Compiler Optimizations for a Time-constrained Environment

Owen C. Anderson

Prof. Susan Fox, First reader

Prof. Libby Shoop, Second reader
Prof. Andrew Beveridge, Third reader

April, 2008

Macalester College

Department of Mathematics and Computer Science

Abstract

Over the last several decades, two important shifts have taken place in the computing world: first, the speed of processors has vastly outstripped the speed of memory, making memory accesses by far the most expensive operations that a typical symbolic program performs. Second, dynamically compiled languages such as Java and C# have become popular, placing new pressures on compiler writers to create effective systems for run-time code generation.

This paper addresses the need created by the lagging speeds of memory accesses in the context of dynamically compiled systems. In such systems memory access optimization is important for resultant program performance, but the compilation time required by most traditional memory access optimizations is prohibitively high for use in such contexts. In this paper, we present a new analysis, *memory dependence analysis*, which amortizes the cost of performing memory access analysis to a level that is acceptable for dynamic compilation. In addition, we present two memory access optimizations based on this new analysis, and present empirical evidence that using this approach results in significantly improved compilation times without significant loss in resultant code quality.

Contents

Abstract **iii**

Contents **v**

List of Figures **vii**

List of Tables **ix**

Acknowledgments **xi**

1 Introduction **1**
- 1.1 Modern Static Optimization 2
- 1.2 Evolution of Just-in-Time Compilation 3
- 1.3 Previous Work 4
- 1.4 Implementation: LLVM 4

2 Alias Analysis **7**
- 2.1 Classical Alias Analysis 7
- 2.2 Basic Alias Analysis 8
- 2.3 Memory Dependence Analysis 8
- 2.4 Results 10

3 Dead Store Elimination **13**
- 3.1 Classical Dead Store Elimination 14
- 3.2 Fast Dead Store Elimination 15
- 3.3 Results 16

4 Redundant Load Elimination **19**
- 4.1 Global Common Subexpression Elimination 19
- 4.2 Global Value Numbering 22
- 4.3 Results 22

5 Conclusions **25**
5.1 Results 25
5.2 Future Work 27

Bibliography **29**

Code Listing **31**

List of Figures

1.1 An example control-flow graph 2
1.2 A simple example in C and LLVM IR 5

2.1 An example of memory dependence analysis 9
2.2 A diagramatic example of the internals of memory dependence analysis 10

3.1 An example where the first `store` is potentially necessary 13
3.2 An example in which the first `store` is dead 14

4.1 An example where the `load` is not redundant 20
4.2 An example where the `load` is redundant 20
4.3 The result of eliminating a redundant `load` 21

5.1 Total time to execute the optimizations in seconds of the four largest testcases from SPEC 26
5.2 Normalized execution time of the four largest testcases from SPEC 26

List of Tables

2.1 The number of alias queries evaluated on the SPEC benchmarks 11

3.1 The number of stores removed on the SPEC benchmarks 16

4.1 The number of loads removed on the SPEC benchmarks 23

Acknowledgments

The author was supported in this work by Apple, Inc. He would particularly like to thank Dr. Chris Lattner for helping create many of the ideas that developed into this work, as well as his other colleagues at Apple. He would also like to thank the entire LLVM community for their support over the years. In addition, he would like to thank Stephanie Abascal for all of her support through this process.

This capstone paper was produced using the Harvey Mudd College thesis template package in LaTex, created by Clare M. Connelly.

Chapter 1

Introduction

As modern software engineering practices introduce more layers of abstraction into the common programming model, optimizations performed at compile-time are increasingly important to achieving acceptable performance in statically compiled programs. Because the programmer is becoming further and further removed from the machine, he or she relies on the compiler to manage the machine's resources efficiently and to eliminate inefficiencies introduced by the nature of the high-level language used. Each additional layer of abstraction requires additional analysis and optimization to reduce or remove the performance impact, leading to a trend of sharply increasing compilation times.

In contrast to the world of statically compiled programs, *Just-in-Time* (JIT) or *dynamic* compilation is all about compilation speed: the program is compiled function-by-function as needed, directly into an executable buffer. Because the compilation delay is visible to the client, an effective JIT system must weigh the benefits of decreased execution time of a function against the costs of optimizing. Most production systems (such as the Java Virtual Machine and the Common Language Runtime) employ a "hot spot" method, which initially compiles functions without optimization, and then recompiles and optimizes them if it detects that they are being executed frequently.

The goal of this paper is to take optimizations designed for the static case and recast them for the dynamic, by choosing only to optimize the cases that occur most frequently in real-world programs, rather than waste time catching the less common ones that will not have a sufficient payoff in decreased execution time. We have chosen to focus our efforts on one analysis, *alias analysis*, and two optimizations that depend on it, *dead store elimination* and *redundant load elimination*.

These choices are based on the observation that, in most symbolic programs, the single most expensive operation is memory access. Alias analysis is a fundamental (and typically quite expensive to compute) analysis for most optimizations related to reducing memory accesses, while dead store elimination and redundant load elimination are the two most elementary applications of this information to the removal on unnecessary memory accesses. By making these available in a JIT environment, we hope to optimize away the largest time sinks in the compiled programs without wasting too much additional compilation time.

1.1 Modern Static Optimization

Before proceeding much further in the discussion of these novel techniques, it is necessary first to review the basics of modern compiler optimizations, so that these new algorithms can be understood in contrast to what has come before them.

In compiler optimizations, a unit of code executed in a straight line without control flow is known as a *basic block*. A function, then, is a directed, rooted graph of basic blocks, called the control-flow graph or CFG, where the directed edges represent the flow of control between basic blocks created by branches. Because of this, many optimizations have the form of graph algorithms. An example of visualizing a function as a CFG appears in Figure 1.1.

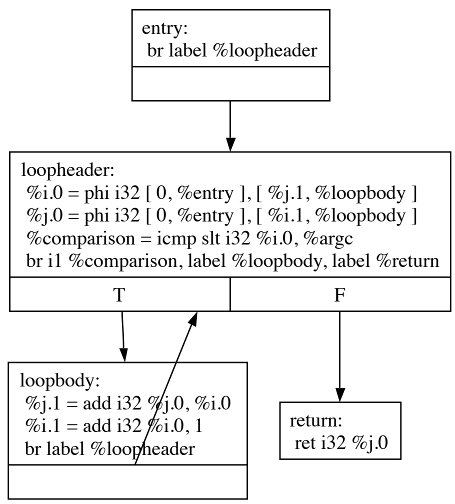


Figure 1.1: An example control-flow graph

The single greatest development in the last few decades of compiler research, and a prerequisite for the optimizations presented in this paper, is Static Single Assignment form, originally described in [11]. Fundamentally, SSA form is a semantics-preserving code transformation that facilitates a number of analyses and optimizations. The underlying concept is that, in SSA form, every register appears on the left-hand side of an assignment only once in the entire function.

A special operation, the ϕ instruction, is introduced in SSA form to merge registers coming in from multiple predecessor blocks, such as after an *if*-statement or a loop header. Two ϕ instructions appear in Figure 1.1, representing two variables whose values are initially zero, and then inherit their value from calculations in the loop body after each iteration. This example also illustrates that it is *Static* Single Assignment form: several of the registers will be assigned to more than once in the actual execution of the program (dynamic behavior), but they are on the left-hand side of only a single assignment statement in the written form of the program (static behavior).

The key benefit of representing a program in SSA form is that optimizations do not need to contain logic to guarantee that the value in a register is constant within their range of optimization; in SSA form, a value and the register it is stored in are equivalent. This greatly simplifies the work of writing an optimization, as well as making various analyses (particularly liveness analysis) much simpler. Of course, programs are not typically written in SSA form by the programmer, so a great deal of research exists into both creating [4] and destroying [6] SSA form efficiently and with optimal results.

1.2 Evolution of Just-in-Time Compilation

Just-in-Time compilation is hardly a novel idea, with the earliest recognizable proposal for it dating back to a 1960 paper [18] on the compilation of LISP. While it is beyond the scope of this paper to give a full account of the history of dynamic compilation (for that, see [3]), it is important to understand the origins of the idea and its practical applications. This will allow us to better understand why cheap optimizations are important for the development of effective JIT systems.

