Implementation of the Mark-Sweep Dead Code Elimination Algorithm in LLVM
Yunming Zhang, Rebecca Smith, Linge Dai, Max Grossman

1. Background

Dead code elimination (DCE) is the deletion of instructions which can never be executed or whose execution we can reliably prove has no impact on the state or results of a program. Dead code can come from a number of sources including intermediate compilation steps, optimization, or a programmer [1].

DCE has a number of benefits. First, it reduces the size of executable files. While this may not seem relevant in the modern age of multi-GB personal computers it can still have an impact in a number of areas. In embedded programming or other environments with constrained resources, smaller binaries free up more storage for data. In general, smaller binaries allow a larger percentage of the application to fit into code cache. Consider the case of a many-iteration for-loop with a large body. Perhaps without DCE the for-loop cannot fit into the code cache, inducing cache thrashing. With DCE and the better cache locality that results, general purpose application performance can improve.

While there are many algorithms and approaches to DCE, this work focuses on the Mark & Sweep (MS) algorithm presented in “Efficiently computing static single assignment form and the control dependence graph” [1], including improvements by Shillingsburg [2]. MS is a two-stage algorithm: Mark followed by Sweep. At a high level, the goal of the Mark phase is to mark all instructions as either critical to correct program execution (live) or not (dead). The Sweep phase then passes through the instructions and removes all dead instructions, performing transformations as necessary for the modified code to remain correct. The algorithm for MS is shown below:
This algorithm raises a number of questions, including:

1. What is the definition of a critical instruction? In the original paper, a critical instruction is defined as an instruction which is "initially assumed to affect program output"; examples would be I/O operations or return statements. How can we discover critical instructions in LLVM?
2. What is the algorithm for calculating the RDF for each block? How can this be implemented in LLVM?
3. How can post-dominators be calculated?
4. How can different types of instructions be identified for special processing?
5. How to delete, insert, and replace existing instructions in a real world compiler?
6. What is the intuition behind rewriting unmarked branches to jumps? An unmarked branch implies that no paths from that jump end in a critical instruction. Rewriting to a jump maintains a proper CFG for future optimizations and handles some edge cases.

The answers to these questions, particularly as they pertain to an implementation in LLVM as part of our project, will be discussed in the next section.

This report will detail the design and implementation of a DCE pass in LLVM based on the Mark-Sweep algorithm. The main points covered in this report will be:

1. Our own understanding of the Mark-Sweep algorithm
2. How we implemented Mark-Sweep in LLVM
3. Any changes or additions made to the Mark-Sweep as part of its implementation in LLVM
4. Evaluation of our implementation, and comparison to existing LLVM DCE passes

In general, we try to provide information on the steps taken and lessons learned as part of this work.
2. Implementation

In LLVM, optimizations are structured as transformation passes. During compilation, any number of such passes can be run in succession. All transformation passes must implement LLVM’s Pass class. LLVM further distinguishes between several subclasses of Pass, depending on the scope of optimization: ModulePass, CallGraphSCCPass, FunctionPass, LoopPass, RegionPass, and BasicBlockPass. To implement Mark-Sweep FunctionPass was the natural choice to extend, as the Mark-Sweep algorithm is intra-procedural. The following section will start with a discussion of LLVM knowledge gained, and then cover the implementation of the MS DCE algorithm in three chunks: Initialization, Mark, and Sweep.

General LLVM Implementation Learnings

At the core of most LLVM transformation passes is the manipulation of LLVM instructions. LLVM provides a general Instruction class with subclasses representing different instruction types. Most pertinent to DCE is the TerminatorInst subclass, a superclass of all operations that end a basic block. These include BranchInst, IndirectBrInst, SwitchInst, InvokeInst, ReturnInst, ResumeInst, and UnreachableInst.

