Chapter 12 Back-End of the Compiler

Type analysis, inlining, and splitting are performed simultaneously as the first pass of the SELF compiler. This chapter describes the remaining passes of the SELF compiler, describing those optimizations included to be competitive with traditional optimizing compilers.

12.1 Global Register Allocation

In many ways, good register allocation is the most important "optimization" in traditional compilers. The effect of register allocation is felt through all compiled code, unlike most other optimizations, and a poor allocator can hurt performance so much that any improvements caused by other optimizations are inconsequential. However, register allocation historically has been much less important in implementations of pure object-oriented languages (and many other kinds of high-level languages), primarily because these high-level languages contained so many procedure calls that register allocation became of secondary importance. Register allocation algorithms must flush caller-save registers to their home stack locations across procedure calls, and in language implementations with many procedure calls there is not enough straight-line code between calls to justify any but the simplest of allocators. In contrast, the SELF compiler is designed to eliminate most procedure calls and frequently compiles whole versions of loops with no internal calls. Global register allocation thus again becomes important, especially in light of our goal to rival the performance of traditional optimizing compilers.

A straightforward register allocator would assign each variable in the source program to a register or stack location, such that two simultaneously live variables are not assigned the same register or stack location. This naive approach would work poorly in an environment like SELF where method inlining is commonplace, since a formal variable name of an inlined method will be simultaneously live with the corresponding actual parameter, and consequently the two variables will be allocated separate locations, even though both variables will always contain the same value. With many layers of inlined methods this duplication of what conceptually should require only a single location can introduce a huge overhead of register copying as layers of inlined methods are entered and exited.

A better system could perform copy propagation prior to register allocation, replacing uses of each of the formal variables in inlined routines with uses of the corresponding actuals. Unfortunately, straightforward application of copy propagation could make supporting complete source-level debugging more difficult, since a given variable name might be replaced by several different variable names at different points during its lifetime.

The SELF compiler incorporates an alternative strategy, introducing a new abstraction between a name and its assigned location called a *variable*. Names that are always *aliases* of one another (such as actuals and formals) are mapped to the same variable object, and each variable object either is assigned a single run-time location (i.e., a register or a stack location) or is marked as a compile-time constant (and so does not need a run-time location). By assigning locations to variables rather than names, the SELF compiler eliminates the impact of artificial name distinctions caused by inlining and reduces the SELF register allocation problem to one similar to that faced by traditional optimizing compilers with built-in control structures and limited inlining.

An earlier version of the SELF compiler included a much more ambitious register allocator. Instead of allocating locations to names or even to variables, the earlier compiler allocated locations to values. This makes perfect sense, since values are precisely those run-time "bit patterns" that are manipulated by the program; names are merely convenient handles by which programmers and the compiler refer to values. The earlier compiler went to the extreme of allowing each use of a value and each straight-line inter-use segment of a value's lifetime to be independently allocated, either in a location or marked as a compile-time constant with no run-time existence. This design provided the allocator with a lot of freedom to implement various parts of a value's lifetime in different ways. Unfortunately, this earlier register allocation design had three main drawbacks:

- The earlier allocator was very slow. This was primarily caused by the extremely fine granularity of allocation (a single use or lifetime segment). The current implementation, like most other register allocators, allocates whole lifetimes in a single step and so runs much faster.
- The earlier allocator had to work hard to make sure that long sequences of lifetime segments and uses of a single value were all allocated to the same location, since otherwise register moves might be inserted in the middle of a value's lifetime unnecessarily. Additionally, a name that is bound to several variables over its lifetime, such as a

counter that gets incremented each time through a loop, should get all its values allocated to the same location so that unnecessary register moves are not inserted at assignments to the name. Naturally the allocator was imperfect, sometimes introducing register moves in awkward places in the generated code, and consequently the earlier SELF compiler did not achieve the same level of code quality as that in a traditional register allocator (traditional register allocators have a much easier job since usually they support neither complete source-level debugging nor massive inlining of user-defined control structures). The current SELF compiler's register allocator overcompensates by requiring that a name is bound to a single variable for its entire lifetime, and a variable be allocated to a single location for all its lifetime.

- The earlier register allocator never supported the source-level debugger. The intention was for the compiler to generate tables describing the mappings from names to values and values to registers as part of the information generated for use by the debugger. (The current version of this debugging information will be described in detail in section 13.1.) These mappings would be indexed with the hardware program counter, since the mappings depended on where in the compiled code the program was stopped. Unfortunately, the design of a space- and time-efficient representation of these mappings proved difficult and was never completed. The current SELF compiler avoids the complexity of time-varying allocations by restricting the mappings to be position-independent.