The earliest JIT systems were created as a memory-saving optimization for early computers that had very little storage, either primary or secondary. In general, high-level source code is more dense than binary executables because a significant amount of the program semantics is implicitly defined by the language in a high-level program, while these semantics must be explicitly represented in the executable binary. The natural solution to this problem was interpretation, direct execution of the high-level source code by an interpreter, but this has serious performance consequences. The very first JIT systems were created to solve this problem. Implementations of early languages like LC^2 [19] and APL [1] were created in this manner.

The next major advancement in the creation of JIT systems was the idea of a "mixed mode" system, wherein frequently executed parts of a program are dynamically compiled while less frequently executed parts are simply interpreted. This concept was independently proposed in [12] and [13]. This approach was first used for implementing BASIC, but was adapted to Fortran, with added infrastructure for heuristically choosing which function to compile, in [14]. This "hot spot" approach remains popular to this day.

Dynamic compilation approaches were later used for optimizing dynamically typed languages, like Self. In such languages, the types of variables are often undecidable at compile time; at run-time, however, a great deal more typing information is known, allowing the JIT compiler to apply optimizations that would not have been possible earlier. The Self compiler was developed in three generations of increasingly sophisticated run-time optimizations based on increased run-time type information. A description of its implementation is found in [22].

The most recent research in JIT compilation has largely been driven by two forces: optimizing the Java Virtual Machine, and binary translation. The former was spurred largely by the success of Java as a teaching and research platform, and the realization that its initial interpreted implementation was unacceptably slow. The list of papers on this topic is too long to reproduce without subdivision by category. [3] offers a survey of the relevant works.

Binary translation is an active area of research in the use of JIT compilation techniques for the evaluation of binary executables for one architecture on a different architecture. Rather than merely simulate the source architecture, binary translation systems treat the input executable as source code and dynamically compile and executed semantically equivalent code for the target architecture. Dynamic instrumentation systems, such as `valgrind` [20], are a subset of binary translators in which the source and target architectures are the same. Again, the papers on this topic are numerous; see [3] for a more thorough treatment and citations.

1.3 Previous Work

In [14], Hansen laid much of the foundational work for just-in-time optimization. He focused on a dynamic compiler for FORTRAN, and was able to produce significantly higher quality code through progressive optimization of frequently executed pieces of code. Notably, the machine-independent optimizations that he found worthwhile were constant folding, common subexpression elimination, and loop-invariant code motion. Of these, common subexpression elimination is similar to the global value numbering process present in Section 4.2.

Cierniak and and Li [7] studied time efficient optimizations for Java, with an emphasis on high-performance programs. They formulated loop transformations in terms of loop-defined variables rather than loop induction variables, which can be computed more efficiently with similar quality of optimization. These loop-based optimizations are largely orthogonal to the optimizations presented in this paper.

Finally, Suganuma, et al., in [21], also explored optimizations for the just-in-time compilation of high-performance Java programs. Their work is more similar to Hansen's in that they focus on exploiting knowledge only available at run-time, such as method execution counts. They propose efficient inlining and code specialization optimizations that use such run-time information to produce efficient code.

1.4 Implementation: LLVM

Our implementation of these algorithms was done in the Low Level Virtual Machine (LLVM), available from `www.llvm.org` under the University of Illinois Open Source License. It was originally developed as a research compiler at the University of Illinois at Urbana-Champagne [17], but has since gained acceptance as a production grade compiler in industry. NASA uses it for code-analysis projects while Apple uses it as a JIT compiler for OpenGL shaders. It is, by design, a modular and flexible set of components rather than a monolithic compiler.

LLVM presents a pluggable infrastructure for applying analyses and optimizations to programs in an SSA-based target-independent intermediate representation, which is then handed off to the backend for machine code generation, either statically or in a JIT engine. This intermediate representation exists in three forms: as in-memory data structures, as an on-disk bitcode format, and as a human-readable textual format. Examples in this paper will be presented in the human-readable textual form, an example of which appears in Figure 1.2.

The LLVM intermediate representation is best described as an abstract machine language. It includes instructions familiar to anyone who knows a RISC-like load-store assembly language, with the addition of the ϕ instruction necessary for SSA form. These instructions operate on an infinite number of fixed-but-arbitrary width virtual registers in SSA form; the language also includes loads, stores, and a special instruction called `getelementptr` for performing pointer arithmetic in a target-independent manner.

Analyses and optimizations in LLVM are represented as "passes" (in the cases presented here, "function passes," because they act on the program one function at a time), and are capable of expressing dependency information to the infrastructure. For instance, our dead store elimination optimization announces to the infrastructure that it depends on alias analysis, so the infrastructure

```
int main(int argc, char **argv) {
  int i = 0;
  int j = 0;
  for (i = 0; i < argc; ++i) {
    j += i;
  }

  return j;
}
```

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  br label %loopheader

loopheader:
  %i.0 = phi i32 [ 0, %entry ], [ %j.1, %loopbody ]
  %j.0 = phi i32 [ 0, %entry ], [ %i.1, %loopbody ]
  %comparison = icmp slt i32 %i.0, %argc
  br i1 %comparison, label %loopbody, label %return

loopbody:
  %j.1 = add i32 %j.0, %i.0
  %i.1 = add i32 %i.0, 1
  br label %loopheader

