Decreasing Process Memory Requirements by Overlapping Program Portions

Richard L. Bowman
Harris, Melbourne
bowman@harris.com

Emily J. Ratliff
IBM, Fort Lauderdale
emilyr@us.ibm.com

David B. Whalley
Florida State Univ., CS Dept
whalley@cs.fsu.edu

Abstract

Most compiler optimizations focus on saving time and sometimes occur at the expense of increasing size. Yet processor speeds continue to increase at a faster rate than main memory and disk access times. Processors are now frequently being used in embedded systems that often have strict limitations on the size of programs it can execute. Also, reducing the size of a program may result in improved memory hierarchy performance. This paper describes general techniques for decreasing the memory requirements for a process by automatically overlapping portions of a program. Live range analysis, similar to the analysis used for allocating variables to registers, is used to determine which program portions conflict. Nonconflicting portions are assigned overlapping memory locations. The results show an average decrease of over 10% in process size for a variety of programs with minimal or no dynamic instruction increases.

1. Introduction

When computers were first developed there was a strong emphasis on minimizing the amount of storage needed for a program. This emphasis is reflected in the design of the FORTRAN EQUIVALENCE declaration, which allows a programmer to specify that two or more variables be assigned the same address in memory. Subtle errors could be easily introduced if the programmer did not realize that the variables specified in an EQUIVALENCE statement could be in use at the same time. Overlays were also used as a technique to overcome the limited size of main memories. Programmers spent much of their time dividing their program into overlays, which were portions that never needed to be active simultaneously. With the advent of larger physical memories and virtual memory, the necessity for economizing the use of data memory was mollified.

Currently the assignment of variables to memory locations is of little concern to most compiler writers. After most optimizations have been performed, a compiler will update the prologue and epilogue of a function to manage space on the run-time stack. Static data is also typically arranged in the order in which the declarations are encountered. Besides ensuring that alignment requirements for the machine are met, the actual locations assigned are considered unimportant.

However, processors are now also being used in an increasing number of applications that are often embedded within some other type of system. These systems frequently do not have virtual memory and have to be able to completely reside within main memory. Thus, embedded systems often have strict limitations on the size of the programs that they can execute. Even if significant performance gains are not achieved, general techniques for reducing the size of a program may be quite useful for embedded system applications.

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 [1]. Thus, avoiding a single page fault by overlapping program portions can result in a significant performance improvement. Furthermore, decreasing the memory used by a processor may improve data and instruction caching performance when the size requirements for data and code are diminished. Secondary caches are often unified and their performance may also benefit from overlapping data and instructions.

An astute programmer may realize that certain variables are never used at the same time. The programmer may reuse the same variable for multiple purposes requiring the same data type. If different types are required, then one may use a mechanism supported by the semantics of the programming language (e.g. a union in C). Declaring one name for a variable that is used for different purposes violates the software engineering principle of using descriptive variable names. Manually overlapping variables in memory is error prone and difficult to maintain.

This paper describes a set of techniques for decreasing process memory requirements by automatically overlapping program portions. Memory on the run-time stack is compressed by overlapping local data. The static data area is compacted by overlapping uninitialized global variables with other static data. The code area is compressed by overlapping instructions via cross jumping and abstractions of code portions. Overlapping uninitialized global variables with relocatable portions of code is also performed. Applying these techniques resulted in significant reductions in process memory requirements.

2. Related Work

While some compiler optimizations do save space, most optimizations are performed with the goal of

reducing time. In fact, several optimizations typically increase the size of a program, which include function inlining, loop unrolling, scalar expansion [2], and avoiding jumps [3] and branches [4]. Techniques that compress the size of a program without causing more instructions to execute would be appealing since the increase in process memory requirements from performing these space-increasing optimizations could be partially offset.

There have been a few optimizations designed to save space. Code hoisting moves identical instructions from multiple blocks in different paths to a single dominating block [5]. Cross jumping moves identical instructions from multiple blocks in different paths to a single post-dominating block [6]. Fraser *et. al.* [7] applied a general text compression algorithm to assembly code. They reported an average 7% decrease in the number of static instructions. Their technique did not use any data flow information and hence required that each common sequence of instructions be contiguous. They also abstracted segments of code using call and return instructions. While this abstraction resulted in code size decreases, the number of instructions executed was typically increased. In contrast, the techniques we apply in this paper for overlapping program portions rarely results in a dynamic instruction increase. Some compilers with limited code motion transformations overlap variables declared in nonconflicting blocks [8]. Ramsey [9] 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.

3. Overview

Figure 1 gives an overview of the environment that was used to compress the memory requirements of a process. The C front end, called *vpcc* [10], expands intermediate code operations into unoptimized RTLs (register transfer lists) that represent machine instructions. The code expander portion of the front end was modified to produce static data directives instead of assembly code for the static data declarations. These directives include the size, alignment requirements, storage class, and initial values for each declaration. The unoptimized RTLs and directives are input to a compiler back end, called *vpo* [11], that performs conventional compiler optimizations. This back end was modified to overlap run-time stack data at the point that the entry and exits of the function are updated to manage the run-time stack, which occurs after most optimizations have been performed. The back end was also modified to produce encoded optimized RTLs and static data directives as output to a file. All of the information from these files for a compiled program was read into memory by a modified version of the back end and a call graph was constructed. Labels were adjusted from each of the files to ensure their uniqueness. Intraprocedural analysis was then performed to overlap uninitialized static data with static data, instructions with instructions, and uninitialized static data with instructions. The static data overlapper was not invoked when indirect calls are encountered since an explicit call graph was needed for accurate interprocedural analysis. Finally, assembly was generated into one file from the RTLs and data directives, which can then be linked and executed.