Compared to the earlier more ambitious register allocation strategy, the current strategy is simple, fast, and reasonably effective.

12.1.1 Assigning Variables to Names

The compiler constructs the name/variable mapping as part of type analysis. In the compiler's internal representation, each name object refers to its corresponding variable object, and each variable keeps track of which names refer to itself, called the variable's *alias set*. When a name is first assigned an initial value, the name is simply added to the alias set of the variable associated with the right-hand-side of the assignment. If the name is subsequently assigned a new value, it can no longer be considered an alias of the initializing expression, and consequently the name is removed from its old alias set and allocated its own fresh variable object. Once type analysis completes, each name will be associated with a single variable that represents the name's alias set: those names that always contain the same value and hence can be allocated the same location.

Also during type analysis, the compiler records extra information with names for each occurrence of the name as an operand of a run-time expression. This per-use information records the preferred location (such as the particular register or stack location required as part of parameter passing) or kind of location (such as any address register on the 68000 architecture) for that use, plus the weight of the control flow graph node containing the use (weights are described in section 10.2.3.2). This extra information helps the register allocator to avoid unnecessary register movement and to concentrate its efforts on the most frequently executed parts of the control flow graph.

12.1.2 Live Variable Analysis

After eliminating redundant constants (described later in section 12.2) and unneeded computations (described in section 12.3), the compiler begins the actual register allocation phase. As with most global register allocators, the SELF compiler computes the *register interference graph* to determine when the lifetimes of two names overlap. The nodes in the interference graph are variable objects rather than name objects.

To compute the register interference graph, the compiler makes a backwards pass over the control flow graph. As the compiler passes over the graph, it maintains a set of *live variables*, computed as the set of variables that have been used downstream in the control flow graph but have not yet had their initial definitions. When the compiler reaches a variable's initial definition, the compiler removes the variable from the live variable set. When the compiler reaches a use of a variable not already in the live variable set, i.e., the last use of some variable, the compiler adds the newly-live variable to the live variable set and adds interference links between the newly-live variable and the other variables already in the live variable set. By scanning backwards through the control flow graph, the compiler can detect the end of a variable's lifetime when it encounters its last use (which will be the first occurrence of the variable in the backwards pass). The beginning of a variable's lifetime can be detected without a corresponding forward scan since the initial assignment to any name is known statically and is represented by a different kind of control flow graph node when the control flow graph is constructed.

Merges, branches, and loops complicate this backwards traversal. When the compiler reaches a merge node in the backwards traversal, it places all the merge's predecessors but the first on a special stack of <control flow graph node,

live variable set> pairs representing work pending, and then the compiler processes the merge's first predecessor. When the compiler reaches a particular branch node for the first time during the live variable analysis, the compiler must stop processing this path and wait for the other branch successor to be analyzed before the compiler can process the branch's predecessors. The compiler records the current live variable set with the branch node, pops a <node, live variable set> pair off the pending nodes stack, and resumes processing this other node with the corresponding live variable set. Once the branch node is reached for the second time from the other successor branch, the branch's predecessor can be processed; the live variable set for the branch's predecessor is the union of the current live variable set and the live variable set recorded as part of the earlier visit to the branch node. In the absence of loops, this search pattern guarantees that a node is processed after all of its successors have been processed (i.e., traverses the graph in reverse topological order), ensuring correctness of the computed live variable sets.

Unfortunately, in the presence of loops, not all nodes can be processed before their successors, and some sort of iterative algorithm is needed to compute the fixed point of the live variable sets. Whenever the compiler runs out of pending <node, live variable set> pairs to process, but there remains at least one branch node whose predecessor is not processed yet (e.g., because its other successor eventually leads to a **`_Restart`** backward branch to an earlier loop head), the compiler simply forces the branch node to process its predecessor early. The compiler initially assumes that the live variables from the unprocessed successor are a subset of the live variables from the processed successor. Once the other branch node's successor is finally processed, the compiler can verify whether this assumption was correct. If it was, then the compiler is done with this branch node, and can find other nodes to process. If, however, the second successor had some live variables that are not in the first successor's live variable set, then the earlier analysis of the branch node's predecessor was inadequate, and the live variable analysis must iterate by reprocessing the branch node's predecessor with a larger live variable set.