Terminator instructions are any instruction that terminates a basic block. Terminator instructions play several roles in our algorithm. For example, in the Initialization phase we define all ReturnInst as critical instructions. In the Sweep phase, handling branches and jumps which have not been marked are two special cases. Thus, we must be able to identify branches and jumps in LLVM. Since a basic block is a maximal length sequence of straightline code, jumps and branches cannot appear anywhere besides the very end of a basic block. Therefore jumps and branches must fall under LLVM’s terminator class. This class contains a method, getNumSuccessors(), which returns the number of basic blocks that are potential targets of that instruction. Using getNumSuccessors(), jumps can be identified as TerminatorInst with a single successor, whereas branches are those with more than one successors. Note that some terminator instructions -- ReturnInst, ResumeInst, and UnreachableInst -- will always have zero successors and thus fall into neither category.

In addition to understanding the classification of instruction types at the core of LLVM’s IR, we needed to understand how to remove and rewrite these instructions. Removing instructions in LLVM is fairly trivial; LLVM's Instruction class offers an eraseFromParent() method which removes the instruction from its containing basic block. However, there are two caveats. First, instructions cannot be removed from a function while using its iterator over the instructions, as calling eraseFromParent() disrupts this iterator. Thus, rather than immediately removing instructions during the Sweep phase, we store them in a list to be processed after the entire sweep is complete.

More interestingly, the order of removal during this post-processing matters. The eraseFromParent() method performs several assertions to ensure that the instruction can be safely removed; among them is a check that no instructions using the instruction’s resulting value remain. The existence of circular dependencies complicates this further. To safely remove an instruction, we first determine whether the result of the instruction is non-void. If so, we replace all uses of this definition with a Null constant. The type check utilizes the Instruction class’ getType() in conjunction with the type class’ isVoidTy(). The replacement invokes the Instruction class' replaceAllUsesWith() with the Constant class's getNullValue() as its parameter. This takes advantage of the fact that if an instruction is non-critical all instructions depending on it must also have been deemed non-critical; Null operands are only added to instructions that are about to be removed anyway.

In addition to removing instructions, the Sweep phase also potentially rewrites some instructions. The simplest way to accomplish this in LLVM is to remove the old instruction and insert a replacement. Inserting instructions is a one-line operation. Each LLVM Instruction type defines Create() methods which take as their parameters all relevant operands, as well as one additional parameter representing the instruction before which to insert the newly-created instruction. Create() then constructs the new instruction and inserts
it at the appropriate offset by modifying internal LLVM structures. Note that while removing instructions during iteration is dangerous, inserting during iteration is safe as long we insert prior to the current instruction.

Finally, a function’s post-dominator tree (PDT) is useful in both the Mark and Sweep phase. LLVM provides getAnalysis<PostDominatorTree>() to retrieve the PDT for the current function.

**Initialization Phase**

The initialization phase consists of a single pass over all instructions in the function. As shown in the background section, the initial set of critical instructions is composed of those which directly affect the output of the program. In LLVM terms, a critical instruction meets one of two conditions: it is either a return statement, or it “may have side effects”. For each instruction, these two conditions are checked via a type check against ReturnInst and a call to the instruction’s mayHaveSideEffects() method, respectively. LLVM breaks down mayHaveSideEffects() into two cases: mayWriteToMemory() or mayThrow(). Naturally, those instructions that throw exceptions merit marking as critical, since exception throwing may result in a dramatic change in control flow, such as early termination of the application. Instructions that may write to memory are kept due to the difficulty of disambiguating memory accesses at compile time. We cannot always identify which stores affect memory from which critical loads will later read, and therefore the safe and conservative approach is to treat all such instructions as critical.

The initialization phase also initializes the set of all “useful” blocks. A block is defined as useful if it includes one or more critical instructions. Useful block are relevant to the Sweep phase of the algorithm, which requires calculation of the nearest Useful post-dominator. Based on this definition, each time we mark an instruction we also add its containing block to the set of useful blocks.Performing this bookkeeping during the Initialization and Mark phases improves the simplicity and efficiency of the Sweep phase.

**Mark Phase**

The Mark phase propagates marks from the critical instructions identified during Initialization backwards along the def-use chains. As shown in Section 1, this is accomplished by iterating on a work list of instructions, which is populated with critical instructions as they are marked. The work list is implemented as a set from the C++ standard libraries. Because the work list is initialized to contain the set of known critical instructions and new instructions are added to the list only upon being marked, we know that the instructions popped are marked and therefore critical. Thus, once the next instruction has been obtained from the queue, several steps must be taken to process it.