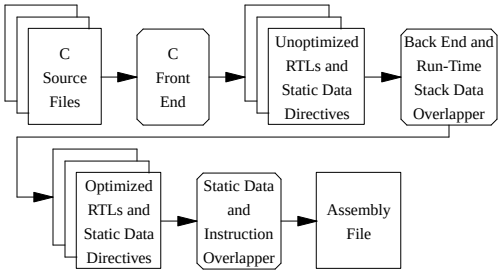


Figure 1. Environment for Overlapping Program Portions

4. Overlapping Data

Overlapping data was implemented in two steps. Run-time stack data is overlapped while optimizing each function. Static data is overlapped after all functions have been compiled since interprocedural analysis is required.

4.1. Overlapping Run-Time Stack Data

Local variables and temporaries not allocated to registers are assigned offsets after most compiler optimizations have been performed in *vpo*. The analysis required for overlapping run-time stack data is very similar to that required for allocating variables to registers, but the goal is to minimize space instead of the number of registers used. Live ranges of directly accessed (e.g. scalar) data are detected in a similar manner that is performed for register allocation in *vpo*. Unfortunately, significant size decreases will only occur from overlapping nonscalar variables, such as arrays, which are typically indirectly referenced (i.e. address is taken, but not immediately used). Detection of accurate live ranges of nonscalar variables is much more difficult since the range of array elements accessed and the loops driving the induction variable(s) associated with the memory reference must be known. For instance, assume a use of the range **`a[1..n-1]`** is followed by a set of the range **`a[0..n-2]`**. The set cannot start a new live range when **`a[n-1]`** is used at a later point in the program. Also, the range of array elements accessed cannot always be statically determined.

At this time we have implemented a simpler approach for dealing with indirect references. We find all references to indirectly used variables in the control flow of the function and create one live range for each of these variables. This one live range is simply the extent from its first reference(s) to its last reference(s), which was calculated by intersecting the basic blocks that can precede the references to the variable and the blocks that can follow the references. Note that the blocks containing the references are included in this one live range.

An example of live ranges of indirectly referenced variables is given in Figure 2. There are four arrays referenced in the control flow. The indirect reference to each array is indicated beside the basic blocks. Consider the variable **a** that is referenced in blocks 2 and 5. The live range of **a**, which is [2,3,4,5], is calculated from the intersection of its possible successor blocks [2,3,4,5,6,7,8,9] and possible predecessor blocks [1,2,3,4,5]. The live ranges of the variables **b**, **c**, and **d** are calculated in a similar manner.

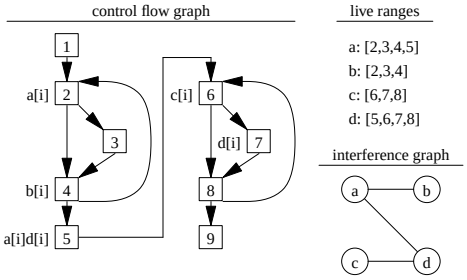


Figure 2. Indirectly Referenced Live Ranges Example

Before calculating the extent of an indirectly referenced variable, the compiler first has to determine where the variable's address is actually referenced. We accomplished this by using a demand-driven approach rather than an exhaustive solution. At each point the address of run-time stack data is taken indirectly, the compiler recursively searches forward marking all memory references that use the address. We found this approach appealing since the distance between taking the address of a local variable and the points where the address is dereferenced was typically close. Note that a single memory reference may possibly be marked as being associated with multiple local variables, which addresses many types of aliasing.

The example source code and corresponding representation in RTLs in Figure 3 illustrate the simple approach used to detect where a taken variable's address is actually

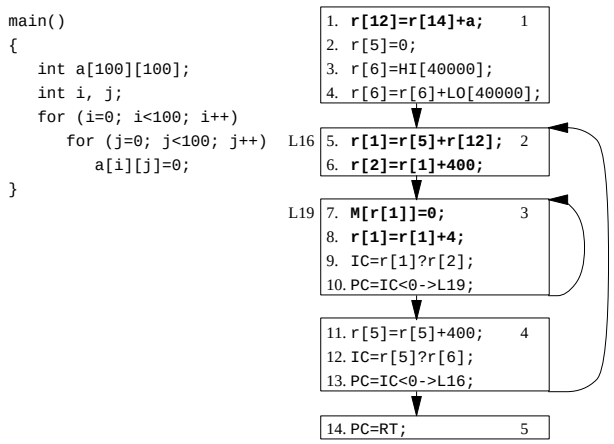


Figure 3. Determining Where Indirectly Taken Addresses Are Dereferenced

dereferenced. Delay slots have not been filled to simplify the example. The address of the local array **a** is stored in register `r[12]` at RTL 1. All of the RTLs that access registers containing the address of **a** or a relative distance from it are shown in boldface. The compiler recursively searches forward and finds that the registers `r[1]` and `r[2]` are assigned addresses relative to **a** (i.e., `r[12]`) at RTLs 5 and 6. The register `r[1]` is found to be dereferenced at RTL 7. This memory reference is marked as referencing **a**. `r[1]` and `r[2]` are not live entering block 4 and `r[12]` is not live entering block 5. No more registers containing an address relative to variable **a** are left at this point and the recursive search terminates.