return:
  ret i32 %j.0
}
```

Figure 1.2: A simple example in C and LLVM IR

ensures that alias analysis has been computed and made available at the time that dead store elimination is run.

The LLVM mid-level pass infrastructure includes many classical optimizations, including (but not limited to) stack to register lowering, scalar replacement of aggregates, dead argument elimination, and various loop optimizations, as well as the appropriate supporting analyses. Prior to the work implemented as part of this research, it included classical dead store elimination and global common subexpression passes. These have since been dropped in favor of the solutions developed in this paper.

1.4.1 Data Collection

All data collected for this paper was gather on an Apple Mac Pro with two dual-core 2.66Ghz Xeon processors and 4GB of RAM, running Mac OS X 10.5.1. The LLVM source was taken from the version 2.2 release branch. Optimizations that are no longer present in the LLVM source were run using the last extant version before their removal, updated only for API changes since their removal.

Tests were executed using the LLVM nightly testing framework, including the SPEC2000 and SPEC2006 tests of both integer and floating point performance as well as other test programs judged useful indicators of compiler performance by the LLVM community. All non-critical and periodic processes were shutdown before running these tests.

Chapter 2

Alias Analysis

Alias analysis, sometimes subdivided into more specific analyses like points-to analysis and mod-ref analysis, is the process of reasoning, at compile time, about which pointers in a program may or may not take on the same (or overlapping) values dynamically. The canonical question to be answered is “Can pointers a and b point to the same or overlapping memory locations?”, though related questions like “Does this function read from/write to a given memory location?” are also within its scope.

In most high level languages, alias analysis is the key enabling ingredient for memory access optimization. Without it, it is essentially impossible to make memory access transformations that are guaranteed to preserve the behavior of the program. Unfortunately, powerful alias analyses are computationally expensive, and there exists a point of diminishing returns after which increased precision of the analysis yields little benefit for optimization [15].

2.1 Classical Alias Analysis

The range of alias analyses developed in the literature is very broad. Rather than try to present any particular algorithms, it is perhaps best to convey a sense of the algorithms developed by describing the four primary axes that form a basis for the space of alias analyses: context-sensitivity, flow-sensitivity, field-sensitivity, and on-demand nature [15]. Most alias analysis algorithms can be expressed simply as a combination of these attributes. [16] provides a nice overview of the major algorithms in use today, as well as some empirical data on the benefits and costs of each.

A context-sensitive analysis is one that considers the context in which a function is called when computing aliasing information for that function. As such, each different context in which it is called effectively creates a new “copy” of the function with different aliasing characteristics due to information inherited from the caller. While this results in a more accurate analysis, it is easy to see how this can result in a combinatorial explosion in the number of computations to be performed.

Flow-sensitivity is the concept of taking into account intraprocedural control flow. Analyses that are flow-sensitive may reason about the order of definitions within a function, for instance, or may use the predicates of if-statements to improve the analysis within the conditional blocks.

Obviously these analyses tend to be more accurate than those without flow-sensitivity, and the cost of computing this additional information is generally not as high as it is for context-sensitivity.

The last form of sensitivity, field-sensitivity, is the need to model memory structures in detail. A field-sensitive analysis keeps track of every field of an object or an array as an independent memory location, while a field-insensitive analysis considers each array or object to be a single memory location (and thus pointers to any fields of the structure alias each other). A field-sensitive analysis is generally very important for object-oriented languages, as well as for enabling loop-oriented optimizations where iteration over the fields of an array is common.

The last major axis of an alias analysis is whether it does its computation up-front or on-demand. Most precise analyses perform their analysis up-front: they make a single pass over the program computing alias information, which they then store until asked for them. On-demand algorithms, in contrast, only perform analysis as needed. If no queries are ever made about a given function, aliasing information is never calculated for it. While on-demand algorithms are very nice for just-in-time systems, most precise analyses require elaborate "solver" mechanisms that are not feasible to adapt to this method of operation.

2.2 Basic Alias Analysis

LLVM includes two alias analyses: Andersen's analysis, a flow- and context-insensitive up-front analysis based on a constraint solver described in [2], and *basic alias analysis*, a minimal on-demand analysis that makes use of trivially computed local information to answer the most important alias queries quickly.

Basic alias analysis achieves this end by being aware of a few rudimentary facts. For example, it knows that pointers to locally allocated structures cannot alias pointers passed in as parameters, and that separate stack allocations cannot alias each other. Similarly, it is also able to prove that derived pointers from the same base pointer cannot alias if the field indices are provably different.

While basic alias analysis is not the focus of this paper, it is worth noting that it brings one important benefit for just-in-time compilation: its on-demand design makes it scale with the number of aliasing queries issued, rather than with the size of the program analyzed. We will exploit this fact throughout this paper to reduce compilation time by issuing fewer alias queries.

2.3 Memory Dependence Analysis

Perhaps the most significant novel contribution of the paper, and certainly the one that underlies the performance gains in the other optimizations, is memory dependence analysis. Simply put, memory dependence analysis is an aggressive caching layer on top of alias analysis that answers the question: "Given a load or a store, what preceding load or store does the state of the referenced memory location depend on?" This significantly amortizes the cost of optimizations that are typically intensive in alias queries, including dead store elimination and redundant load elimination. As an example, consider Figure 2.1. In this example, the second store depends on the load, which itself depends on the first store.

Of course, the idea of a caching layer is not novel in and of itself. What is distinctive about memory dependence analysis and what makes it more aggressive than a trivial caching layer, is its intelligent cache invalidation policy. In the context of optimizations like dead store elimination and redundant load elimination, the cache is invalidated by the removal of instructions. In a trivial caching scheme, any computational effort expended on an instruction becomes wasted if its dependee is removed. The key insight behind memory dependence analysis is that this is not necessarily the case.

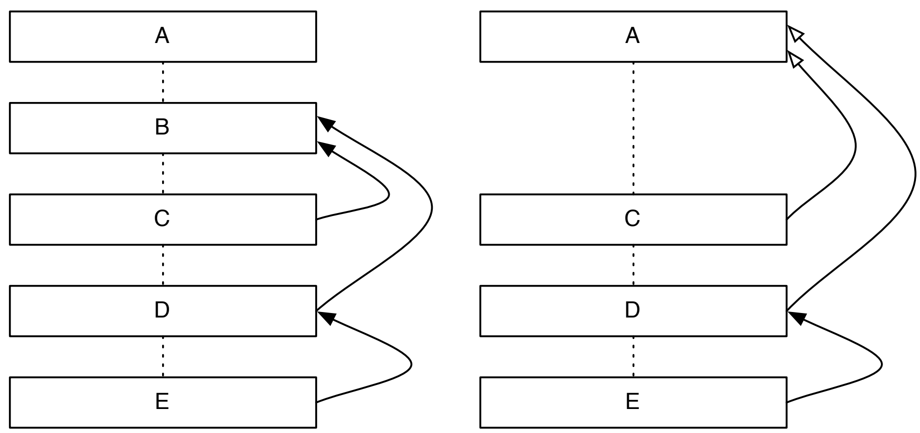
Imagine a situation as shown in Figure 2.2(a). The filled-in arrowheads represent confirmed dependencies between instructions. To determine the dependency for an instruction for which no information is known, the analysis walks backwards from that instruction, inspecting each preceding instruction to see if it has the appropriate opcode and if its operand could alias the operand of the instruction in question. Once it has found the earliest dependency, it inserts the appropriate edge (or a sentinel if the beginning of the basic block was reached) in the memory dependence graph.

Now, consider what happens when an instruction is deleted, perhaps because it was a dead store or a redundant load. Such a case is illustrated in Figure 2.2(b). When instruction B is removed from the memory dependence graph, a trivial caching system would discard all edges that previous pointed to instruction B, losing a significant amount of information. Note that this is information that was computed for queries on instruction other than B, so it is possible (perhaps likely, depending on the client) that this information will be required again in the future.

Rather than discard these edges and lose information, memory dependence analysis moves the target of these edges to the predecessor of B, and marks them as unconfirmed edges (denoted by a white arrowhead). In this way, the analysis preserves the information that instructions that previously depended on instruction B still do not depend on any instruction later than B's predecessor. The next time that a query is made for instruction D, for instance, the analysis will scan up the basic block looking for dependencies, *starting with the unconfirmed dependency*. This behavior allows for radically fewer queries to be forwarded to the underlying alias analysis by avoiding recomputing information.

```
define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca i32
  store i32 0, i32* %a
    ...
  %b = load i32* %a
    ...
  store i32 1, i32* %a
  ret i32 %b
}
```

Figure 2.1: An example of memory dependence analysis



(a) A memory dependence graph (b) The same graph after removing instruction B

Figure 2.2: A diagramatic example of the internals of memory dependence analysis

2.4 Results

It is challenging to measure the effects of memory dependence analysis on compilation time because the algorithms designed to make use of it are inherently different than those that work directly with alias queries. The limited range of questions that MDA can answer forces this dichotomy. In addition, the algorithms developed to work with it were themselves designed to reduce the number of alias queries performed. As such, numerical differences between operation with and without memory dependence analysis must be looked at from a high level, rather than as precise measures.

Table 2.1 presents the number of alias queries that were evaluated during the process of executing dead store elimination on the C-based tests in the SPEC suite, a popular set of benchmarks produced by the Standard Performance Evaluation Corporation. While in some data points (`401.bzip2` and `473.astar`) the difference is most likely due to more intelligent choices made by the new dead store elimination algorithm, in most cases the superior scalability of memory dependence analysis is evident.

Measurements of compilation time for this and the other algorithms in this paper are presented together in Section 5.1.

SPEC2006	Old DSE	New DSE
`400.perlbench`	27514	260
`401.bzip2`	3584	0
`403.gcc`	53009	81
`429.mcf`	699	0
`433.milc`	4068	75
`444.namd`	13545	10
`445.gobmk`	21747	59
`447.dealII`	1296659	99
`456.hmmr`	11574	12
`458.sjeng`	11941	118
`462.libquantum`	9474	212
`464.h264ref`	21140	150
`470.lbm`	815	3
`471.omnetpp`	13379	4
`473.astar`	2342	0

SPEC2000	Old DSE	New DSE
`164.gzip`	1335	35
`175.vpr`	5868	7
`176.gcc`	26443	66
`177.mesa`	116920	1
`179.art`	449	0
`181.mcf`	701	0
`183.equake`	1289	3
`186.crafty`	11229	12
`188.ammp`	4713	9
`197.parser`	3085	6
`252.eon`	463745	43
`253.perlbmk`	17070	188
`254.gap`	36503	21
`255.vortex`	62684	819
`256.bzip2`	467	7
`300.twolf`	10540	16

Table 2.1: The number of alias queries evaluated on the SPEC benchmarks

Chapter 3

Dead Store Elimination

Dead store elimination is, on the surface, a simple optimization: if a memory address is stored to twice without an intervening load, the earlier store is dead and can be eliminated. Complications arise, however, when one must take into account the possibility of stores of different sizes, overlapping but not containing stores, and, of course, the possibility of imprecise alias analysis: it is not always possible to tell if an intervening load accesses the memory address in question, or even if the second store entirely overwrites the first one.

For example, in Figure 3.1, we cannot remove the first `store` because the function `@foo` might read the memory location pointed to by `%ptr`. If additional analysis were to discover that `@foo` does not read from the memory location at `%ptr`, then the `store` may indeed be safe to remove, as in Figure 3.2.

This optimization is a subset of the broad class of optimizations called "dead code elimination." More general algorithms exist which try to eliminate all forms of dead code using unreachability information, alias analysis, etc. Such techniques are described in many papers, such as [11], which is also a fundamental paper on Static Single Assignment form. Dead store elimination is distinguished from these by virtue of being focused solely on the removal of stores through the use of alias analysis information. This makes it more efficient to compute than a more inclusive dead code elimination pass.

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %ptr = alloca i32
  store i32 3, i32* %ptr
  call @foo(%ptr)
  store i32 4, i32* %ptr
  %value = load i32* %ptr
  ret i32 %value
}
```