One key operation of the Mark phase is finding the defining instruction for each operand of each critical instruction. In LLVM this is relatively straightforward, as the Instruction class provides an iterator over the operands’ defining instructions. However, in the case of an operand that is a constant, the iterator will yield the constant itself rather than an instruction. We identify these constant operands by checking whether they can be cast to an Instruction type and skipping them as appropriate. For the remaining operands, we check their defining instructions and if they are not already marked, we mark them and push them onto the worklist.

Another key operation is the actual marking of instructions. We considered several implementations for this step. One candidate implementation utilized the built-in metadata field of the Instruction class. Implemented as a hash table, metadata can be added to an instruction as a (key, value) pair, where the key is an integer and the value is a pointer to an instance of LLVM’s MDNode type containing the metadata. This metadata can then be retrieved quickly by a keyed lookup. This implementation provides O(1) marking and checking. While we had this implementation working on some builds of LLVM, there were unresolved difficulties getting it to work across all platforms and so we rejected it in favor of a less platform-specific implementation.

In place of using LLVM metadata, we used the LLVM SmallPtrSet data structure to store Instructions which
have already been marked. Marking, unmarking, and checking an instruction then translate into insert(),
erase(), and count() calls on SmallPtrSet.

The Mark phase also requires the computation of the Reverse Dominance Frontier (RDF) of each block. To
determine how best to compute the RDF sets in LLVM, we first studied its existing support. There is
existing support for computing dominance frontiers (DFs) based on the algorithm presented in [1]; likewise,
there is support for computation of the PDT. Furthermore, we discovered that LLVM's CFG provides each
block with access to its predecessors as well as its successors, allowing us to walk the CFG backwards.
Finally, we noted that LLVM's DF computation differs slightly from the algorithm in [1] in that it computes
the DF on a per-block basis rather than on the CFG in its entirety. Having made these observations, we
decided that the best way to leverage LLVM's existing infrastructure was to modify its DF computation
algorithm to compute the RDFs on the CFG.

The complete LLVM-specific RDF computation algorithm is shown in the Appendix.

This algorithm computes the RDF of a specific block X by first adding to its RDF all predecessors of X which
does not immediately post dominate. Then the algorithm expands X’s RDF with all of the blocks that X
does not post dominate but which are in the RDF of the blocks X immediately post dominates.

Shown in the appendix, this algorithm iterates until a WorkList is empty. The WorkList is implemented as a
dynamic list of WorkObjects, which is used as a stack. A WorkObject contains the block being visited
(currentB) as well as the block previously visited (prevB). prevB is there to check whether currentB is the
input block (if prevB is null), and to propagate currentB’s RDF to its immediate post-dominator. In the
implementation, it also contains references to the nodes in the PDT that represent currentB and prevB. A
dictionary (visited) is created to record the number of times a block has been visited. The algorithm visits
each block only once.

The WorkList is seeded with a WorkObject containing only the input block. Within each iteration, we check
if the current block has been visited. If it has, there is no need to compute the local RDF. Otherwise, we
iterate through the predecessor blocks of the current block and add to the local RDF those whose
immediate post-dominator (IPDOM) is not the current block. LLVM provides a function getIDom on the node
in the PDT to obtain the IPDOM of the node.

After the algorithm finishes computing the local RDF of the current block, it iterates through the blocks that
the current block immediately post dominates and pushes a WorkObject to the WorkList containing the
current block as the previous block and the post dominated block as the current block. The PDOMI set of
the current block is obtained by iterating through the successor nodes of the node that represents the
current block in the PDT. visitedPDom is a boolean value to indicate whether any of the block in PDOMI has
been visited. If all of the blocks are visited (visitedPDom is false), their RDFs are assumed to be complete
and can be added to the RDF of currentB.

