Incremental generation of high-quality target code

by MARY P. BIVENS and MARY LOU SOFFA
University of Pittsburgh
Pittsburgh, Pennsylvania

ABSTRACT

Although conventional compilers frequently apply optimization techniques in the generation of target code, some current incremental compilers do not support commonly used optimizations. This work extends the concept of incremental compilation to fine-grained, high-quality target code generation. The proposed incremental code generator changes only the affected target code and register allocations in response to a source program edit. In this paper, we first discuss some issues and analyze the actions and information needed for developing incremental code generators. From the analysis, incremental techniques for allocating registers and generating target code are developed. Both local and global register allocation are considered, using graph coloring as the allocation scheme. To evaluate the performance of the incremental system, both incremental and non-incremental systems are implemented on a VAX, and their performance is compared in terms of the quality of the target code and the savings (40% to 80%) in time for making changes incrementally rather than completely regenerating the target code.
INTRODUCTION

In the evolution of compiler technology, the production of high quality code has been an overriding concern, especially for embedded and real-time systems. Thus, numerous techniques to improve the quality of code through the application of machine independent transformations, effective register allocation schemes, and machine dependent optimizations have been developed. Major advances in these optimization techniques have increased their effectiveness in reducing the time and space requirements of target code.\(^1\)

Recently, the growing recognition of the importance of programming environments in software development has led to an interest in incremental compilers.\(^2\)\(^3\)\(^4\)\(^5\)\(^6\) While a traditional compiler uses the entire program as its compilation unit, an incremental compiler decreases the size of the compilation unit and recompiles only the affected parts. Fine-grained incremental systems use a source or intermediate code statement as the incremental unit and recompile only those statements directly changed by the programmer or indirectly affected when a program is edited. Thus, compilation time is reduced and response time improved, especially for small changes in large programs.

Although traditional compilers frequently apply optimizations including sophisticated register heuristics to reduce time and space requirements of target code, work in incremental compilers has mainly focused on the front end of compilation. Thus, current incremental compilers either do not support the traditional compiler techniques for machine dependent and independent optimizations, code generation and register allocation, or they limit the applicability of the techniques to the changed incremental unit. For example, the incremental programming environment of Feiler and Medina-Mora\(^7\)\(^8\)\(^9\) regenerates target code for an entire procedure in response to a change, while Fritzson\(^2\) regenerates code only for the source code statement that is changed. In both cases, all register allocation and optimizations are restricted to the concerned incremental unit. This restriction severely affects the quality of target code produced, especially when using a statement as the incremental unit, for the capability of allocating registers or performing optimizations across statements is prohibited. Although the quality of code improves on increasing the size of the incremental unit to a procedure, it is reported that if a procedure rather than a source code statement is recompiled in response to a change, recompilation costs are greater by a factor of ten.\(^3\) Other restrictions sometimes placed on existing incremental compilers include the assumption that all references to a particular variable are always mapped into a single, unchanging location. Thus, these restrictions on current incremental compilers negatively affect the quality of code produced and the performance of the compiler itself.

The problem of incrementally compiling intermediate code that has been optimized through machine independent transformations was recently undertaken.\(^7\)\(^8\)\(^9\) In addition to identifying important aspects of incremental optimizations, techniques for incrementally compiling both local and global machine independent optimizations were developed, demonstrating the feasibility of extending the concept of incremental compilation to include these types of optimizations.

Incrementally generating optimized target code complicates the incremental process in that the forms of the intermediate and target code, the code generation algorithm, and the register heuristics must all be considered. An analysis must be performed to determine the effect on the target code of changing a source code statement, and this analysis is complex for high-quality code. Introducing a change in a source statement causes both direct and indirect changes in the intermediate code and target code. If a variable in the intermediate code resides in a register in the target code, inserting and deleting uses and definitions of that variable can result in changes to register allocations as well as changes to the target code instructions that load and store values between memory and registers. A number of allocations and deallocations of registers during incremental code generation may result in register fragmentation, which could lead to less efficient code than that produced in a non-incremental environment.