Instead of completely repeating the analysis as would be the case with a normal iterative data flow algorithm, the SELF compiler uses a special mode of live variable analysis to update the earlier results incrementally. In this mode, the compiler propagates the *difference* between the previous live variable set and the new live variable set. This *incremental live variable set* is initialized as the set difference between the live variable sets of the second predecessor and the first predecessor. Incremental analysis differs from normal non-incremental analysis in that no variable ever needs to be added to the incremental live variable set, since the variable would have been present in the previous normal live variable set and so is not present in the difference between the previous and new live variable sets. Variables still are removed from the incremental live variable set as their initial definitions are reached, since they no longer appear in the new live variable set. Once the incremental live variable set becomes empty, the previous and new live variable sets are the same, and incremental analysis of the path can stop. Incremental re-analysis is faster than non-incremental re-analysis since the incremental live variable sets are smaller and it is faster to check whether the incremental live variable set is empty than to check whether the previous and current live variable sets are the same.

The compiler represents both live variable sets and the per-variable sets of interfering variables with a dual bit-vector/linked-list data structure. This representation of sets supports both constant-time set member testing via bit vectors (each variable is allocated a unique integer index used as its position in the bit vectors) and linear-time (in the size of the set) iteration through the elements of the set via linked lists. Execution profiling of the SELF compiler shows that neither the extra space cost for two representations of the same set nor the extra compilation time cost to copy two representations of the same set at merge nodes is significant. Therefore, this dual representation supports set operations better than either traditional representation.

12.1.3 Register Selection

After the register interference graph is built, the compiler visits each variable and allocates it a location. The compiler uses the weights of the uses of the names associated with the variables to determine the order in which to allocate variables to registers, with the highest weight variables allocated first. If the variable has no run-time uses and either the variable is a compile-time constant or the variable is not a source-level variable and so not visible to the outside world, then it is not allocated a run-time location. Otherwise, the preferences specified with the uses of the names associated with the variable are used to pick a register or stack location that is different from any already chosen for an interfering variable.

This allocation strategy is simple and fast. However, it is limited by only allocating a single location or compile-time constant to each variable. Since each name is associated with a single variable, completely disjoint portions of a name's lifetime, sometimes created by splitting, cannot be allocated to different registers or more importantly cannot be associated with different compile-time constants. This is especially common for names bound to either the **`true`** constant or the **`false`** constant, such as the results of comparison operations. If the contents of this name might be visible from the debugger, then with the current register allocation limitation the name must be allocated a run-time location and this location must be initialized at run-time with either **`true`** or **`false`**. This is true even though the program itself will have no need for the run-time value, since after splitting the value will be encoded in the position in the program (just as traditional compilers encode the result of boolean tests within **`if`** statements by positions in the compiled code).

Fortunately, in most common circumstances this particular problem can be side-stepped by creating new names for formals of inlined methods (rather than reusing the names of the actuals as would be more natural and concise) and specially handling the debugger's view of expressions used as arguments to inlined methods. For example, if the **`ifTrue:`** message is sent to the result of a comparison, then typically two **`ifTrue:`** methods will get inlined, one for receivers of type *true* and another for receivers of type *false*. If name of a formal were the same as the name of the corresponding actual, then in this case the single name which is bound to the result of the comparison would be reused as the names of the receiver formals in both inlined versions of **`ifTrue:`**. If this formal is visible to the debugger, then this approach would cause the register allocator to allocate a run-time location to hold the value of the result of the comparison. Instead, the current compiler creates new names for formals, and assigns the actuals to the formals. Then the receivers of the two **`ifTrue:`** methods have their own names, which can correctly be allocated to compile-time constants. The original comparison result name is no longer used by the debugger, and so can be left unallocated. Thus for this common case the compiler will not introduce extra run-time overhead.

However, in general, the limitation of a single compile-time constant or location for each name is still a problem. The earlier register allocator was able to solve this problem by allowing the allocation to be different for different parts of the control flow graph, but proved too inefficient for practical use. Register allocators for other compilers frequently allocate disjoint subregions of a variable's lifetime to different registers, and sometimes can even split a variable's lifetime into separately-allocatable regions [CH84, CH90], but these systems do not simultaneously support complete source-level debugging. We continue to search for some "happy medium" register allocator that combines some form of flexible position-dependent allocation with high allocation speed and compact debugging information.

12.1.4 Inserting Register Moves

After allocation, a pass through the control flow graph inserts register move control flow graph nodes where needed. Places where the compiler inserts register moves include before assignments nodes whose left- and right-hand-sides are allocated to different registers and before message send nodes to move arguments into the locations required by the calling convention.