The last part of the algorithm propagates the RDF information of the current block back to its IPDOM. It first
checks if the current block is the input block by checking if the previous block is null. If it is, the RDF of the
input block is complete and the algorithm finishes. Otherwise, the algorithm iterates through the blocks in
the current block’s RDF and uses PDT.properlyDominates() to filter out blocks that are strict (not equal)
descendents of prevB in the PDT. This is essentially the same as checking whether prevB is each block’s
IDOM. The rest of the blocks are added to the RDF of prevB. The algorithm then pops the current
WorkObject from the WorkList and continues to the next loop iteration.

One final addition made to the Mark algorithm in our LLVM implementation is handling of Phi instructions.
The need for this addition arose during testing, and is related to optimizations performed by the mem2reg
pass, which will be discussed in the Evaluation section. As an example, consider the LLVM bytecode
below, generated from the mem2reg pass:
define i32 @foo(i32 %a, i32 %b) nounwind uwtable {
  entry:
  %add = add nsw i32 %a, %b
  %sub = sub nsw i32 %a, %b
  %cmp = icmp sgt i32 %add, %sub
  br i1 %cmp, label %if.then, label %if.else
  if.then:
    br label %if.end
  if.else:
    br label %if.end
  if.end:
    %c.0 = phi i32 [ %add, %if.then ], [ %sub, %if.else ]
    ret i32 %c.0
}

While the defining instructions of the Phi instruction are easily identifiable as the assignment to %add and %sub in the entry block, the Phi function needs to know from which basic block execution arrives in order to select the correct value for assignment to %c.0. In a way, that information is meta-input to the Phi instruction. As a result, our implementation marks the block-ending instruction for each block referenced in a Phi instruction. Without these modifications, the jumps in the if.then and if.else blocks would not be marked, they would eventually be removed by the Sweep phase (more information on this later), and information critical to the Phi function would be lost.

Sweep Phase

The Sweep phase is a single pass over all unmarked instructions in the input function. For each unmarked instruction, we first use type checking and LLVM’s Instruction classes to check if the instruction is not a TerminatorInst or CallInst. If it is not, we add it to a list of instructions to be removed and continue to the next unmarked instruction to process.

Call instructions are always a special case, and are never deleted. The reasoning behind this is that Mark-Sweep is an intra-procedural algorithm, and making changes to the inter-procedural structure of the application is beyond the scope of this work. For example, consider the case where the called function writes to a global variable. It would not be possible to determine if the callee has any side effects, and so removing the Call may invalidate the correctness of the program.

TerminatorInst is potentially a special case. Recall that a TerminatorInst ends a basic block, and may have zero or more successors. If a TerminatorInst has more than one successor (i.e. is a branch) but is unmarked (non-critical), the Mark-Sweep algorithm calls for replacing that branch with a jump to its nearest useful post-dominator. Calculating the nearest useful post-dominator is accomplished by combining the Useful blocks discovered during the Initialization and Mark phases with the PDT retrieved from LLVM. Our implementation walks the PDT of the current basic block and selects the first basic block which is marked as useful to be the nearest useful post-dominator. The TerminatorInst being processed can then be replaced by an unconditional BranchInst to that post dominator by marking the TerminatorInst to be removed and inserting a BranchInst using Create(), as described earlier under general LLVM learnings.

3. Previous Work

LLVM includes three transformation passes that implement dead code elimination: the dead code elimination (DCE) plugin, the aggressive dead code elimination (ADCE) plugin, and the simplify CFG (SCFG) plugin.

DCE Plugin
DCE is a dead code elimination pass which works with the Static Single Assignment (SSA) form provided by LLVM. The transformation is an implementation of a classic fixed point algorithm using a single worklist. It begins by adding all of the instructions to a worklist. It then processes each of the instructions by popping them from the worklist one at a time and identifying “trivially dead” instructions as those whose values are never used. Each time an instruction is marked as dead, its defining instructions are added to the worklist, and the removal of dead instructions potentially creates more dead instructions. The algorithm repeats the process of popping and adding instructions until the worklist stabilizes. In the end, all the instructions that are not erased are live instructions.