This work addresses the problem of incrementally generating high quality target code by further extending fine-grained incremental compilation, using techniques that are compatible with traditional compilers.\(^10\) To be widely applicable, this work concentrates on local and global register allocation, the machine independent phases of target code generation. The machine dependent task of instruction selection is handled by pattern matching on a set of templates. A goal of the work is that the incrementally generated code be similar in quality to that produced by non-incremental compilers.

In order to help meet the goal of high quality code, the incremental unit chosen is the three-address intermediate statement, thus supporting commonly used register heuristics and machine independent optimizations. The intermediate representation of the source program is the standard control flow graph with nodes that are basic blocks.\(^11\) A change in the source program is represented by a set of changes to intermediate code statements. We assume the target code is 2-operand register-based assembly code that has not been peephole optimized. The available general purpose registers are assumed to be partitioned into two sets, one set for local allocation and the other for global allocation. A local register is allocated for use within a basic block; a global register is allocated for use across several basic blocks. In keeping with the goal of high quality code, the value of a variable is kept in a register until it has no more uses or until a register is needed and there are none available.
OVERVIEW

This paper describes the overall design and implementation of an incremental code generator of optimized target code. The first step in this work is to analyze how various changes to intermediate code statements affect the target code and, in particular, the register allocation. Both changes that affect statements within the basic block and those that affect the control flow graph structure are considered. One use of this analysis is to determine the information that should be gathered and maintained to permit incrementally changing instructions and register allocations. To make incremental changes, a record of register usage is necessary, so a model that represents the mapping of variables into physical registers is developed. And lastly, techniques are developed to incrementally update target code instructions, the model of register usage, and register allocations. This produces either target code or attributes the intermediate code with information about the target code that can be used to generate target code when execution is demanded.

ANALYSIS OF EFFECTS OF PROGRAM EDITS

When a source code statement is edited, the incremental compiler front end performs incremental syntactic and semantic analyses and creates a list of intermediate code statement insertions and deletions and a list of flow graph changes for the incremental code generator. To determine the effect that a change has on target code and incrementally incorporate the change, we make the distinction between a variable and the value of a variable, termed a name. A span of a name in a basic block consists of all its occurrences in the block. A name in a program can either have global extent or local extent, based on its usage. A name has local extent if all uses of the name are confined to the defining basic block; otherwise it has global extent.

Each name span in a basic block is mapped into a local or global physical register. For efficiency, if a name $X$ with local extent gives up its register at the last use in the basic block to the result $Y$ of an operation, then both $X$ and $Y$ would be mapped into the same register. A name with global extent has a name span in each block that has an occurrence of the name. This sequence of name spans forms a global span, which is the unit for global register allocation.

The mapping of names into the available local and global registers is performed using a register allocation heuristic. All names with local extent have local allocation, although the span of a local name may have to give up its register because of competition for registers. If insufficient global registers are available, a variable with global extent may have local allocation rather than global allocation, necessitating a "spill" from a register to memory. Spill code stores definitions of a spilled variable and then loads the value of the spilled variable into a register at its next use.

We examine the direct effects which include modifying the target code and register allocations for the changed statement and then consider the indirect effects which are modifications to other target code and register allocations, not marked as being changed by the edit. When a program edit causes the deletion of an intermediate code statement, the direct effect is the deletion of the corresponding target code instructions. In a system that does not keep values in registers across source code statements and does not include target code optimizations, this would be the extent of the target code changes. Similarly, the insertion of an intermediate code statement would result in the generation and insertion of the appropriate target code. In neither case would any other target code be affected. However, when the code generation scheme includes efficient use of registers and target code optimizations, intermediate code changes also cause indirect effects.

Changes That Affect One Basic Block

We first consider changes whose direct and indirect effects are local to the basic block that contains the change and thus have no effect on target code or register allocation in other blocks. Since values are kept in registers between source code statements, if a variable in a changed intermediate code statement is at the beginning or at the end of a span of uses of that variable, then the change can affect the register allocation by extending or reducing the span. These modifications can also result in creating or removing spills and in reassigning a physical register for that span of uses.