Figure 3.1: An example where the first `store` is potentially necessary

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %ptr = alloca i32
  store i32 3, i32* %ptr
  store i32 4, i32* %ptr
  %value = load i32* %ptr
  ret i32 %value
}
```

Figure 3.2: An example in which the first `store` is dead

It is worth noting that, in most systems, the performance of stores is not critical to the overall performance of a program, due largely to now-common write-back cache policies. The elimination of stores remains a critical performance optimization, however, because it exposes further opportunities to eliminate loads, as will be discussed in Section 4.

While the removal of unnecessary stores is only an indirect performance win for traditional computers, it is a very direct win for mobile and embedded devices where power consumption and heat are key metrics. The registers in which the value is already stored are already consuming power, so there is no cost to keeping the value there. Storing to memory, however, requires, at the very least, supplying power to the cache and possibly to primary memory as well. In such devices, any optimization that reduces the need for power to be supplied to other components is a definite win.

3.1 Classical Dead Store Elimination

The classical form of dead store elimination was implemented in LLVM using a unification-based set data structure, called an `AliasSet`. When a pointer value is inserted into this set, it is only stored if it is not must-alias with a value already in the set. However, when a pointer is erased from the set, any pointers in the set that may-alias it are also removed.

The general operation of the dead store elimination optimization was very simple: While performing a reverse walk of each basic block of the function, whenever a store is encountered, its target is added to the `AliasSet`. When a load is encountered, its source is erased from the set. If a store is encountered whose target was already in the set, then that store can be safely removed. Algorithm 1 presents pseudocode for this operation.

While conceptually simple, this implementation hid significant implementation details as well as poor average time complexity in the `AliasSet`. Every time a pointer is inserted or removed from the set, the unification operations impose a complexity that proved, empirically, quadratic for common cases, as well as issuing potentially huge numbers of alias queries. Because of this, several cases were found in the SPEC benchmarks where functions with a particularly large number of loads and stores took minutes to optimize, even on fast machines.

Algorithm 1 Classical Dead Store Elimination

```
Require: a function F
  for each basic block B in F do
    AS ← empty AliasSet
    for each instruction I in B in reverse order do
      if I is a store then
        P ← target(I)
        if P in AS then
          erase I
        end if
      else if I is a load then
        remove(AS, source(I))
      end if
    end for
  end for
```

3.2 Fast Dead Store Elimination

Fast dead store elimination, as our new algorithm is called, does away completely with the unification mechanism around which the old algorithm was based. Instead, memory dependence analysis, with its aggressive caching mechanism, is used to provide similar information with much improved average complexity.

The new algorithm (shown in Algorithm 2) begins by walking each block forwards, rather than backwards as in the classical implementation. As it performs this walk, it records a mapping between pointers and the last-seen stores to those pointers. If a store to a pointer is found to which there is already a store in the last-seen mapping, then the store in the mapping is a candidate for deletion.

Algorithm 2 Fast Dead Store Elimination

```
Require: a function F
  for each basic block B in F do
    lastStore ← empty map
    for each instruction I in B in forward order do
      if I is a store then
        P ← target(I)
        if P in lastStore and lastStore[P] = getDependency(I) then
          erase lastStore[P]
        end if
        lastStore[P] ← I
      end if
    end for
  end for
```

SPEC2006	Old DSE	New DSE
400.perlbench	109	531
401.bzip2	3	6
403.gcc	173	184
429.mcf	0	0
433.milc	3	3
444.namd	31	31
445.gobmk	9	9
447.dealII	2770	3048
456.hmmr	5	5
458.sjeng	34	34
462.libquantum	3	5
464.h264ref	64	66
470.lbm	6	6
471.omnetpp	98	98
473.astar	30	30

SPEC2000	Old DSE	New DSE
164.gzip	6	6
175.vpr	2	2
176.gcc	141	147
177.mesa	0	0
179.art	0	0
181.mcf	0	0
183.equake	4	4
186.crafty	22	22
188.ammp	7	7
197.parser	1	1
252.eon	2549	2846
253.perlbmk	10	288
254.gap	13	13
255.vortex	174	174
256.bzip2	2	2
300.twolf	20	20

Table 3.1: The number of stores removed on the SPEC benchmarks

It is at this point that memory dependence analysis enters the picture: in order to ensure the safety of a deletion, fast dead store elimination simply asks MDA if the current store's immediate dependency is the instruction in the last-seen map. If it is, then there cannot have been any intervening loads, and the deletion is safe to perform. It is worth noting that this process will miss dead stores in the face of must-aliased pointers. However, because of the imprecision of alias analyses that are practical for just-in-time compilation, this does not make a difference in practice, as we will see in Section 3.3.

Note that this query to memory dependence analysis is of linear complexity, possibly lower due to caching. This is far better for compile time than the complex unification operations that an `AliasSet` would have to perform for every load. Additionally, the number of queries issued by fast dead store elimination is proportional to the number of potential deletions present, whereas classical dead store elimination had to perform the unification step for every load encountered, regardless of whether or not it was useful.

3.3 Results

The old and new implementations of dead store elimination must be compared on two criteria: effectiveness in eliminating stores, and speed of optimization. For a just-in-time compiler, speed of compilation is the first priority, but we also do not want to sacrifice generated code quality if we can help it. Fortunately, the new dead store elimination algorithm performs exceptionally well in practice!

As can be seen in Table 3.1, new DSE eliminated as many or more stores as the classical version on every testcase. In fact, on some tests, such as `400.perlbench`, it eliminated drastically more.

This is largely due to two factors: first, the `AliasSet` structure in the classical version forced conservative assumptions that caused it to miss some dead stores. Secondly, the weakness of new DSE, must-aliased pointers, are rare without precise (and therefore costly) alias analysis. Because this experiment is within the context of a just-in-time compilation system, a costly alias analysis is infeasible.

Measurements of compilation time for this and the other algorithms in this paper are presented together in Section 5.1.

Chapter 4

Redundant Load Elimination

The complementary optimization to dead store elimination, redundant load elimination is the process of eliminating the later of two loads if there are no intervening stores. This process is itself muddled by most of the same issues that afflict dead store elimination, in that, in the face of imprecise alias analysis, it is not always possible to tell if two loads access the same memory location, or whether an intervening store might touch that location.

Because of the possibility for intervening stores complicates the process of redundant load elimination, it is usually desirable to have run dead store elimination immediately beforehand, in the hopes of exposing more opportunities for redundant load elimination. A more aggressive approach would be to repeatedly execute both optimizations until the program converged to a fixed point: redundant load elimination could expose more opportunities for dead store elimination which could expose more redundant loads, etc.

Figures 4.1 and 4.2 illustrate examples of non-redundant and redundant loads respectively. In the former, the block containing the load has two predecessors: `%entry` and `%true_branch`. The load has a dependency with the correct pointer value in `%true_branch`, but no dependency at all in `%entry`. Thus the `load` is not redundant. In Figure 4.2, on the other hand, the `load` in `%return` has dependencies in both of its predecessors: the `store` in `%true_branch` and the `load` in `%false_branch`. Figure 4.3 shows the result after eliminating the redundant `load`.

4.1 Global Common Subexpression Elimination

The existing form of redundant load elimination in LLVM was, incorrectly, called Global Common Subexpression Elimination. GCSE correctly refers to the work of Cocke in [9], which is a form of redundant instruction removal based on using Gaussian elimination to solve dependency constraints. This older form of redundant instruction removal is no longer commonly used in practice. Instead, the optimization known as "GCSE" in LLVM is in fact a form of global value numbering.

Global value numbering, possibly best described in [5] and [8], is a newer technique that takes advantage of static single assignment form. It is conceptually quite simple: in SSA form, every variable is defined only once. Thus, for each instruction, we can compute a unique "expres-