12.1.5 Future Work

The compiler's computation of variables as the set of aliased names is not always as large as possible. For example, consider the following simple sequence appearing in most looping control structures:

```
i: i + 1.
```

After executing the **+** message but before executing the **`i:`** message, both the original value of **`i`** and the result of **+** are simultaneously live and holding different values. Consequently, the compiler assigns the two expressions different variables. However, if the compiler knows that **`i`** is an integer and inlines the **+** and then the **`intAdd`** primitive call down to a simple **`add`** instruction (ignoring the overflow check for the moment):

```
i ← add i, 1
```

the original value of **`i`** is never used after the **`add`** instruction. Thus the actual run-time lifetimes of **`i`** and the result of the **+** message do *not* overlap, and they could (and usually should) be allocated to the same register.

We say that two different variables are *adjacent* if their lifetimes do not overlap and one variable is assigned to the other. Adjacent variables usually should be allocated to the same location to minimize register moves when one variable is assigned to the other. One way of achieving this would be to combine the two variables into a single larger variable; this transformation is called *subsumption*. Unfortunately, this combining cannot be performed as part of forwards type analysis, since when the second variable's lifetime begins (such as after the **+** message above) the compiler does not yet know that there will be no additional uses of the first variable. However, it may be possible to augment the live variable analysis pass to detect adjacent variables, mark them as such, and extend the register allocator to allocate adjacent variables to the same location. An alternative approach would compute variables in a separate backwards pass after type analysis, thereby providing last-use information to the analysis and implicitly coalescing together what would have been different but adjacent variables.

12.2 Common Subexpression Elimination of Constants

After type analysis, the compiler performs a pass over the control flow graph to eliminate redundant loads of constants. This pass creates extra names to represent available constants and replaces subsequent redundant loads of constants with simple assignments to these new names.

Unfortunately, this technique does not interact well with the current register allocation strategy. New variable objects are created during this phase for the newly created names of constants. Since distinct variables are allocated independently and hence frequently to different registers (as described in section 12.1.3), extra register moves may be inserted at both the point of the initial constant load instruction (if the register for the new variable is different from the register for the result of the load instruction) and at each eliminated use of the constant (if the new variable is allocated to a register different from the original variable for the use). These extra register moves diminish the benefits of common subexpression elimination of constants. Some of these problems could be avoided by performing common subexpression elimination of constants in the same pass as type analysis and name/variable mapping construction, thus eliminating the extra register move when using a saved constant; unfortunately, subtle interactions with other parts of the compiler complicate this approach. Also, the techniques for allocating adjacent variables to the same location as described in section 12.1.5 might help.

However, even ignoring these problems, common subexpression elimination of constants (and in fact common subexpression elimination of any relatively cheap computation such as an arithmetic operation) can still *slow programs down*, if the variable for the constant is allocated to a stack location rather than a register. Storing and subsequently fetching a constant from a stack location is significantly costlier than redundantly loading a 32-bit constant value into a register. Common subexpression elimination and register allocation should to work together to avoid these sorts of inadvertent pessimizations. Unfortunately this kind of inter-pass cooperation is notoriously difficult to manage.

In practice, common subexpression elimination of constants has a mixed effect. According to the results shown in section B.3.10 of Appendix B, some benchmarks speed up with this optimization, while others slow down. Clearly this technique should be redesigned and reimplemented to improve its effectiveness.

12.3 Eliminating Unneeded Computations

In a third phase the SELF compiler scans all names to find those whose value is computed but never used. If such a computation exists that has no side-effects, such as a simple assignment, a load-constant instruction, a memory fetch instruction, an arithmetic instruction, an object clone primitive call, or a call to any other side-effect-free primitive, then the compiler simply splices the unnecessary operation out of the control flow graph.

This optimization improves the performance of SELF programs by a few percent on average, according to the measurements shown in section 14.3.

12.4 Filling Delay Slots

After register allocation, the SPARC version of the SELF compiler attempts to fill branch and call delay slots. In the SPARC architecture, branch instructions have a one-instruction delay slot that either is always executed or is executed only if the branch is taken (an *annulled* branch). The SELF compiler attempts to fill these delay slots with useful instructions, preferably from before the branch instruction, otherwise from the most likely successor, and finally from the remaining branch successor. Call and jump/return instructions similarly have one-instruction delay slots that the SELF compiler attempts to fill from before the instruction.

Many nodes in the control flow graph do not generate machine instructions, such as assignment nodes and other bookkeeping nodes, and therefore this searching for an instruction to move to a delay slot may scan several control flow graph nodes before either locating a node that generates a single instruction or finding a node that cannot fit into a delay slot (such as a message send node). If a node is found to put in the delay slot, then all the nodes between the branch or call and the target node are spliced out of the control flow graph (adjusting the predecessor or successor links of the branch or call appropriately) and the removed chain of nodes is stored in the branch or call node for later code generation.