The indirect effects for locally allocated variables generally involve the target code for the intermediate code statements that correspond to the last and next occurrences in the basic block of the variables in the altered intermediate code statement. These changes consist of inserting or deleting target instructions that move variables between memory and registers.

Since the code generation scheme being considered includes the efficiency of storing only the last definition of a variable in a basic block, we must be able to mark each intermediate code statement that stores its result and update this information in response to a change. When a statement is marked for deletion and contains the last definition of a variable $X$ in the basic block, then the previous definition of $X$ must be marked as a store and a store instruction added to its target code. If the statement is an insertion, then the store status of the previous definition of $X$ must be changed and the target code that stored it to memory deleted.

If a changed statement contains the first occurrence in the basic block of a variable $X$, then it must be determined if changes, such as the insertion or deletion of an instruction that loads a register, must be made to the target code for the next use of this variable. For example, if the statement containing $X$ is to be deleted and $X$ has a next use, then target code must be inserted at the next use to load $X$ from memory to a register.

If the insertion or deletion of an intermediate code statement that contains a variable $X$ results in a change in the local liveness (a value of $X$ is locally live if it has a further use in the basic block) of the last occurrence of $X$, then target code may be changed at the statement that contains the last occurrence of $X$. Given the intermediate code statement $Z := X + Y$, if $X$ is not locally live, then the result $Z$ can be generated in the register that contains $X$. However, if $X$ is live, then the value of $X$ should be copied to a new register and the result $Z$ computed in the new register. Thus, a change in the liveness...
of a variable that is a first operand can result in changes in the number of registers necessary to compute the statement, in the span of uses for the variable and possibly in the physical register allocations.

The effects of changes on the name spans and register allocation are illustrated by an example. In Figure 1, we examine the effects of increasing the length of a name span. Figure 1a represents a basic block with each circle denoting an occurrence of a variable. The spans A, B, C, and D are assigned local registers LR1, LR2, LR1, and LR2, respectively, which is an optimal register assignment for this block. However, when the span B is lengthened, so that B and D overlap, these spans cannot both be assigned LR2. If a third register LR3 is available, then the assignment in Figure 1(b) can be made and the register number in the target code is altered. When a third register is not available, span B is spilled, so that a register is freed, as in Figure 1(c), and spill code is inserted.

Changes That Affect Several Basic Blocks

In a scheme that includes global register allocation, a change to an intermediate code statement can affect the target code in a number of blocks. A change affecting a control statement can produce a change in the flow graph such as the insertion or deletion of an edge or a node, or the merging or splitting of a node. This may result in altering the extent of variables and the size of name spans, as well as reallocation of global and local registers.

Program changes can alter the span of occurrences of a name in the following ways:

1. create a new name
2. delete a name
3. change the length of the span of occurrences of a name such that
   a. the extent remains unchanged
   b. the extent changes from local to global or global to local
4. change the priority of a name for global allocation

A name with global extent and allocation is assigned a global register for the basic blocks in its global span. The effects of changing the length of a global span are similar to the local example in Figure 1. The results of changing the extent of a name from global to local are shown in Figure 2. The names X and Y are defined respectively at statements Si and Sj in block B1, and the only uses of these definitions outside B1 are in B2. Although both X and Y have global extent, X has global allocation and resides in register GR1, Y has local allocation and is assigned registers LR1 in B1 and LR2 in B2 (see Figure 2a). The deletion of the only use of X at Sk in B2 changes the extent of X from global to local, which frees the register GR1 in B1 and B2 and requires that a local register be found for X in B1. Since X no longer has global extent, its span is deleted from the set of global spans, which allows Y to be globally allocated and assigned register GR1. The spill code for Y in B1 and B2 is removed.

The changes to the name spans for block B1 are the deletion of a span with local allocation for Y and insertion of a span with local extent and allocation for X. The change to the name spans in B2 is the deletion of the span for Y. In both blocks, these changes may cause creation or removal of spills and changes in register allocation. The changed register assignments are shown in Figure 2b.