```
define i32 @nonredundant(i32* %ptr, i1 %cond) {
entry:
 br i1 %cond, label %true_branch, label %return

true_branch:
 store i32 0, i32* %ptr
 br label %return

return:
 %a = load i32* %ptr
 ret i32 %a
}
```

Figure 4.1: An example where the `load` is not redundant

```
define i32 @redundant(i32* %ptr, i1 %cond) {
entry:
 br i1 %cond, label %true_branch, label %false_branch

true_branch:
 store i32 0, i32* %ptr
 br label %return

false_branch:
 %b = load i32* %ptr
 br label %return

return:
 %a = load i32* %ptr
 ret i32 %a
}
```

Figure 4.2: An example where the `load` is redundant

```
define i32 @redundant(i32* %ptr, i1 %cond) {
entry:
 br i1 %cond, label %true_branch, label %false_branch

true_branch:
 store i32 0, i32* %ptr
 br label %return

false_branch:
 %b = load i32* %ptr
 br label %return

return:
 %a.rle = phi i32 [ %b, %false_branch ], [ 0, %true_branch ]
 ret i32 %a
}
```

Figure 4.3: The result of eliminating a redundant `load`

sion" for its value. These expressions contain the opcode of the defining instruction as well as the value numbers of the instruction's operands. These expressions are then used as indexes into a hashtable, mapping to value numbers. Instructions with no operands (such as function arguments, or function calls) are each given a unique value number, as is each expression the first time it is encounter. Subsequent expressions with the same opcode and operand value numbers receive the same value number.

The process of removing redundant instructions is then quite simple: an instruction is redundant if another instruction with the same value number is already available at that program point. The simplest way to compute availability is for each basic block to inherit the set of available values from its immediate dominator, and add its own additions before passing them on to the blocks that it dominates. More aggressive means of propagating availability information have been explored, for example in [10].

Applying this technique to memory operations, however, is difficult. Because loads and stores affect program state beyond what is represented by the virtual registers of SSA form, the single-definition assumption does not hold for them. Because optimizing these instructions is critical for program performance, most global value numbering techniques integrate some method of handling them.

LLVM's existing redundant load elimination functionality was achieved in a way that was transparent to the redundant instruction removal system. It implements an additional analysis called load value numbering, which assigned value numbers from the same value numbering pool as global value numbering. The analysis' key task was to assign, in the face of arbitrary control flow, value numbers to load instructions such that two loads that were separated by a store never receive the same number. With that guarantee, the instruction removal process was able to operate on loads just as it does on register-register arithmetic.

The problem with this approach is that it scales very poorly. The analysis is forced to consider essentially every path between every pair of loads to determine if they should receive the same value number. This quite obviously leads to a very high order of growth. Relatively small testcases could produce unacceptably long analysis times.

4.2 Global Value Numbering

Our new implementation, in addition to correcting the aforementioned nomenclature issue by being named "GVN," is intended to correct this issue of scalability. The redundant instruction removal functionality is based on the description of GVN given in [23], though other versions such as [10] could be used instead. The key difference is in how we handle load instructions.

Rather than trying to retrofit the analysis of loads into the value numbering paradigm, we instead use the facilities provided by memory dependence analysis to simplify the problem while simultaneously providing aggressive caching. The reasoning is simple: we perform a query for the dependencies (including non-local ones) of each load that we consider. If all of these dependencies are loads or stores to the same pointer as the original load, then that load is redundant. We then perform normal ϕ construction to make the results of the preceding loads and the store value of the preceding stores available at the current instruction. Finally, we replace all uses of the current load with the result of ϕ construction and delete the original load.

Note that this approach contains an inherent inaccuracy: it does not attempt to handle cases in which the dependee load is from a must-aliased pointer. As we shall see in Section 4.3, this does not occur often enough in practice to have a significant impact of the effectiveness of the optimization.

The key advantage of this approach over the classical method is that the amount of work performed to eliminate redundant loads scales, on average, with the number of loads that we consider to be candidates for removal. This is in contrast to the classical approach in which the entire set of loads in a function must be partitioned into equivalence classes, leading to a scaling factor of $O\left(n^2\right)$.

4.3 Results

In Table 4.1 we present the number of redundant loads removed by both GVN and GCSE when run on the SPEC 2000 and 2006 testsuites. The most immediate observation is that GCSE outperforms GVN in only a single case (`176.gcc`), while on all the others GVN was at least as, if not more effective than GCSE. This is in spite of the inherent inaccuracy with respect to must-aliased pointers GVN suffers, clearly illustrating that they are not a common enough occurrence given the reality of imprecise alias analysis to be of significance for optimization.

It is also worth noting that while GCSE outperforms GVN on `176.gcc`, the updated version of the same testcase, `403.gcc`, is better optimized by GVN. On manual inspection, it appears that most of the redundant stores that are missed on `176.gcc` are in error handling routines where

SPEC2006	GCSE	GVN
`400.perlbench`	15137	15408
`401.bzip2`	641	964
`403.gcc`	33948	34045
`429.mcf`	85	86
`433.milc`	783	832
`444.namd`	735	773
`445.gobmk`	4057	4311
`447.dealII`	0	16828
`456.hmmr`	2950	2988
`458.sjeng`	1285	1322
`462.libquantum`	175	177
`464.h264ref`	8633	8836
`470.lbm`	257	260
`471.omnetpp`	1856	2024
`473.astar`	462	482

SPEC2000	GCSE	GVN
`164.gzip`	293	339
`175.vpr`	958	1019
`176.gcc`	17479	17015
`177.mesa`	3668	3720
`179.art`	199	210
`181.mcf`	81	82
`183.equake`	541	557
`186.crafty`	1751	2016
`188.ammp`	1068	1078
`197.parser`	439	472
`252.eon`	4841	4925
`253.perlbmk`	7519	7529
`254.gap`	6948	6968
`255.vortex`	2999	3477
`256.bzip2`	174	193
`300.twolf`	4224	4559

Table 4.1: The number of loads removed on the SPEC benchmarks

pointers to global error strings are manipulated. Because GVN does not comprehend must-alias relationships, it fails to realize that these loads are redundant.

Measurements of compilation time for this and the other algorithms presented in this paper are presented together in Section 5.1.

Chapter 5

Conclusions

While we have presented numbers of loads and stores removed, as well as numbers of alias queries issued, we have not yet addressed how these numbers translate into real world performance, both in code quality and compile time. In this final section, we will present empirical evidence that the algorithms presented in the earlier sections result in measurable improvements in compile time without significant changes in code quality.