Compared to the Mark & Sweep algorithm implemented in this work, DCE is:

1. Less aggressive. Our algorithm begins by assuming all instructions are dead and then successively marks live instructions. In contrast, DCE is a pessimistic algorithm and initially adds all instructions to the worklist as live, narrowing the set of live instructions as it discovers those that are dead.
2. Less complete. Mark-Sweep doesn’t assume all branch instructions are critical. Instead, we rewrite them as jumps to the nearest useful post-dominator if needed. This allows us to remove more instructions in the Sweep stage.

**ADCE Plugin**

Aggressive Dead Code Elimination takes an alternate approach to eliminating dead instructions. Rather than identifying dead instructions, it identifies live instructions. A live instruction is either a Terminator Instruction, Debug Instruction, a LandingPad Instruction, or an instruction that may have side effects.

Like the DCE plugin, it implements a fixed point algorithm using a worklist. In an initialization stage, it iterates through all the instructions in the program. For each instruction that is currently live, the algorithm marks it as live and adds it to the worklist.

Once the initial worklist is created, the algorithm pops instructions which are already marked as live from the worklist one at a time. For each instruction popped off the worklist, ADCE identifies the instructions that define its operands. The newly identified defining instructions are marked as live and added to the worklist. The algorithm repeats itself until the worklist stabilizes. In this way, ADCE is able to propagate the initial set of live instructions to include all the live instructions.

Compared to the Mark-Sweep algorithm implemented in this work, ADCE shares the same weaknesses with DCE.

**SCFG Plugin**

The SimplifyCFG (SCFG) plugin tries to find or create basic blocks that can be entirely deleted from a function. It uses a number of techniques to do this. First, it analyzes instructions within a basic block to find instructions which are unreachable, and marks them as so. For example, any instructions in a basic block which follow a function call that does not return are unreachable. It also looks for basic blocks whose only useful instruction is a return. These can be merged into earlier basic blocks, eliminating the need for branches and phi functions. Additionally, it discovers which basic blocks can be removed in entirety by traversing the CFG from the function entry point and marking basic blocks encountered as alive.

This algorithm differs from our implementation and the other plugins discussed because it performs removals at the granularity of basic blocks, and does not actually remove individual instructions. This limits the amount of dead code elimination it can perform, but makes it complementary to the 3 other dead code elimination plugins: DCE, ADCE, and our Mark-Sweep pass.
4. Evaluation

This section compares the effectiveness of three dead code elimination LLVM plugins on a suite of test cases in the C programming language developed by the authors. The plugins tested are the DCE, ADCE, and our Mark-Sweep pass as described in Section 2 and 3. The benchmarks used were constructed with the goal of testing each part of our dead code elimination implementation.

The metric used in this comparison will be number of instructions removed. Some previous work has compared execution time before and after dead code elimination. We chose not to use execution time as a metric for three reasons. First, the tests used here are small enough that the variation in execution time due to removed instructions would not be significant or interesting. Second, using execution time would make these tests dependent on the input data set used, possibly skewing results. Third, while dead code elimination may remove instructions that would normally be executed but not have any effect on program execution, it can also remove instructions that would never be executed in any program execution. Using execution time as a metric would hide the impact of removing these instructions because they would not be executed in either the baseline or optimized version.

In addition to running a single pass of each dead code elimination plugin, we also test how these plugins enable other optimizations by running a pass of the SimplifyCFG LLVM plugin following a pass of each dead code elimination plugin. An important part of code optimization is correct and optimal interleaving of multiple compiler passes. It is therefore vital to show that our plugin is a good member of this compiler ecosystem.

The graph below shows the results obtained from tests of the plugins with and without SimplifyCFG. The table below summarizes the performance of our dead code elimination plugin relative to the other two with and without a SimplifyCFG pass. All results show that in general, our plugin is able to perform equivalently or better than current LLVM plugins on benchmarks that test a wide variety of C language constructs. Note: we are just as surprised at how positive these results are as you are.

To understand where our DCE plugin is improving on the existing passes, we compare performance of our pass against DCE without SimplifyCFG on testEraseLotsOfInstructions. On testEraseLotsOfInstructions, our plugin was able to remove 10 instructions compared to 4 instructions by DCE. This was the result of the DCE plugin being unable to identify some instructions as dead because it assumes that all block-ending instructions (branches, jumps) are live. As a result, many instructions which are actually dead cannot be marked as dead, simply because they define values which are used in a branch comparison.