From this analysis of changes to variables with local and global allocation, it is evident that a solution to the incremental code generation problem requires the gathering of pertinent information, the development of models to represent this information, and the creation of techniques to update the models in response to a change. The models and algorithms must be able to perform these functions:

1. maintain a mapping between the intermediate code and the target code instructions
2. determine name spans and whether the name has local or global extent
3. determine allocation and physical register assignment of name spans
4. detect register spills

INFORMATION STRUCTURES AND MODELS

Based on the characteristics of the system and on the analyses of the direct and indirect effects of a change in an intermediate code statement, the allocation of registers through a coloring scheme was found to be particularly suitable in an incremental setting. Coloring in this work is applied to local register allocation as well as global allocation, although the techniques for incremental allocation of global registers using coloring are valid without local coloring. Thus, the following models...
and information structures are developed for use in the incremental code generator.

The information needed about the intermediate code, structured as a linear list or a directed acyclic graph, includes:

1. a mapping between the intermediate code and the target code
2. for each variable in an intermediate code statement—the last occurrence, next occurrence, local liveness and the register usage

Since register allocation can be changed by the incremental code generator, it is necessary to keep a record of register usage. For each basic block, the register usage is represented by a list of virtual register spans, each composed of one or more name spans. Each virtual register span comprises the following information:

1. the list of name spans that forms this span
2. a bit to indicate the type of extent of the span
3. a bit to indicate the class of allocation, i.e. local or global
4. the physical register assigned to the span
5. two bits to indicate the spill status of a span

The extent represents the ideal class of register for a virtual register span and the allocation is the actual class of register that is assigned to the span. A name with global extent has a global span. The global span consists of virtual register spans in each block that has an occurrence of the name. Data flow information is used to determine the extent of a name.

Spilling and physical register assignment for virtual register spans with either local or global extent are done by building interference graphs. The nodes in an interference graph are either virtual register spans or global spans, and there is an edge between two nodes if the regions of the two spans overlap. The nodes in a local interference graph are a subset of the virtual register spans in that block and contain all those with local allocation. The local interference graphs are interval graphs and, unlike general graphs, can be optimally colored in time linear in the number of edges.

The global interference graph is similar, with a node representing all the virtual register spans in the global span. In general, two global spans interfere if a virtual register span of one interferes with a virtual span of the other. The global interference graph is not an interval graph, so a heuristic is used to obtain a good, but not necessarily optimal, coloring.

### INITIAL CODE GENERATION

When target code is initially generated non-incrementally, intermediate code information structures, virtual register spans, global spans, and interference graphs are constructed. The virtual register spans in the basic blocks are first constructed. From the traditional data flow information, the extent of the names are determined and the global spans are constructed.

The global spans are sorted according to a priority that weights the number of occurrences of the variables in the global span by the depth of nesting of the blocks and by the number of blocks in the global span. The global interference graph, represented as an adjacency matrix, consists of nodes that are global spans and edges that represent interferences. Two global spans interfere if they have a basic block in common. If there are \( k \) global registers, then any node with fewer than \( k \) interferences can always be colored, since its neighbors will use at most \( k - 1 \) registers. Such nodes are called unconstrained nodes; nodes with \( k \) or more interferences are constrained. The interference graph is colored by assigning registers to the constrained nodes with highest priority until all the nodes are colored or until no more nodes can be colored. Constrained nodes that are not colored are locally allocated and will compete for local registers in each basic block of their global span. Finally the unconstrained nodes are colored.

After the global spans have been allocated, the local interference graph, represented as an adjacency matrix, is built for each basic block. In the construction of the graph, if adding a node causes that number of interferences to exceed \( k \), the number of local registers, then one of the registers must be spilled. Common heuristics for local register allocation include spilling the span least recently used, the span least recently loaded, the span with fewest remaining uses or the span with next use furthest in the future. Regardless of the heuristic used to determine which span should be spilled, spilling involves splitting a virtual register span into two virtual spans and inserting spill code in the target code. When spilling is done, the second part of the spilled virtual register span becomes a new node. Spilling ensures that the interference graph can be colored with \( k \) colors; that is, that the virtual register spans can be mapped into \( k \) physical local registers.