In addition, we present results for all four possible combinations of old and new optimizations. Because of the nature of loads and stores, the interactions between a given pair of optimizations may be significant: removing more loads may expose more dead stores, and vice versa. Indeed, possible interactions are not limited to just between these two optimizations; a change in the output of redundant load elimination could potentially cause changes in the performance of other optimizations, or in the effectiveness of code generation. For example, a strong value numbering implementation tends to keep values in registers longer, which, while better in theory, puts more pressure on the register allocator to produce performant object code. In the end, the only way to obtain a representative picture of optimizations in practice is to measure their effectiveness as part of a realistic compilation process, rather than measuring each in isolation.

5.1 Results

Figure 5.1 shows the total optimization time of the four largest testcases from SPEC2000 and SPEC2006 with the four possible combinations of GVN, GCSE, new DSE, and old DSE. This measurement represents the time to execute all optimizations, not just ours. We present these four in part because, for the smaller testcases, the optimization time was too small to be accurately measured with the available instrumentation.

In all four cases, the total optimization time decreased, particularly drastically in `447.dealII` and `403.gcc`. In `447.dealII` in particular, the total optimization time decreased by almost half. We have observed that, in general, the decrease in optimization time is greater for larger testcases, suggesting an improvement in the order of growth of the algorithms as opposed to a decrease by a constant factor.

In Figure 5.2, we present the normalized execution time of those four testcases when optimized

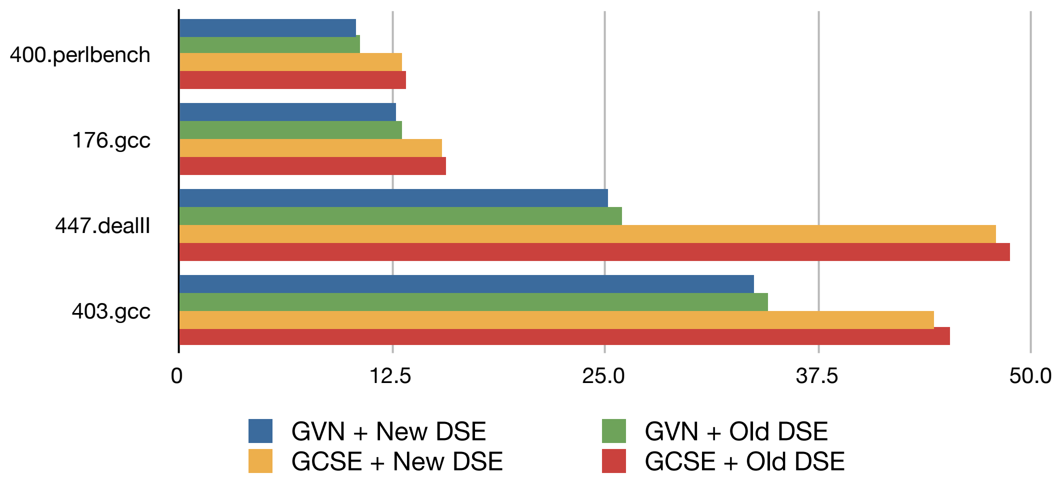


Figure 5.1: Total time to execute the optimizations in seconds of the four largest testcases from SPEC

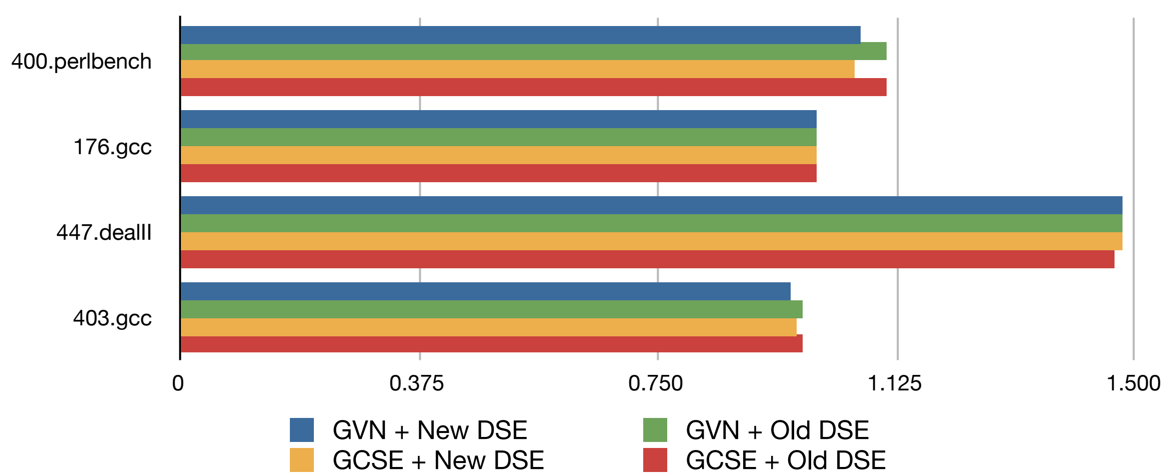


Figure 5.2: Normalized execution time of the four largest testcases from SPEC

with each of the four combinations. In this situation, the normalized execution time is the ratio of the execution time of the testcase when compiled with the system compiler (`gcc`) to the execution time when compiled with LLVM (with our optimizations). Because of this, longer bars are better. This normalization allows us to present the bars for all for testcases, which have significantly different absolute execution times, on the same scale.

The normalized results indicate that the four possible combinations are of approximately equal effectiveness. On three of the testcases (`176.gcc`, `447.dealII`, and `403.gcc`) all four results differ only within the range of noise. On `400.perlbench`, old DSE appears to have outperformed new DSE slightly in the effectiveness of its optimization.

These results are repeated throughout the entirety of the SPEC testsuites. On most testcases, the differences between all four techniques fall within the range of noise. One other notable case in which the older approaches perform better than the new ones is `179.art`, in which the normalized execution time went from 1.82 with GCSE and old DSE to 1.78 with GVN and new DSE. After investigation, it was discovered that this was not due to loads or stores that were not removed, but rather because of GVN's increased aggression: its more aggressive elimination of loads introduced ϕ functions that were not present in the output form GCSE. The `scalarrepl` pass, which breaks up aggregates into scalars when profitable, was confused by these ϕ functions, causing it to miss several opportunities for optimization.

5.2 Future Work

This work has presented one new analysis and two optimizations based on it that are designed with dynamic compilation in mind. While memory access optimization is one of the most profitable forms of optimization, there are many other classical optimizations that could possibly be adapted to a just-in-time context, including code placement, constant propagation, and loop transformations.

Bibliography

[1] P. S. ABRAMS, *An APL machine*, PhD thesis, Stanford University, 1970.

[2] L. ANDERSEN, *Program analysis and specialization for the C programming language*, PhD thesis, University of Copenhagen, 1994.

[3] J. AYCOCK, *A Brief History of Just-in-Time*, ACM Comput. Surv., 35 (2003), pp. 97–113.

[4] P. BRIGGS, K. D. COOPER, T. J. HARVEY, AND L. T. SIMPSON, *Practical Improvements to the Construction and Destruction of Static Single Assignment Form*, Software—Practice and Experience, 28 (1998), pp. 859–881.

[5] P. BRIGGS, K. D. COOPER, AND L. T. SIMPSON, *Value Numbering*, Software—Practice and Experience, 27 (1997), pp. 701–724.

[6] Z. BUDIMLIC, K. D. COOPER, T. J. HARVEY, K. KENNEDY, T. S. OBERG, AND S. W. REEVES, *Fast copy coalescing and live-range identification*, in PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, 2002, ACM, pp. 25–32.

[7] M. CIERNIAK AND W. LI, *Just-in-time optimizations for high-performance Java programs*, Concurrency: Practice and Experience, 9 (1997), pp. 1063–1073.

[8] C. CLICK, *Global code motion/global value numbering*, SIGPLAN Not., 30 (1995), pp. 246–257.

[9] J. COCKE, *Global common subexpression elimination*, in Proceedings of a symposium on Compiler optimization, 1970, pp. 20–24.

[10] K. COOPER AND T. SIMPSON, *SCC-based value numbering*, tech. report, Rice University, 1995.

[11] R. CYTRON, J. FERRANTE, B. K. ROSEN, M. N. WEGMAN, AND F. K. ZADECK, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst., 13 (1991), pp. 451–490.

[12] R. J. DAKIN AND P. C. POOLE, *A mixed code approach*, The Computer Journal, 16 (1973), pp. 219–222.

[13] J. L. DAWSON, *Combining interpretive code with machine code*, The Computer Journal, 16 (1973), pp. 216–219.

[14] G. J. HANSEN, *Adaptive systems for the dynamic run-time optimization of programs.*, PhD thesis, Carnegie Mellon University, 1974.

[15] M. HIND, *Pointer Analysis: Haven't We Solved This Problem Yet?*, in 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), Snowbird, UT, 2001.

[16] M. HIND AND A. PIOLI, *Which pointer analysis should I use?*, in International Symposium on Software Testing and Analysis, 2000, pp. 113–123.

[17] C. LATTNER AND V. ADVE, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[18] J. L. MCCARTHY, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, Communications of the ACM, 3 (1960), pp. 184–195.

[19] J. G. MITCHELL, *The design and construction of flexible and efficient interactive programming systems*, PhD thesis, Carnegie Mellon University, 1970.

[20] N. NETHERCOTE AND J. SEWARD, *Valgrind: a framework for heavyweight dynamic binary instrumentation*, SIGPLAN Not., 42 (2007), pp. 89–100.

[21] T. SUGANUMA, T. YASUE, M. KAWAHITO, H. KOMATSU, AND T. NAKATANI, *Design and evaluation of dynamic optimizations for a Java just-in-time compiler*, ACM Trans. Program. Lang. Syst., 27 (2005), pp. 732–785.

[22] D. UNGAR, R. B. SMITH, C. CHAMBERS, AND U. HÖLZLE, *Object, Message, and Performance: How they Coexist in Self*, Computer, 25 (1992), pp. 53–64.

[23] T. VANDRUNEN, *Partial Redundancy Elimination for Global Value Numbering*, PhD thesis, Purdue University, 2004.

Code Listing