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. Likewise, the same action occurs if the address is passed to a function since interprocedural analysis has not yet been performed. In both cases this action causes the compiler to conservatively consider the variable live from the point of the store or call to the end of the function.

The assignment of offsets for each live range is accomplished using an interference graph. The live range representing the extent for each indirectly referenced local variable is added to the interference graph that was constructed for the directly referenced local data. Any two live ranges that conflict are not allowed to be overlapped in memory. Note that different live ranges of the same directly referenced variable will never conflict. Figure 2 also depicts the corresponding interference graph for the live ranges shown in the same figure. Each live range is represented as a node in the graph. An edge exists between two nodes if the two live ranges conflict. Thus, the live ranges for variables **a** and **d** have two conflicts each and both of the live ranges for variables **b** and **c** have only a single conflict.

A heuristic often used to help guide the order in which the live ranges of a function will be assigned to registers is to assign the live ranges that have the greatest conflict levels first [12]. A similar heuristic was used to determine the order in which live ranges not allocated to registers should be overlapped. The rationale is that if a live range conflicts with most of the other live ranges in the function, then assigning its offset within the activation record early will give it the best chance of being overlapped with the few live ranges with which it does not have any conflicts. Size of the data was also used as a criteria so padding for alignment requirements could be minimized.

There are a few complications in assigning live ranges to offsets in an activation record that are not encountered when live ranges are assigned to registers. First, alignment requirements have to be observed. Second, variables can be of different size. Figure 4 shows the algorithm used to assign live ranges of run-time stack data to offsets. The current live range of a variable is assigned to the first offset within the activation record that does not overlap with any previously assigned live ranges conflicting with

the current live range.

```
WHILE any live ranges left to assign DO
   curr_lr := live range not yet assigned with
              biggest size and highest conflict level;
   curr_lr->offset := first offset in activation record
                      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 +
                            alignment padding;
   curr_lr->status := assigned;
```

Figure 4. Stack Live Range Offset Assignment Algorithm

Figure 5 shows the results of two different offset assignments for the live ranges shown in Figure 2. The figure also shows the declaration of the four different arrays. The first offset assignment attempts to assign the live ranges in the order of least conflicts to most conflicts (**b**, **c**, **a**, **d**). The variables **b** and **c** were overlapped. However, **a** and **d** could not be overlapped since they conflicted with each other and **b** or **c**. The second offset assignment attempts to assign the live ranges in the order of most conflicts to least conflicts (**a**, **d**, **b**, **c**). Each array required the same number of bytes to simplify the example. However, the algorithm shown in Figure 4 can be used to efficiently overlap live ranges requiring different amounts of memory as well.

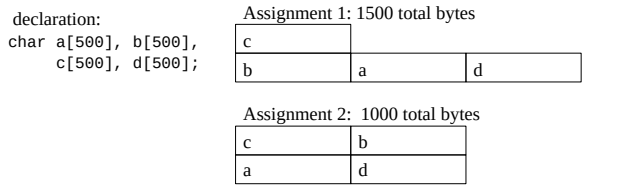


Figure 5. Offset Assignment of the Live Ranges in Figure 2

Many compilers generate assembly code that references local variables using a symbolic offset from a stack pointer. For instance, on the SPARC a local variable (e.g. called **i**) may be symbolically referenced within an instruction (e.g. **ld [%sp+.i],%g2**) since the symbolic offset can be defined (e.g. **.i = 104**). The mechanism for overlapping local variables on the SPARC was accomplished by using a different symbolic offset for each live range. For instance, the first live range of **i** and the second live range of **j** can be overlapped by assigning the same value to each symbolic offset (**.i_1 = 112** and **.j_2 = 112**). This mechanism did not require modification of the instructions when offsets were assigned and simplified debugging of analysis errors.

4.2. Overlapping Static Data

It has been observed that a large percentage of static data are arrays or other aggregate data structures [1]. If most of the composite data structures in a program are declared as static data, then overlapping data may be of limited value unless static data can be candidates for being overlapped as well. There are several types of static data in a C program. Global and top-level static variables are placed in the static data area of a process. Local variables that are declared to be static are placed in the static data area since their values are retained between calls. Composite constants, such as strings, and floating-point constants are also typically placed in the static data area.

Much of the analysis to determine the extent of use for each static data was similar to that performed for run-time stack data. However, complications due to performing this analysis interprocedurally required some changes to the analysis algorithm and some concessions to ensure that static data was safely overlapped. A call graph of the compiled functions comprising the program was constructed, which was useful for obtaining summary analysis data of called functions.

At this point, all static data was analyzed in a manner similar to that used for indirectly referenced run-time stack data. The static data overlapper finds each point in the control flow where the address of static data is constructed. The overlapper recursively searches forward marking all memory references that use the address in a similar manner for detecting where run-time stack data was referenced. When a run-time stack data address was passed to a function, the run-time stack data overlapper had to assume that the data could be referenced from that point to the end of the function since no interprocedural analysis had yet been performed. The static data overlapper does not have this restriction and instead analyzes the called routine for memory references using the passed static data address. If a static data address is passed to a library function, then the point between the call and the following block is treated as a reference to the static data. If a static data address was stored in memory (or passed to a library function, such as *setbuf*, which is known to store an address in memory), then that point was marked as a reference to the static data. In addition, the returns from the *main* function and all calls to *exit* were also marked as references. Thus, the live range of the static data whose address is stored in memory will extend from the point of the store to the end of the program.