If when searching for an instruction after a branch node the compiler encounters a merge node, the compiler attempts to locate a candidate instruction from after the merge node. If it finds one, then the compiler duplicates the control flow graph between the merge and the located node, delaying the merge until after the located node. Then the normal mechanism of splicing out the nodes between the branch and the target node applies; the branch target is automatically adjusted to jump to the instruction after the original merge point as part of the splicing operation. This "splitting" of merges when filling delay slots is important to fill as many delay slots as possible. No additional code space is needed for this kind of splitting, since at most one real instruction is copied and inserted into a delay slot that would be wasted otherwise.

Filling delay slots is important for good performance. According to the results shown in section 14.3, delay slot filling improves performance by more than 10% for many benchmarks, and reduces compiled code space consumption by more than 10% on average.

Other code generators for RISC machines perform additional kinds of scheduling of instructions, such as reordering memory loads to minimize stalls caused by waiting for the result of a load. The current SELF compiler currently does no such instruction scheduling.

12.5 Code Generation

The final pass of the compiler traverses the control flow graph and generates native machine code. Instruction selection is nearly trivial in the SELF compiler, since the compiler is designed primarily for modern RISC machines with simple, regular instruction sets. No peephole-style optimizations are performed, since (in most cases) none are needed. The compiler also generates debugging information (described in section 13.1) in this pass. Once generation is complete, the compiler passes the buffers used to hold the instructions and debugging information to the compiled code cache manager, which creates a new compiled method and adds it to the cache (possibly throwing out other old methods or compacting the cache to make room for the new method). The compiler then returns the starting address of the compiled code to the run-time message lookup routine which invoked the compiler. This lookup routine then completes the message send by jumping into the newly generated machine code.

12.6 Portability Issues

The SELF compiler has not been designed specifically to be retargetable to a new machine architecture without programming changes, but it has been designed and implemented to make porting the compiler to a related architecture relatively straightforward. For example, the compiler describes the set of registers on the target machine and other register- and stack-related calling conventions through a table of compile-time constants that can be easily changed for an architecture with a similar system of registers. Other parts of the compiler use a minimal number of **`#ifdef`**'s to isolate machine dependencies. Most control flow graph nodes are intended to map to a single target machine instruction (to allow as much low-level optimization such as delay slot filling as possible), and so there is some dependence on the kind of instructions provided by the target architecture in the kinds of control flow graph nodes. This approach works best for RISC-style architectures (the SELF compiler's primary intended target) and less well for CISC or other machine styles; some sort of peephole optimizer or generalized instruction selector would be needed to support these other kinds of architectures effectively. The largest machine dependency in the compiler is on instruction formats; this dependency is isolated in a per-machine subsystem of the compiler that is responsible for producing machine-specific instruction formats for higher-level control flow graph nodes such as **`add`** and **`branch`**. Some control flow graph nodes are only used for some architectures (such as the **`sethi`** control flow graph node on the SPARC), and some phases such as delay slot filling are executed only on architectures that need them. The difference between machines with three-operand instructions and machines with only two-operand instructions is handled in the register allocator by constraining the destination register to be the same as one of the sources for two-operand machines.

We expect that the SELF compiler could be ported to a new RISC architecture and generate reasonably good code in a week; other parts of the SELF implementation such as the memory system or some of the low-level assembly code support might take longer to port. CISC machines could also be supported in the same amount of time, but without a peephole optimizer or instruction selector run-time performance may not be as good as other compilers for the same machine. Just like other compilers, the SELF compiler could benefit from current work in table-driven compiler compilers to improve portability.

12.7 Summary

After the type analysis/inlining/splitting phase, the compiler executes several additional phases on the way to generating machine code:

- The compiler eliminates redundant loads of constants. This currently does not interact well with register allocation and variable assignment, and so much of the benefit is wasted in unnecessary register traffic.
- The compiler eliminates unneeded computations such as arithmetic and memory loads.
- The compiler performs live variable analysis, allocates registers to variables, and inserts any needed register moves. The compiler uses variables to represent sets of aliased names in response to the heavy use of inlining in the SELF compiler. The register allocator should be extended to avoid inserting register moves for adjacent variables and to support finer grained position-dependent allocation.
- The compiler fills delay slots of branches and calls. This could be extended to schedule load delays as well.
- The compiler generates native machine instructions and debugging information, and adds the new compiled method to the compiled method cache.

Porting the SELF compiler to a new architecture would likely be neither overly difficult nor trivial.