```
//===- llvm/Analysis/MemoryDependenceAnalysis.h - Memory Deps --*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines an analysis that determines, for a given memory operation,
// what preceding memory operations it depends on. It builds on alias analysis
// information, and tries to provide a lazy, caching interface to a common kind
// of alias information query.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_ANALYSIS_MEMORY_DEPENDENCE_H
#define LLVM_ANALYSIS_MEMORY_DEPENDENCE_H

#include "llvm/Pass.h"
#include "llvm/Support/CallSite.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/Support/Compiler.h"

namespace llvm {

class Function;
class FunctionPass;
class Instruction;

class MemoryDependenceAnalysis : public FunctionPass {
  private:
    // A map from instructions to their dependency, with a boolean
    // flags for whether this mapping is confirmed or not
    typedef DenseMap<Instruction*, std::pair<Instruction*, bool> >
            depMapType;
    depMapType depGraphLocal;

    // A map from instructions to their non-local dependencies.
```

```
  typedef DenseMap<Instruction*, DenseMap<BasicBlock*, Value*> >
          nonLocalDepMapType;
  nonLocalDepMapType depGraphNonLocal;

  // A reverse mapping form dependencies to the dependees. This is
  // used when removing instructions to keep the cache coherent.
  typedef DenseMap<Value*, SmallPtrSet<Instruction*, 4> >
          reverseDepMapType;
  reverseDepMapType reverseDep;

  // A reverse mapping form dependencies to the non-local dependees.
  reverseDepMapType reverseDepNonLocal;

public:
  void ping(Instruction* D);

  // Special marker indicating that the query has no dependency
  // in the specified block.
  static Instruction* const NonLocal;

  // Special marker indicating that the query has no dependency at all
  static Instruction* const None;


  // Special marker indicating a dirty cache entry
  static Instruction* const Dirty;

  static char ID; // Class identification, replacement for typeinfo
  MemoryDependenceAnalysis() : FunctionPass((intptr_t)&ID) {}

  /// Pass Implementation stuff.  This doesn't do any analysis.
  ///
  bool runOnFunction(Function &) {return false; }

  /// Clean up memory in between runs
  void releaseMemory() {
    depGraphLocal.clear();
    depGraphNonLocal.clear();
    reverseDep.clear();
    reverseDepNonLocal.clear();
  }

  /// getAnalysisUsage - Does not modify anything. It uses Value Numbering
  /// and Alias Analysis.
  ///
  virtual void getAnalysisUsage(AnalysisUsage &AU) const;

  /// getDependency - Return the instruction on which a memory operation
  /// depends, starting with start.
  Instruction* getDependency(Instruction* query, Instruction* start = 0,
                             BasicBlock* block = 0);

  /// getNonLocalDependency - Fills the passed-in map with the non-local
```

```
    /// dependencies of the queries.  The map will contain NonLocal for
    /// blocks between the query and its dependencies.
    void getNonLocalDependency(Instruction* query,
                               DenseMap<BasicBlock*, Value*>& resp);

    /// removeInstruction - Remove an instruction from the dependence analysis,
    /// updating the dependence of instructions that previously depended on it.
    void removeInstruction(Instruction* rem);

    /// dropInstruction - Remove an instruction from the analysis, making
    /// absolutely conservative assumptions when updating the cache. This is
    /// useful, for example when an instruction is changed rather than removed.
    void dropInstruction(Instruction* drop);

  private:
    Instruction* getCallSiteDependency(CallSite C, Instruction* start,
                                       BasicBlock* block);
    void nonLocalHelper(Instruction* query, BasicBlock* block,
                        DenseMap<BasicBlock*, Value*>& resp);
  };

} // End llvm namespace

#endif
```

```cpp
//===- MemoryDependenceAnalysis.cpp - Mem Deps Implementation --*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file implements an analysis that determines, for a given memory
// operation, what preceding memory operations it depends on. It builds on
// alias analysis information, and tries to provide a lazy, caching interface to
// a common kind of alias information query.
//
//===----------------------------------------------------------------------===//

#include "llvm/Analysis/MemoryDependenceAnalysis.h"
#include "llvm/Constants.h"
#include "llvm/Instructions.h"
#include "llvm/Function.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetData.h"
#include "llvm/ADT/Statistic.h"

#define DEBUG_TYPE "memdep"

using namespace llvm;

namespace {
  // Control the calculation of non-local dependencies by only examining the
  // predecessors if the basic block has less than X amount (50 by default).
  cl::opt<int>
  PredLimit("nonlocaldep-threshold", cl::Hidden, cl::init(50),
            cl::desc("Control the calculation of non-local"
                     "dependencies (default = 50)"));
}

STATISTIC(NumCacheNonlocal, "Number of cached non-local responses");
STATISTIC(NumUncacheNonlocal, "Number of uncached non-local responses");

char MemoryDependenceAnalysis::ID = 0;

Instruction* const MemoryDependenceAnalysis::NonLocal = (Instruction*)-3;
Instruction* const MemoryDependenceAnalysis::None = (Instruction*)-4;
Instruction* const MemoryDependenceAnalysis::Dirty = (Instruction*)-5;

// Register this pass...
static RegisterPass<MemoryDependenceAnalysis> X("memdep",
                                                "Memory Dependence Analysis");
```

```
void MemoryDependenceAnalysis::ping(Instruction *D) {
  for (depMapType::iterator I = depGraphLocal.begin(), E = depGraphLocal.end();
       I != E; ++I) {
    assert(I->first != D);
    assert(I->second.first != D);
  }

  for (nonLocalDepMapType::iterator I = depGraphNonLocal.begin(), E = depGraphNonLocal.end();
       I != E; ++I) {
    assert(I->first != D);
  }

  for (reverseDepMapType::iterator I = reverseDep.begin(), E = reverseDep.end();
       I != E; ++I)
    for (SmallPtrSet<Instruction*, 4>::iterator II = I->second.begin(), EE = I->second.end();
         II != EE; ++II)
      assert(*II != D);

  for (reverseDepMapType::iterator I = reverseDepNonLocal.begin(), E = reverseDepNonLocal.end();
       I != E; ++I)
    for (SmallPtrSet<Instruction*, 4>::iterator II = I->second.begin(), EE = I->second.end();
         II != EE; ++II)
      assert(*II != D);
}