Static data can be initialized as part of its declaration and is translated into assembly data directives. The static data overlapper marks the entry block of the main function as having a reference to each initialized static data referenced in the program. Thus, initialized static data will be viewed as being live from the beginning of the program to its last reference. If the address of one static data variable or constant is referenced in the initialization of the declaration of other static data (i.e. its address is initially in memory), then references for it are marked wherever the other static data is referenced, at the returns from the *main* function, and at all calls to *exit*. This effectively causes such static data to be regarded as live from the first references to the other static data to the end of the program.

If a top-level variable does not contain an explicit initializer as part of its declaration, then the static data overlapper assumes that the variable is uninitialized. The definition of C states that uninitialized top-level variables will have a default initializations of zero (i.e. its bits cleared). This is not true in many other languages, such as Ada. In fact, relying upon default initialization is typically considered a bad programming practice since this does not explicitly show that an initial value will be used. The static data overlapper performs analysis in an attempt to detect if an uninitialized top-level variable is ever used before it is set. When this occurs a warning message is issued that instructs the user to explicitly initialize the variable rather than relying on the default initialization.

After marking the blocks where static data is referenced, live range analysis is performed in a similar manner as the analysis for live ranges of local variables that were indirectly referenced. Some enhancements were required to address the problem of dataflow merges at the entry and exits of functions being invoked from more than one site. Figure 6 shows the control flow of an example program that is used to illustrate the analysis. A variable **x** is referenced in blocks 5 and 6 only. Calls and returns are depicted using dashed directed edges. Naive analysis treating the calls and returns as regular transitions would result in conservatively calculated live ranges. For instance, **x** would be denoted as live in function **b**.

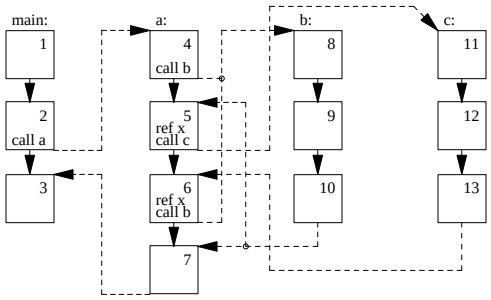


Figure 6. Example Control Flow for Live Variable Analysis

The calculation of live range information for static data was accomplished in three steps. (1) An iterative algorithm was used to calculate the blocks that can precede and follow references to each static data variable and constant. This iterative algorithm allowed information to flow from a called function, but not into it. The blocks in Figure 6 that are denoted as possibly executed preceding the reference to the variable **x** are blocks 1, 2, 4, 5, and 6. Likewise, the blocks that are found to possibly execute following the reference to **x** are blocks 3, 5, 6, and 7. Blocks 8 through 13, which are in the functions **b** and **c**, are not included since this information is not yet propagated into called functions. (2) The information about the preceding and succeeding blocks are intersected to determine the live range representing the extent from the first reference(s) to the last reference(s) of each static data variable or constant. Thus, blocks 5 and 6 are initially denoted as the live range of the variable **x**. (3) The static data live range information is propagated to the called functions. The static data denoted as live in a block that is terminated by a call to a compiled function is intersected with the static data denoted as live in the block after the call. The intersection contains the static data that is live across the call. This static data is denoted as live in the called function and any functions it in turn calls. Thus, blocks 11, 12, and 13 in function **c** are marked as part of the live range of **x** since **x** was live in blocks 5 and 6 (i.e. across the call to **c**). The live range of **x** does not include the blocks in function **b** since **x** was never live across a call to **b**.

The conflicts between static data are calculated after the live range of each static variable and constant is determined. Sometimes the live ranges of two static data may have blocks in common and still not conflict. Figure 7 has two calls to the utility **f**. One call passes the address of **x** and the other passes the address of **y**. The address in the parameter is dereferenced in block 5. So both **x** and **y** are live in that block. Yet, the live ranges do not conflict since the calls occurred from different sites. For each static data that is live in a block, there will be a conflict denoted with any other static data that is also live in the same block when either static data did not have its address passed to the function containing the block. If both addresses were passed in, then the overlapper checks the blocks containing calls to compiled functions that can reach the current function. If both static data addresses are live and one or both had its address taken in the function containing the call, then the live ranges are marked as conflicting.

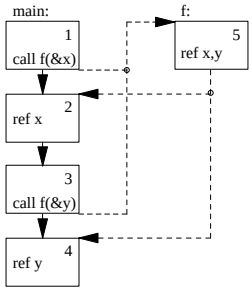


Figure 7. Example Control Flow for Conflict Analysis

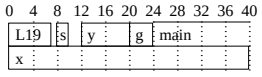
The static data within a program were assigned offsets from the beginning of the data segment in a manner similar to assigning offsets for run-time stack data. The mechanism for overlapping static data is quite simple in most assembly languages. Consider the C code segment in Figure 8(a). Assume that **y**, **g**, **s**, and the string passed to **printf** can be overlapped with **x** as depicted in Figure 8(b). The SPARC assembly directives in Figure 8(c) were generated to overlap **x** with the other static data. Note that the directives for the static data are generated in the order in which they are assigned offsets instead of the order in which they are declared. A discussion of how static data is overlapped with instructions is given in a later section.

