A Procedure Inliner for Machine SUIF2

Russ Ross & Deborah Abel
{rross,dabel}@fas.harvard.edu

May 16, 2000

Abstract

The optimization of procedure inlining, while generally not a win in its own right, can enable other compiler optimizations such as constant propagation and dead code elimination. In creating an inlining pass for Machine SUIF2, we have kept the framework as general as possible, allowing for inlining at any phase in the compilation sequence as well as making it straightforward to research heuristics about when inlining is useful and how deeply it should be nested.

1 Introduction

An optimization pass currently missing from Machine SUIF2 is that of a procedure inliner, which replaces call instructions with the body of the called procedure. This allows for optimizations such as constant propagation and dead code elimination to work across what was previously a procedure boundary. In addition, inlining can eliminate the need to honor calling conventions, thus reducing the overhead of loads and stores associated with a call instruction and allowing for more effective register scheduling.

Inlining has two distinct components: deciding when a particular call site should be inlined and actually merging the instruction list of the called procedure into the host procedure. The former is the subject of most of the literature in the field and is essentially a combination of heuristics taking into account the data and control flow of the program. The latter is closely tied to the underlying compiler architecture and involves only simple data-flow analysis; the bulk of the work is in transforming an independent procedure into a form suitable for inclusion in the middle of an existing procedure. Our project focuses on the second component with the goal of creating a flexible framework so that others can test different inlining heuristics without having to address the lower level issues.

We accomplish this by automating the process of inlining marked call sites in the pass `do_inliner`. `do_inliner` expects call sites to be suitably annotated if they are to be inlined. A separate pass, `do_pre_inliner`, actually marks selected call sites for inlining. Currently, `do_pre_inliner` does a linear scan through the instruction lists of all the procedures in the file, asking the user at each call site if they want to inline it and to what depth. In addition, we gather some simple statistics about procedure lengths and the number of call sites within each procedure as a starting point for someone writing an automated annotator.

We begin by explaining our implementation decisions and details, some of which were dictated by quirks in the Machine SUIF2 system. Then we give simple instructions for using our code.

2 The Inlining Process

There are several steps that must be taken and issues that must be addressed in actually inlining a sequence of instructions. Here we list the most important points that came up.

2.1 Pre-inlining work

We currently allow inlining only from within the same file. This simplification eliminates the need to have a comprehensive catalog of procedures available for inlining along with the ability to find them. Thus, the `do_pre_inliner` pass finds all the procedures and call sites in the file, collecting statistics about the length of the procedure and the number of call sites within that procedure. Our current implementation does not use this information; however the data may be useful as heuristics for automatically marking call sites to be inlined. The annotation contains the name of the procedure being called and the maximum depth to which it should be inlined. Our current implementation scans through the instruction lists of each of the procedures in the file, stopping at each call site and asking the user if they want to inline the procedure and to which depth.

The target of a call instruction is not trivially available from the call site itself. Instead, the procedure address is loaded into a register (virtual or hard) and possibly moved one or more times before the call actually occurs. This requires us to do some simple local data-flow analysis to determine the target of the call. We do a linear scan of the instruction list while keeping a mapping of registers to their contents (if known) and doing copy propagation in the mapping when the contents of one register are moved to another. Whenever we reach a label in an instruction stream, we clear the mapping, conservatively assuming that the label could be a join point or the target of a control-transfer instruction. In addition, any other instructions with a given register as a destination will kill the current mapping for that instruction. We don't keep track of variable moves into registers, so this could at times cause our code to fail to find the procedure that is being called. In these cases, we cannot inline the call site. Instruction re-ordering could also make the call target inaccessible to our simple scanner by moving the initial load above a label.

The annotation for inlining depth exists to allow flexibility in how much nested inlining to do at each call site. For example, in some situations, one may want to inline a procedure but keep all the calls within the inlinee as calls. Or perhaps one may want to inline a recursive call three times. This inlining depth parameter allows for limited control of nested inlining at each call site instead of strictly requiring a global policy governing nested and recursive inlining.

As an example, consider a procedure called `inner` that is called in two locations. The first is in `main` as part of an initialization sequence, and the other is within a nested loop that is executed many times. `inner` itself contains several call sites, and one may determine that in the nested loop some or all of these calls should be inlined as well. Under our system, this is expressed by annotating the calls inside `inner` to be inlined, and then using different annotations for the two sites that call `inner`. Within `main`, the annotation would include a maximum depth of one, which the other call site would include a higher maximum depth. When the call in `main` is inlined, the annotation will ensure that no further inlinings occur as a result of `inner` being inlined. Without the limiting maximum depth, however, the inlining of `inner` at the other call site will then cause the annotated call sites within `inner` to also be inlined.