After a \( k \)-colorable graph is constructed, it is colored by assigning a local register to each node, in the order that each was added to the graph. The graph can be colored in \( O(1) \) time since each edge is represented only once in the graph and each edge is only considered once. After coloring, the target code for the basic block is generated, using the physical register assignments obtained from both global and local graph coloring.

Figures 3 and 4 illustrate the data structures for a basic block in which all the variables have local register allocation. Figure 3 is the intermediate and target code for a basic block. In Figure 4, virtual register spans and the adjacency matrix

---

### Intermediate Code vs. Target Code

<table>
<thead>
<tr>
<th>Intermediate Code</th>
<th>Target Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A := B - C</td>
<td>MOVE B R1</td>
</tr>
<tr>
<td></td>
<td>MOVE C R2</td>
</tr>
<tr>
<td></td>
<td>MOVE R1 R3</td>
</tr>
<tr>
<td></td>
<td>ADD R2 R3</td>
</tr>
<tr>
<td>2. C := C - A</td>
<td>ADD R3 R2</td>
</tr>
<tr>
<td></td>
<td>MOVE R2 C</td>
</tr>
<tr>
<td>3. A := B - C</td>
<td>ADD R2 R1</td>
</tr>
<tr>
<td></td>
<td>MOVE R1 A</td>
</tr>
<tr>
<td>4. X := A - Z</td>
<td>MOVE Z R2</td>
</tr>
<tr>
<td></td>
<td>ADD R2 R1</td>
</tr>
<tr>
<td></td>
<td>MOVE R1 X</td>
</tr>
</tbody>
</table>

Figure 3—Intermediate code and target code for a basic block before changes
are shown. The matrix indicates that nodes one and two overlap, that node three interferes with nodes one and two, and that node four overlaps node one. Without spilling, this graph can be colored with three registers; the physical register assignments are indicated.

INCREMENTAL TECHNIQUES

Using the analyses and the models described above, algorithms have been developed to incrementally generate target code for a group of changes to intermediate code statements. In the incremental process, the values of the affected intermediate code attributes are changed, target code is inserted and deleted, and virtual register spans, global spans, and physical register assignments are changed. These algorithms produce code that is of similar quality to that produced in a non-incremental environment, since they detect the destruction and creation of opportunities to use registers efficiently. The algorithm for the incremental process is outlined below.

ALGORITHM: GLOBAL_PROCESS_CHANGES.

This algorithm processes a list of changes in a procedure P. The changes are insertions and deletions of statements in a basic block B and changes in the flow graph.

BEGIN {GLOBAL_PROCESS_CHANGES}

For each basic block B with intermediate code changes DO BEGIN

FOR each basic block B marked as changed DO BEGIN

Update_Local_Adj_Matrix(B);
ReColor_Local_Adj_Matrix(B);
Update_Target_Code(B);
END;
END.

Changes to Locally Allocated Variables

The incremental process for variables with local register allocation is illustrated by inserting an intermediate code statement in the example described in Figures 3 and 4, and incrementally updating the information structure, the virtual register spans, the local adjacency matrix and the target code. We assume that three physically registered registers are available for local allocation.

To insert the statement \( Z := B + A \) before statement 1 in Figure 3, the following is done. Since the inserted statement becomes the first statement in the basic block, none of its variables has a last occurrence. Statement 0 is marked to store its result as this is the only definition of \( Z \) in the block.

The next occurrence of \( A \) is a definition at statement 1, so \( A \) is not locally live and a new virtual span is created for it. The variable \( B \) is locally live since it has a next use at statement 1, and it will become part of the virtual span that includes \( B \) at statement 1. As an indirect effect, the target code for statement 1 is marked to remove the instruction that loads \( B \) from memory. The result \( Z \) has a next use at statement 4, so \( Z \) is locally live and will become part of the virtual span that is a use of \( Z \) at statement 4. The target code for statement 4 is marked to remove the instruction that loads \( Z \) from memory.