Uninitialized static data is typically placed in a separate segment from initialized static data. For instance, Figure 9 shows that the static data area is split into two segments in

SunOS. Often operating systems provide special support for uninitialized static data by zero-filling, which avoids an initial disk access and reduces the size of the executable files [13]. Any uninitialized static data that could not be overlapped with other initialized static data (or instructions) is placed at the end of the static data area in the uninitialized data segment. Note that two or more uninitialized static data variables can still be overlapped in the uninitialized data segment by the static data overlapper.

```
int x[10];
int y[] = { 0, 1 };
int g = -1;
short s;
...
printf("Data: ");
```

(a) C Code Segment



(b) Offset Assignment

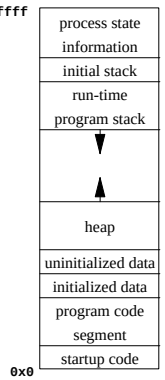


Figure 9. SunOS Virtual Address Space Organization

```
   .seg    "data"        ! switch to the data segment
   .global _x            ! make _x known to the linker
_x:                      ! assoc _x address at offset 0
L19:                     ! label of string at offset 0
   .ascii "Data: \0"     ! string value
   .skip   1             ! skip forward to offset 8 to align _s
   .global _s            ! make _s known to the linker
_s:                      ! assoc _s address at offset 8
   .skip   2             ! skip forward to offset 12
   .global _y            ! make _y known to the linker
_y:                      ! assoc _y address at offset 12
   .word   0             ! _y[0] set to 0
   .word   1             ! _y[1] set to 1
   .global _g            ! make _g known to the linker
_g:                      ! assoc _g address at offset 20
   .word   -1            ! _g set to -1
   .global _main         ! make _main known to the linker
_main:                   ! assoc _main address at offset 24
   save    %sp,-96,%sp ! first inst within _main
   ...                   ! rest of insts in relocatable portion
   .seg    "text"        ! switch to the code segment
   ...                   ! all insts not overlapped with data
```

(c) SPARC Assembly Directives and Code

Figure 8. Static Data Declarations and Assembly

5. Overlapping Instructions

Overlapping instructions was accomplished by two different types of transformations deemed likely to be beneficial. First, a general cross-jumping transformation is performed. Next, separate relocatable portions of code are abstracted into one portion when possible. These techniques for overlapping instructions have similar goals as the approach used by Fraser *et. al.* [7], with the additional goal that the number of executed instructions not increase.

5.1. Cross Jumping

Our algorithm for performing cross jumping has many similarities to other implementations of this space-saving transformation. For each basic block in the program we examine its immediate predecessors. When identical RTLs are found in multiple predecessors, the common RTLs can be overlapped without additional instructions executed when one of two conditions occur. (1) When no block falls into the current block, fall-through blocks can be created to contain the common RTLs. (2) When a fall-through block already exists, the RTLs to be overlapped must already be in this block. Each predecessor need not have the same number of common RTLs. Transfers from other blocks with common RTLs can be adjusted to jump to its first common RTL within the appropriate fall-through block. A predecessor having no common RTLs would simply transfer control to its original target.

Unlike most implementations of cross jumping, our improving transformation allows other RTLs to follow common RTLs in the predecessor blocks. We perform analysis that determines if the RTLs in the predecessor blocks can be reordered so the common RTLs appear last. This analysis first detects the registers that are set and used in each RTL within the program and efficiently represents these sets and uses as bit vectors. A search within one predecessor block for an RTL that appears in a different predecessor block is terminated when a register the RTL uses or sets is found to be set by a different RTL.

To facilitate fast comparisons of instructions, we calculate a checksum for each RTL as their sets and uses are determined. When comparing two RTLs for equivalence, the checksums for each RTL are first compared. A full comparison of the two RTLs is only required when the checksums are identical.

Traditionally, cross jumping occurs only on the instructions within a single function [6]. We also apply cross-jumping on calls to the same function from different sites. Often similar arguments are passed to the same function from different calls. Figure 10 shows an example of five calls to the function *pfnote* in the program *ctags*. Three RTLs were found to be the same preceding four of the calls. These three RTLs are overlapped before the original entry of the function. The call RTLs are updated to transfer control to a new label preceding the overlapped RTLs. Other calls to the function can still transfer control to the original label. Without inspecting the sets and uses of the RTLs to allow reordering of instructions, the first common RTL would not be detected and only six RTLs would be compressed instead of nine.

5.2. Abstracting Relocatable Code Portions

The instructions within a program may be divided into relocatable portions of code. Each portion starts with a basic block that is not fallen into from another block and ends with a block containing an unconditional transfer of control, which cannot be a call instruction. Each

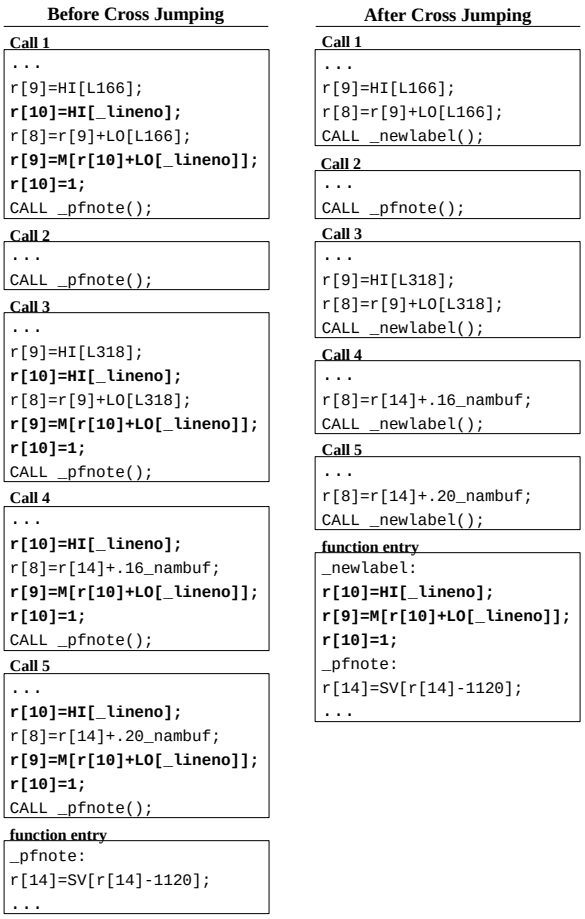


Figure 10. Example of Interprocedural Cross Jumping

relocatable code portion is compared against other relocatable portions in the program. If one code portion is entirely equivalent to another code portion or a subset of it, then the two code portions are overlapped. Branches and jumps that transfer control to the same relative location within the code portions are considered equivalent even though they reference different labels.

