 (June 1995).

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, NJ (1978).

[7] A. S. Tanenbaum, *Modern Operating Systems,* Prentice Hall, Englewood Cliffs, NJ (1992).

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition,* Morgan Kaufmann, San Francisco, CA (1996).

[9] A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, MA (1977).

[10] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler,* American Elsevier, New York, NY (1975).

[11] C. W. Fraser, E. W. Myers, and A. L. Wendt, “Analyzing and Compressing Assembly Code,” *Proceedings of the SIGPLAN ’84 Symposium on Compiler Construction,* pp. 117-121 (June 1984).

[12] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, “Storage Assignment to Decrease Code Size,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation,* pp. 186-195 (June 1995).

[13] Richard L. Bowman, Emily J. Ratliff, and David B. Whalley, “Decreasing Process Size by Overlapping Program Portions,” Technical Report, Florida State

University, Tallahassee, Florida (January 1997).

[14] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers,* MIT Press, Cambridge, MA (1989).

[15] N. Ramsey, “Relocating Machine Instructions by Currying,” *Proceedings of the SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pp. 226-236 (May 1996).

[16] M. E. Benitez and J. W. Davidson, “A Portable Global Optimizer and Linker,” *Proceedings of the SIGPLAN ’88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[17] J. W. Davidson and D. B. Whalley, “Quick Compilers Using Peephole Optimizations,” *Software—Practice & Experience* **19**(1) pp. 195-203 (January 1989).

[18] M. E. Benitez, “Register Transfer Standard,” Computer Science Report No. RM-91-01, University of Virginia, Charlottesville, Virginia (March 1991).

[19] E. Duesterwald, R. Gupta, and M. Soffa, “A Practical Data Flow Framework for Array Reference Analysis and Its Use in Optimizations,” *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, pp. 68-77 (June 1993).

[20] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register Allocation via Coloring,” *Computer Languages* **6**(1) pp. 47-57 (1981).

BIOGRAPHICAL SKETCH

Emily Ratliff was born in Lexington, Kentucky in March 1970. In 1992, she received her Bachelor of Science from the College of William and Mary (including a junior year abroad in Muenster, Germany) with a double major in Computer Science and German. She subsequently went to Munich, Germany for an internship with BMW. Upon completion of the internship, she went to Florida State University to obtain her Master of Science. With that goal accomplished, she now plans to re-enter industry.

ACKNOWLEDGEMENTS

I wish to thank my major professor, Dr. David Whalley, for his incredible patience and support. I would like to thank my parents, Don and Sherry Ratliff for the love, guidance, and opportunities they have given me over the years. I would like to thank my sisters Heidi Ratliff and Elizabeth Congdon for the laughter and encouragement. I would like to thank Wayne E. Sprague for his editorial and personal support. And finally, I would like to thank my committee members Dr. Hruska and Dr. Baker for their valuable assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Most compiler optimizations focus on saving time, sometimes at the expense of increasing size. Processor speeds continue to increase at a faster rate than main memory and disk access times. Reducing the size of a program may result in improved memory hierarchy performance. This thesis describes a way to reduce process size and memory requirements by automatically overlapping run-time stack data. To overlap run-time stack data, the live ranges of each overlappable variable was computed. An interference graph was produced to detect conflicts among the live ranges. The live ranges were sorted according to their alignment requirement and number of conflicts. Each live range was then assigned a stack offset. If two or more live ranges do not conflict, they can be assigned offsets in such a way that they are overlapped in the activation record. The results show very little improvement when the optimization is used alone, and significant improvements when combined with inlining.