One can think of the inlining process as flattening a call tree. Each call that is inlined is a branch and each node is a procedure. Calls that are not inlined are not represented in this tree. If `main` inlines two procedures, it would have two branches, and if one of those procedures inlined three others, then it would have three branches. Two calls to the same procedure would be represented as separate branches. The maximum depth annotation guarantees that the tree below a given call site will not exceed the given depth, i.e., it prevents nested inlinings that might otherwise occur.

Perhaps the simplest case to consider is a recursive procedure. To inline a recursive procedure three levels deep, one would mark the call site within the recursive procedure to allow no more than two additional levels of inlining. Or, alternately, one could mark it to allow very deep inlining, and then annotate a maximum depth at each location where the recursive procedure is called. If the annotation within the recursive procedure allows for ten levels of inlining but a given call site only allows for four, the smaller number will govern what actually happens.

This does not allow infinite flexibility, but it does make the simple cases easy to handle (depth one annotations), recursive procedures can be handled in a straightforward way, and one still has the option of constructing more complex inlining sequences.

2.2 Collecting compilation units

Because `do_pre_inliner` has taken care of annotating call sites for inlining, the `do_comp_unit` method needs only to collect the compilation units of all the procedures in the file so that when the inliner finds a site marked for

inlining, it can test if it is legal to inline (see below) and copy the inlinee's instruction list in order to integrate it into the host procedure.

2.3 Finding sites to inline

A simple linear scan through the instruction list of each procedure in the `finalize()` method allows us to find which call sites have been annotated for inlining. It is at this point that we do a number of tests to make certain that it is safe and legal to inline that procedure. In particular, we test to make certain that the procedure being called is in our catalog of compilation units for the current file. Next we ensure that the procedure does not have a variable-length argument list, because the binding of formal parameters to arguments in that case becomes too complicated. If the call site passes both of these tests, then it is ready to be inlined.

2.4 Cloning

At this point, if we do not already have a copy of the host procedure (the one into which things are being inlined), we make a clone. The reason for this is that we will be modifying the original instruction list of the host procedure, and if later another procedure wants to inline the host, without a clean copy it will be getting all the nested inlining of the host even if it did not want to. We also want to ensure predictable inlining semantics; the final result of inlining should not depend on which order the procedures appear in the file. For example, if procedure `food` inlines procedure `foo`, and procedure `bar` inlines procedure `food`, with one level of inlining we would expect `bar` to inline `food` but not to inline `foo` from within the inlined copy of `food`. To address this issue, we keep a catalog of unchanged clones for procedures that get modified. We do lazy cloning, so a procedure is only cloned the first time it is changed. If a procedure does not inline any of its call sites, it will not be cloned.

When we prepare to inline a procedure, we get a clone of the inlinee's compilation unit to modify and then get rid of when we are finished. The presence of the inlinee's compilation unit in the clone map indicates that the original has been modified somehow; thus when inlining we should make a copy from the clone map, not from the original. Otherwise, it is safe to make our working copy from the original.

2.5 Removing/altering annotations

At the beginning of each procedure is a null instruction with an annotation indicating that it is the start of the procedure. It is important for us to remove this instruction so that later passes of the compiler do not get confused when finding information about a procedure start within another procedure.

We also need to find any inlining annotations on call sites within the inlinee, changing their inlining depth to reflect the minimum of the inlinee's depth and the call site's marked depth - 1. If this causes the maximum depth to become 0, then we remove the annotation altogether. As discussed above, the annotated value at a call site gives an upper bound on the depth of the inlining tree that will result from inlining that call.

Recursive/nested inlining is accomplished by restarting our linear scan for annotated call sites at the beginning of a newly-inlined procedure rather than at the end. Thus, if a call site within an inlinee still has an annotation, then the nested inlining will occur just as if it were part of the original host procedure.

2.6 Merging symbol tables

All symbols from the procedure being inlined must be incorporated into the host procedure's symbol table. Here, we must take care to avoid name clashes. Thus, when any symbol from the inlinee is inserted into the host's symbol table, we first check for name conflicts and rename the inlinee's symbol if necessary.

2.7 Parameter symbol substitution

We replace symbols from the formal parameter list in an inlined procedure with ordinary `VarSym` symbols so that later passes will not encounter parameter symbols in unexpected places. While we were making this change, Glenn was changing the later passes to handle `ParameterSymbol`s that occur in unexpected places. As far as we have been able to determine, both solutions were successful.

2.8 Renumbering virtual registers

Virtual registers in the host and inlined procedures can also conflict, and merging the symbol tables does not resolve the conflicts. As we copy the instructions from the inlinee to the host, we apply an `OpndFilter` that renumbers virtual registers. This same filter does parameter symbol substitution as well.