Figure 11 shows an example of one relocatable segment being overlapping with a portion of another in the function *yyparse* within the *lex* program. Blocks 928, 929, and 930 comprise one relocatable segment. Blocks 969 and 970 comprise another. Blocks 969 and 970 are identical to blocks 929 and 930, respectively. The code segment at blocks 969 and 970 is deleted. Block 929 is updated to have the label that was formally at block 969 to allow transfers of control to that label, such as the reference from the indirect jump table, to still be valid.

The target distance associated with branches or jumps to blocks within or from a relocated code portion may be significantly increased since the code portions being overlapped may be in different functions. Figure 12 shows the format of a SPARC branch or jump instruction. The offset

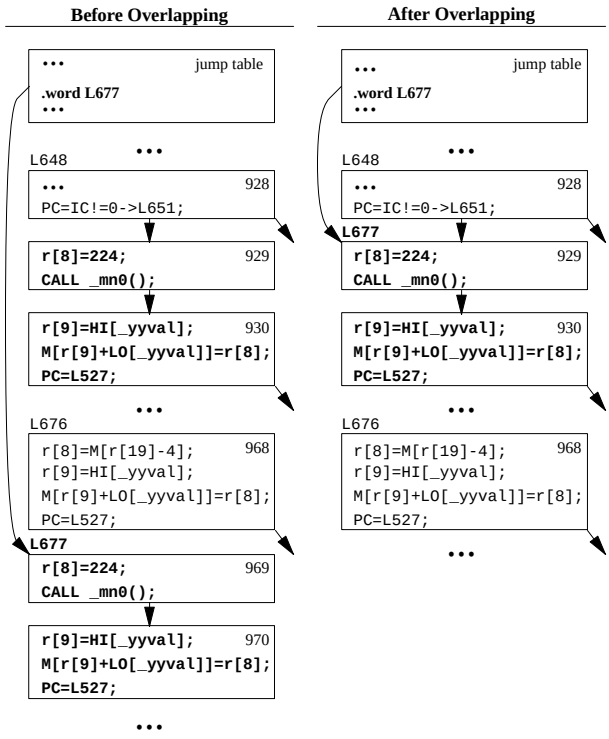


Figure 11. Example of Overlapping Instructions by Relocating Code Segments

from the instruction following the branch is a sign-extended 22 bit value representing a displacement in words (i.e. SPARC instructions), not bytes. Thus, the distance between a branch or jump and its target can be over 2 million instructions. Abstracting relocatable code portions is not performed when the distance between code portions is too great, which rarely occurs for a SPARC.

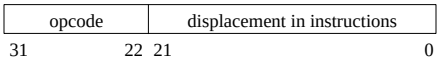


Figure 12. SPARC Branch and Jump Instruction Format

6. Overlapping Static Data and Instructions

Uninitialized static data and instructions may be overlapped in memory on a SPARC running SunOS without increasing the dynamic number of instructions executed. There are two reasons that this overlapping may occur. First, the code is next to the static data in memory, which is shown in Figure 9. Second, the displacement for branches and unconditional jumps on the SPARC as shown in Figure 12 is typically large enough to transfer between the code segment and static data area without requiring an additional instruction. Note that consistency between separate instruction and data caches need not be maintained since the overlapped area will never be

referenced for instructions after being referenced for data.

Relocatable code portions are identified in the same manner as was accomplished for overlapping instructions. A relocatable code portion can be viewed as data with an initial value. Thus, a live range for a code portion would include the code portion itself and all of the blocks in the program that can be executed preceding the portion. A code portion cannot overlap in memory with any static data with which it has conflicts. All initialized static data will conflict with code portions. The uninitialized static data that conflicts with a code portion is any static data that is live in any of the possible predecessors of the last block in the code portion.

Each relocatable code portion is assigned the address of the first offset within the data segment that does not overlap with any static data conflicting with the code portion or overlap with any previously assigned code portions. If a code portion partially conflicts with an uninitialized static variable, then only the basic blocks that conflict are moved past the variable. Any code portions that cannot be partially or completely overlapped are kept in the code segment. The same assembly skip directives used to position static data were used to position code portions within the static data area as shown in Figures 8(b) and 8(c). The assembler converts the instructions in these code portions to machine code, but leaves them in the data segment.