<table>
<thead>
<tr>
<th>Mark-Sweep performance relative to...</th>
<th># Benchmarks Worse</th>
<th># Benchmarks Same</th>
<th># Benchmarks Better</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCE</td>
<td>0</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>DCE w/ SCFG</td>
<td>0</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>ADCE</td>
<td>0</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>ADCE w/ SCFG</td>
<td>0</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
In order to prove that our algorithm can work with real applications, we also tested our optimization on some of the larger existing benchmarks in the LLVM test suite. We ran the optimizations on Stanford Benchmark, which included popular applications such as Quicksort and Queens. These benchmarks had 200 - 300 lines with multiple functions. The number of instructions eliminated is not presented because neither our pass or the existing LLVM DCE passes could eliminate significant number of instructions, but we have proven that our algorithm works on larger examples.

One challenge encountered during evaluation was that the mem2reg pass, which promotes the LLVM IR to SSA form, also performs some basic dead code elimination. LLVM requires that the IR be in SSA form prior to performing any transform passes and mem2reg is the only existing pass that converts into SSA form. Thus, immediately prior to running our transformation pass, we always run mem2reg. For our earliest and simplest test cases, mem2reg ended up performing the DCE that we had hoped our own pass would. However, with the construction of a new and more complex test suite, mem2reg's partial dead code elimination met a limit and the results of our pass became observable.

5. Conclusions

In this report we present the successful design and implementation of an LLVM pass based on the DCE algorithm in [1]. Details of our implementation provided in Section 2 show that we navigated the various libraries, APIs, and utilities included with LLVM to build an efficient and correct compiler pass. As a result, we learned about integrating our own code into a large-scale and industry-quality compilation system. As icing on the cake, the evaluation of this work shows comparable or better performance by our DCE pass relative to two other pre-existing LLVM DCE passes on a suite of test cases using number of instructions removed as a metric for comparison.

6. Citations


7. Appendix
Data: b representing the block of which the algorithm computes the RDF
Data: PRED(x) representing the set of predecessors of the block x in the CFG
Data: IPDOM(x) representing the immediate post dominator of the block x
Data: PDOMI(x) representing the set of blocks the block x immediately post dominates
Result: RDF(b) representing the Reverse Dominance Frontier of the block b

worklist $\leftarrow \emptyset$
// dictionary to record the number of times a block has been visited
visited $\leftarrow$ new dictionary
// the first element of the item in the worklist represents the node being visited
// the second element of the item in the worklist represents the node previously visited
worklist.push((b, null))

while worklist $\neq \emptyset$ do
    // get the first item from the worklist without popping it
    (currentB, prevB) $\leftarrow$ worklist.top()

    // if the current block has not been visited
    if visited(currentB) = $\emptyset$ then
        // marking the current block to be visited
        visited(currentB) $\leftarrow$ 1
        for Block p in PRED(currentB) do
            // add the predecessors that current block does not post dominate to RDF
            if IPDOM(s) $\neq$ currentB then
                RDF(currentB) $\leftarrow$ RDF(currentB) $\cup$ s
            end
        end
    end

    // visit the blocks the current block immediately post dominates
    visitedPDom $\leftarrow$ false
    for Block pdomB in PDOMI(currentB) do
        if visited(pdomB) = $\emptyset$ then
            worklist.push((pdomB, currentB))
            visitedPDom $\leftarrow$ true
        end
    end

    // if all the post dominated blocks are visited
    if visitedPDom = false then
        // if the currentB is the original input block b, RDF(b) is complete
        if prevB = null then
            break
        end
        // otherwise, add blocks in RDF(currentB) that prevB does not post dominate to RDF(prevB)
        for Block pdfB in RDF(currentB) do
            if prevB does not post dominate pdfB then
                RDF(prevB) $\leftarrow$ RDF(prevB) $\cup$ pdfB
                worklist.pop()
            end
        end
    end
end