2.9 Machine-specific entry code

Each architecture has different calling conventions. For example, SUIFvm includes its arguments as part of the call itself; thus the arguments must be bound to the formal parameters of the inlinee. In contrast, the Alpha expects arguments in registers `$16--$21` with any overflow on the stack, so prior to each call instruction the arguments are moved into the appropriate places, and at the beginning of each procedure the arguments are moved into the formal parameter variables.

We never received final instructions on how the multiple, target-specific versions should be integrated into the Machine SUIF2 architecture, so manual selection is required. To invoke the proper inliner, edit `suifpass.h`, choosing `InlinerAlphaPass` or `InlinerSUIFvmPass` as appropriate. This will instantiate the proper subclass of InlinerPass that handles the machine-specific parts of the pass correctly. The different versions of the inliner override a virtual method that provides a series of instructions to be placed at the beginning of the inlined procedure. The Alpha inliner inserts no extra entry code. However, the SUIFvm inliner inserts move instructions to copy the arguments as specified in the call instruction to the formal parameter variables as indicated in the inlined `CompUnit`.

2.10 Inserting the instruction list

Once we have inserted the machine-specific entry code, we walk through the instructions in the inlinee, editing the instructions as needed and inserting them one at a time into the host procedure. Whenever we reach a return site, we replace the return instruction with machine-specific exit code.

2.11 Machine-specific exit code

As with the procedure entry code, we deal with the return sites of the inlined code in a machine-specific manner. At each return site in the inlinee, we need to insert one or more instructions. For the Alpha target, calling conventions already dictate that the result be put in `$0`, and the calling procedure expects to find it there. So all that is needed is for each return to be replaced with an unconditional branch to a label immediately following the inlined code. This label is inserted in all cases; we replace the call site of the procedure being inlined with a label and insert all the inlined and any other new instructions before the label.

For SUIFvm, things are slightly more complicated. The call instruction indicates into which virtual register it expects the result, and the return instruction actually carries the value that goes into that register. Thus, at each return site, we replace the return with a move, putting the value from the return instruction to the location specified by the call instruction. Then, as with the Alpha target, we add an unconditional branch to move control past any other code being inlined.

As with the machine-specific entry code, each subclass of the basic inliner overrides a virtual method that constructs the appropriate sequence of instructions to replace the return instruction.

3 Using the inliner

3.1 Invoking the main pass

The inliner pass is invoked just as any other compiler pass, with the command `do_inliner <inputfile> <outputfile>`. This will read in the file and annotate call sites according to the currently implemented policy, writing the result to `<outputfile>`, which can be used as input to other compiler passes (we suggest constant propagation and/or dead code elimination). To choose either the Alpha target or the SUIFvm target, edit `suifpass.h`. Under `class inliner_pass`, uncomment the correct line, either `InlinerAlphaPass inliner;` or `InlinerSUIFvmPass inliner;`. Use `gmake` with the supplied `Makefile` to compile the version of the inliner that you wish to use.

3.2 Call site annotations

The annotation decisions are made in the `do_comp_unit` method in `pre_inliner.cpp`. Look at the existing code for a model of how to make the annotations. The statistics collected in `do_comp_unit` are available from the class field `proc_map`, which maps procedure names (`LStrings`) to a `struct` containing one `CompUnit*` (field `cu`) and two integers, fields `size` and `call_sites`. Compile as usual and use as indicated above.

An annotation should be placed on each call site that should be inlined. The annotation should be of type `k_inline` and should contain a string indicating the name of the target procedure and an integer indicating the maximum inlining depth for this site. the `do_pre_inliner` pass steps through the instruction list of its input file, determines the target of each call site (when it can), and prompts the user for the inlining depth to be applied at that site.

3.3 Known problems

When inlining Alpha code with too many arguments to fit in the conventional registers, the code produced will not work correctly. This is due to the expectations of a later pass that are violated by inlining, and the solution is not yet available.

When passing structures by value, incorrect results are sometimes obtained. This may be related to the above problem, but we haven't conclusively determined the problem yet. For simple cases, it works fine. The tests seem to break only when large structures are passed, suggesting that the problem may be with stack handling.

Inlined Alpha code references hard registers that are no longer strictly necessary, and the register allocator assumes that such references are for a reason. While this doesn't affect the correctness of the final code, it would be preferable to either do some copy propagation in the inliner or to change the register allocator to recognize this situation and rename the registers as appropriate.

4 Acknowledgements

Special thanks go to Glenn Holloway and Mike Smith who endured our bug-finding and assumption-shattering in various parts of Machine SUIF2. Their willingness to change infrastructure and add methods made the inliner possible.