Figure 13 depicts how static data and instructions were overlapped in the program *cal*, which is one of the test programs used in the Results section. Figure 13 (a) shows a portion of the code in *cal*. The **string** variable is the only uninitialized static data in the program. Figure 13 (b) shows a mapping (automatically produced by the static data and instruction overlapper) that depicts how this variable was overlapped with initialized static data and relocatable code segments. The labels refer to string constants

```
...
char string[432];

main(argc, argv)
char *argv[];
{
   int y, i, j;
   int m;

   if(argc < 2) {
      printf(...);
      exit(0);
   }
...
   m = number(argv[1]);
...
   cal(m,y,string,24);
...
}

number(str)
char *str;
{
...
}
...
```

(a) Portion of *cal* Program

name	address range	num bytes	bytes saved
_string	000-431	432	0
L31	000-024	25	25
L74	025-038	14	14
L43	039-048	10	10
L55	049-056	8	8
L54	057-060	4	4
L44	061-064	4	4
L56	065-066	2	2
block range	**address range**	**num bytes**	**bytes saved**
1-3	068-103	36	36
42-44	104-123	20	20
45-45	124-135	12	12
46-50	136-199	64	64
51-51	200-207	8	8
4-18	268-483	216	164

(b) Mapping **string** with Static Data and Relocatable Code Segments

Figure 13. Overlapping Static Data and Instructions in *cal*

that are only referenced prior to the first time **string** is referenced. The ranges of blocks represent relocatable code segments. Blocks 1 to 3 is the first code segment in *main*. This segment can only be accessed before the **string** variable. Blocks 4 to 18 conflicts with the **string** variable starting at block 15. This block immediately follows the call to the function *cal*, which accesses the **string** variable. Thus, only the portion of the second code segment that does not conflict with the **string** variable, blocks 4 to 14, is overlapped with the end of the variable. Blocks 42 to 44 is the code segment at the end of the *main* function, which is only accessed when the command line arguments are invalid. The next three code segments listed are within the function *number*, which is also never invoked after **string** is accessed.

Overlapping static data and instructions requires knowing the exact number of machine instructions associated with each basic block in the program. Each RTL in the *vpo* compiler is associated with a single assembly instruction, which is quite useful when performing any span-dependent transformation. Some assemblers, such as the MIPS, may translate an assembly instruction into multiple machine instructions. Fortunately, assemblers for most machines translate each assembly instruction into a single machine instruction.[1]

7. Results

Table 1 shows the amount of overlapping that occurred in the different areas of memory. A variety of benchmarks, Unix utilities, and application programs were used to test the extent that overlapping could be applied. The number of bytes for the run-time stack was 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 was not included since library functions are dynamically linked. Little overlapping could occur on run-time stack data. Most directly referenced data (i.e. local scalar variables) were allocated to registers by *vpo*. This resulted in 90.99% of the functions having no local variables in the activation record and hence no possibility for overlapping run-time stack data. The remaining functions with indirectly referenced data provided few overlapping opportunities. There were several opportunities for overlapping static data. Often string constants and scalar global variables could be overlapped with uninitialized global variables. Typically, the larger data structures were live over most of the program and could not be overlapped with each other. Overlapping instructions with instructions could have occurred much more frequently if procedural abstraction had been used.

[1] The authors did discover a few undocumented cases where the SPARC assembler inserted an additional instruction between floating-point instructions to resolve data hazards not handled by the hardware. The *vpo* compiler was updated to modify such RTL sequences to have an accurate correspondence between RTLs and machine instructions.

Table 1: Overlapping Results

Program	Overlapping Run-Time Stack Data			Overlapping Static Data			Overlapping Instructions					Overall
								with Instructions		with Static Data		
	Bytes Orig	Bytes Saved	Pct Less	Bytes Orig	Bytes Saved	Pct Less	Bytes Orig	Bytes Saved	Pct Less	Bytes Saved	Pct Less	
cal	312	8	2.56%	725	67	9.24%	1376	20	1.16%	304	22.09%	16.54%
cmp	768	0	0.00%	16677	157	0.94%	1576	-4	-0.25%	792	50.25%	4.97%
csplit	1488	0	0.00%	2688	641	23.85%	7736	132	1.71%	510	6.55%	10.78%
ctags	8144	0	0.00%	6644	185	2.78%	9816	80	0.81%	892	9.09%	4.70%
dhrystone	644	0	0.00%	10816	66	0.61%	1956	32	1.64%	18	0.92%	0.86%
grep	592	0	0.00%	1827	105	5.75%	4048	80	1.98%	1400	34.58%	24.51%
join	480	0	0.00%	2754	127	4.61%	2736	56	2.05%	996	36.40%	19.75%
lex	9472	0	0.00%	11836	353	2.98%	35724	732	2.05%	1076	3.01%	3.79%
linpack	1504	48	3.19%	646502	76	0.01%	10588	268	2.53%	5468	51.64%	0.89%
mincost	1216	0	0.00%	11436	151	1.32%	4428	120	2.71%	768	17.34%	6.08%
sdiff	2536	0	0.00%	5312	2305	43.39%	7476	272	3.64%	1852	24.77%	28.90%
tr	192	0	0.00%	845	36	4.26%	1692	20	1.18%	488	28.84%	19.93%
tsp	3008	8	0.27%	83015	91	0.11%	4788	32	0.67%	248	5.18%	0.42%
whetstone	568	0	0.00%	466	52	11.16%	4812	184	3.82%	76	1.58%	5.34%
yacc	4232	0	0.00%	233818	1117	0.48%	31236	476	1.52%	15432	49.40%	6.32%
average	2345	4	0.40%	69024	369	7.43%	8666	167	1.81%	2022	22.78%	10.25%

However, this abstraction would have required the insertion of calls and returns, which would have increased the execution time. The negative result in the program *cmp* occurred due to interference with filling delay slots by relocating code segments. Overlapping uninitialized static data and instructions resulted in surprisingly large decreases in instruction memory requirements. Often relocatable code portions of the *main* function or various initialization functions could be overlapped with static data that had not yet been referenced. The overall savings was determined by calculating the sum of the bytes saved from the areas of memory and dividing by the sum of the original bytes. Often the static data and instructions had a larger effect since these areas typically were larger than the area used in the run-time stack.