Figure 5 shows the updated virtual spans and the updated adjacency matrix. When span 3 is added to the matrix, it overlaps three other nodes so four physical registers are needed. Since only three registers are available, one virtual span must be spilled. Virtual spans 1 and 2 are used at the start of span 3, so only span 4 can be spilled.

Span 4 is broken into span 4A with range \([0,0]\) and span 4B with range \([4,4]\). The adjacency matrix after spilling is also shown in Figure 5. \( Z \) is spilled, and statement 1 is checked to
see if the value of Z in memory is current. Statement 4 is marked to insert an instruction to load Z from memory. The updated adjacency matrix is incrementally colored by examining the changed nodes and keeping their old colors if possible. The intermediate code, the old target code and the updated target code are shown in Figure 6.

Incremental Global Coloring

When a change involves variables with global extent, the incremental process may include creating, deleting, merging, and splitting global spans, and updating and recoloring the global adjacency matrix. Although similar to local changes, the algorithms for incorporating these changes are more complex since one change can affect several basic blocks. In addition, the global interference graph is not an interval graph, since two virtual spans may represent parallel regions in the control flow graph and cannot be ordered. To give a flavor of the technique, we briefly describe the incremental coloring algorithm.

The incremental global update algorithm processes changed nodes in order of their priority. Changes to nodes in the graph include deletion, insertion, change in the length of a span, increase in priority and decrease in priority. The deletion of a node Ni that has global allocation causes each of its neighbors with lower priority and local allocation to be marked for recoloring since the global register assigned to Ni is available to another span. The deletion will also reduce the number of interferences for each neighbor and may change some constrained nodes to unconstrained. When the priority of a node with local allocation is increased, or the priority of a node with global allocation is decreased, the node is marked for recoloring. Similar changes are made when a node is inserted and when the size of the span of a node is changed. After all changed nodes have been processed, nodes with local allocation that are now unconstrained have their allocation changed to global and are marked for recoloring.

Recoloring is done in order of decreasing priority. A constrained node Ni, marked for recoloring, is globally allocated if there is a color that is not used by its higher priority neighbors, or else it will be locally allocated. When a color is available for a node, the colors of its neighbors with lower priority are checked to see if the color is being used. If the color is in use by a neighbor, this neighbor is marked for recoloring. Since nodes are recolored in order of their priority and a neighbor with higher priority is never disturbed, incremental recoloring can be done in one pass.

PERFORMANCE AND IMPLEMENTATION

To experimentally evaluate the performance of the incremental code generator, a prototype of the system has been implemented for the VAX/780 as well as a non-incremental code generator that uses the same register allocation heuristics. The performance of the incremental system is evaluated by:

1. comparing the quality of the target code that is produced by the incremental system versus that produced non-incrementally
2. measuring the savings in compilation time when a change is incorporated incrementally rather than regenerating the target code

The implementation uses the front end of an ADA compiler to generate 3-address intermediate code. A data flow analyzer then creates data and control flow information for the intermediate code, and the non-incremental target code generator creates the virtual registers, global spans and target code for the program.

The incremental code generator takes as input the structures built by the non-incremental code generator and a list of program changes and incrementally incorporates the local and global changes. Although the code produced by the two systems may not be identical, when it differs, it is only in the physical register assignments.

Experimental studies done so far indicate savings of from 40% to 80% when approximately 4% of the intermediate code is changed. Edits that involve only local changes and those that include both local and global changes were both tested. The results for both types of changes are in the same range. The variance in the results is based on the perturbation caused by the changes. If a change is local and involves a virtual register span that is at the beginning of the basic block, it is more likely to disrupt the register allocation for that basic block than is a change that occurs at the end of the block. Likewise, a change in a high-priority global span that has many interferences will disrupt its neighbors more than a change in a low-priority span that has few interferences. Work is currently underway on categorizing the changes that are well-suited for incremental target code generation.

ACKNOWLEDGMENTS

This work was partially supported by NSF under Grant DCR 811934.
REFERENCES