Table 2 shows the results for overlapping run-time stack data after inlining and overlapping instructions after cloning. Inlining provided many opportunities for overlapping run-time stack data since more variables were candidates for being overlapped in a single function. 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 the files of intermediate code produced from each source file in the program, resolves conflicting labels between the files, and removes functions that are no longer referenced. Note 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 percentage decreases for overlapping run-time stack data may be larger for other architectures since compiled functions for the SPARC typically required 92 bytes to support register windows and other state information for its calling sequence. The cloning measurements were obtained by manually updating the source code of the test programs to clone the functions. The code area of cloned programs increased in size whenever cloning was possible. Cloning provided more opportunities for overlapping instructions with instructions since a cloned relocatible code segment was often identical to the original segment. While the original size decreases for overlapping data in the run-time stack and overlapping instructions with instructions were not large, more opportunities for overlapping were obtained when code duplication transformations are applied.

Table 2: Overlapping after Inlining and Cloning

Program	Overlapping Run-Time Stack Data With Inlining			Overlapping Instructions with Cloning		
	Bytes Orig	Bytes Saved	Pct Less	Bytes Orig	Bytes Saved	Pct Less
cal	232	8	3.45%	1868	344	18.42%
cmp	192	0	0.00%	1576	-4	-0.25%
csplit	728	0	0.00%	7988	148	1.85%
ctags	24544	88	0.36%	10308	53	0.50%
dhrystone	200	8	4.00%	2000	40	2.00%
grep	304	0	0.00%	4604	76	1.65%
join	96	0	0.00%	4280	40	0.93%
lex	7208	8	0.11%	44900	1700	3.79%
linpack	3312	112	3.38%	11464	330	1.92%
mincost	192	8	4.17%	4500	164	3.64%
sdiff	5784	16	0.28%	7972	292	3.66%
tr	96	0	0.00%	1692	20	1.18%
tsp	2216	56	2.53%	4788	28	0.59%
whetstone	488	296	60.66%	4812	184	3.82%
yacc	1360	8	0.59%	32800	628	1.91%
average	3130	41	5.30%	9703	270	3.04%

Occasionally, small dynamic instruction increases were observed due to overlapping program portions. Overlapping run-time stack data after inlining changed the location of local variables within activation records. Offsets larger than 4095 had to be calculated in two instructions. The dynamic instructions sometimes increased or decreased. The estimated frequency that each variable is referenced should also be a criteria for ordering the overlapping of run-time stack data after inlining. Overlapping instructions also occasionally caused very small dynamic increases since it could affect the performance of filling

delay slots. A more sophisticated instruction scheduler could undo the effects of cross jumping when needed. It also provided more opportunities for branch chaining. Overall, dynamic instructions decreased slightly.

8. Future Work

There are several areas to investigate that can potentially result in greater reductions in process memory requirements. One obvious area is to use live ranges of indirectly referenced data instead of simply calculating the extent throughout the function (run-time stack data) or program (static data) that the data can be referenced. While the current data size decreases are beneficial, the results might be substantially improved after performing such live range analysis. Another area is to experiment with heuristics for choosing the order of data to be overlapped. We used the number of conflicts and the size of the data. Many different heuristics have been investigated for register coloring algorithms. Likewise, different heuristics need to be investigated to determine the best order in practice for overlapping run-time stack and static data portions. Another promising area to investigate is automatic overlapping of fields within a structure or record, which would have the effect of unions in C or variant records in Pascal. Finally, the overlapping techniques were shown to be more beneficial after inlining and cloning. Other code or data size increasing transformations may provide additional overlapping opportunities.

While the decrease in process memory requirements has been measured, a more detailed analysis of the effect that overlapping program portions has on performance is still needed. Little impact was observed on primary instruction and data caching. While the caching performance of many programs changed, the average performance was almost identical. This may indicate that the positioning of data or instructions has a greater caching impact than reductions in size. The impact on secondary caches needs to be measured since overlapping of relocatable code portions with uninitialized static data could result in greater locality. The effect on paging of concurrent processes also needs to be observed.

Compile-time overhead for determining static data live ranges was evident when analyzing larger programs. This overhead could be significantly reduced if demand-driven instead of exhaustive solutions were used [14].

9. Conclusions

This paper described techniques for overlapping portions within the run-time stack, static data, and code areas of a program. Relocatable portions of code were also overlapped with uninitialized static data. Overlapping run-time stack data and overlapping instructions were shown to be quite beneficial for decreasing memory requirements after inlining or cloning. The techniques will be also useful for embedded systems, which often have strict memory limitations. Automatic overlapping of variables supports the software engineering practice of using appropriately named variables for different purposes. The results show significant decrease in process memory requirements for a variety of programs.

10. References

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

[2] 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).

[3] 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).

[4] 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).

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

[6] 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).

[7] 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).

[8] S. C. Johnson, "A Tour Through the Portable C Compiler," *Unix Programmer's Manual, 7th Edition* **2B** p. Section 33 (January 1979).

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

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

[11] 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).

[12] 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).

[13] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley (1990).

[14] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven Computation of Interprocedural Data Flow," *Symposium on Principles of Programming Languages*, (January 1995).