/// getAnalysisUsage - Does not modify anything. It uses Alias Analysis.
///
void MemoryDependenceAnalysis::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
  AU.addRequiredTransitive<AliasAnalysis>();
  AU.addRequiredTransitive<TargetData>();
}

/// getCallSiteDependency - Private helper for finding the local dependencies
/// of a call site.
Instruction* MemoryDependenceAnalysis::getCallSiteDependency(CallSite C,
                                                             Instruction* start,
                                                             BasicBlock* block) {

  std::pair<Instruction*, bool>& cachedResult =
                                          depGraphLocal[C.getInstruction()];
  AliasAnalysis& AA = getAnalysis<AliasAnalysis>();
  TargetData& TD = getAnalysis<TargetData>();
  BasicBlock::iterator blockBegin = C.getInstruction()->getParent()->begin();
  BasicBlock::iterator QI = C.getInstruction();

  // If the starting point was specifiy, use it
  if (start) {
    QI = start;
    blockBegin = start->getParent()->end();
  // If the starting point wasn't specified, but the block was, use it
  } else if (!start && block) {
    QI = block->end();
```

```
  blockBegin = block->end();
}

// Walk backwards through the block, looking for dependencies
while (QI != blockBegin) {
  --QI;

  // If this inst is a memory op, get the pointer it accessed
  Value* pointer = 0;
  uint64_t pointerSize = 0;
  if (StoreInst* S = dyn_cast<StoreInst>(QI)) {
    pointer = S->getPointerOperand();
    pointerSize = TD.getTypeStoreSize(S->getOperand(0)->getType());
  } else if (AllocationInst* AI = dyn_cast<AllocationInst>(QI)) {
    pointer = AI;
    if (ConstantInt* C = dyn_cast<ConstantInt>(AI->getArraySize()))
      pointerSize = C->getZExtValue() * \
                    TD.getABITypeSize(AI->getAllocatedType());
    else
      pointerSize = ~0UL;
  } else if (VAArgInst* V = dyn_cast<VAArgInst>(QI)) {
    pointer = V->getOperand(0);
    pointerSize = TD.getTypeStoreSize(V->getType());
  } else if (FreeInst* F = dyn_cast<FreeInst>(QI)) {
    pointer = F->getPointerOperand();

    // FreeInsts erase the entire structure
    pointerSize = ~0UL;
  } else if (isa<CallInst>(QI)) {
    AliasAnalysis::ModRefBehavior result =
                AA.getModRefBehavior(CallSite::get(QI));
    if (result != AliasAnalysis::DoesNotAccessMemory &&
        result != AliasAnalysis::OnlyReadsMemory) {
      if (!start && !block) {
        cachedResult.first = QI;
        cachedResult.second = true;
        reverseDep[QI].insert(C.getInstruction());
      }
      return QI;
    } else {
      continue;
    }
  } else
    continue;

  if (AA.getModRefInfo(C, pointer, pointerSize) != AliasAnalysis::NoModRef) {
    if (!start && !block) {
      cachedResult.first = QI;
      cachedResult.second = true;
      reverseDep[QI].insert(C.getInstruction());
    }
    return QI;
  }
```

```
  }

  // No dependence found
  cachedResult.first = NonLocal;
  cachedResult.second = true;
  reverseDep[NonLocal].insert(C.getInstruction());
  return NonLocal;
}

/// nonLocalHelper - Private helper used to calculate non-local dependencies
/// by doing DFS on the predecessors of a block to find its dependencies
void MemoryDependenceAnalysis::nonLocalHelper(Instruction* query,
                                              BasicBlock* block,
                                         DenseMap<BasicBlock*, Value*>& resp) {
  // Set of blocks that we've already visited in our DFS
  SmallPtrSet<BasicBlock*, 4> visited;
  // If we're updating a dirtied cache entry, we don't need to reprocess
  // already computed entries.
  for (DenseMap<BasicBlock*, Value*>::iterator I = resp.begin(),
       E = resp.end(); I != E; ++I)
    if (I->second != Dirty)
      visited.insert(I->first);

  // Current stack of the DFS
  SmallVector<BasicBlock*, 4> stack;
  stack.push_back(block);

  // Do a basic DFS
  while (!stack.empty()) {
    BasicBlock* BB = stack.back();

    // If we've already visited this block, no need to revist
    if (visited.count(BB)) {
      stack.pop_back();
      continue;
    }

    // If we find a new block with a local dependency for query,
    // then we insert the new dependency and backtrack.
    if (BB != block) {
      visited.insert(BB);

      Instruction* localDep = getDependency(query, 0, BB);
      if (localDep != NonLocal) {
        resp.insert(std::make_pair(BB, localDep));
        stack.pop_back();

        continue;
      }
    // If we re-encounter the starting block, we still need to search it
    // because there might be a dependency in the starting block AFTER
    // the position of the query.  This is necessary to get loops right.
    } else if (BB == block && stack.size() > 1) {
```

```
      visited.insert(BB);

      Instruction* localDep = getDependency(query, 0, BB);
      if (localDep != query)
        resp.insert(std::make_pair(BB, localDep));

      stack.pop_back();

      continue;
    }

    // If we didn't find anything, recurse on the precessors of this block
    // Only do this for blocks with a small number of predecessors.
    bool predOnStack = false;
    bool inserted = false;
    if (std::distance(pred_begin(BB), pred_end(BB)) <= PredLimit) {
      for (pred_iterator PI = pred_begin(BB), PE = pred_end(BB);
           PI != PE; ++PI)
        if (!visited.count(*PI)) {
          stack.push_back(*PI);
          inserted = true;
        } else
          predOnStack = true;
    }

    // If we inserted a new predecessor, then we'll come back to this block
    if (inserted)
      continue;
    // If we didn't insert because we have no predecessors, then this
    // query has no dependency at all.
    else if (!inserted && !predOnStack) {
      resp.insert(std::make_pair(BB, None));
    // If we didn't insert because our predecessors are already on the stack,
    // then we might still have a dependency, but it will be discovered during
    // backtracking.
    } else if (!inserted && predOnStack){
      resp.insert(std::make_pair(BB, NonLocal));
    }

    stack.pop_back();
  }
}

/// getNonLocalDependency - Fills the passed-in map with the non-local
/// dependencies of the queries.  The map will contain NonLocal for
/// blocks between the query and its dependencies.
void MemoryDependenceAnalysis::getNonLocalDependency(Instruction* query,
                                      DenseMap<BasicBlock*, Value*>& resp) {
  if (depGraphNonLocal.count(query)) {
    DenseMap<BasicBlock*, Value*>& cached = depGraphNonLocal[query];
    NumCacheNonlocal++;

    SmallVector<BasicBlock*, 4> dirtied;
```

```
    for (DenseMap<BasicBlock*, Value*>::iterator I = cached.begin(),
         E = cached.end(); I != E; ++I)
      if (I->second == Dirty)
        dirtied.push_back(I->first);

    for (SmallVector<BasicBlock*, 4>::iterator I = dirtied.begin(),
         E = dirtied.end(); I != E; ++I) {
      Instruction* localDep = getDependency(query, 0, *I);
      if (localDep != NonLocal)
        cached[*I] = localDep;
      else {
        cached.erase(*I);
        nonLocalHelper(query, *I, cached);
      }
    }

    resp = cached;

    return;
  } else
    NumUncacheNonlocal++;

  // If not, go ahead and search for non-local deps.
  nonLocalHelper(query, query->getParent(), resp);

  // Update the non-local dependency cache
  for (DenseMap<BasicBlock*, Value*>::iterator I = resp.begin(), E = resp.end();
       I != E; ++I) {
    depGraphNonLocal[query].insert(*I);
    reverseDepNonLocal[I->second].insert(query);
  }
}

/// getDependency - Return the instruction on which a memory operation
/// depends.  The local paramter indicates if the query should only
/// evaluate dependencies within the same basic block.
Instruction* MemoryDependenceAnalysis::getDependency(Instruction* query,
                                                     Instruction* start,
                                                     BasicBlock* block) {
  // Start looking for dependencies with the queried inst
  BasicBlock::iterator QI = query;

  // Check for a cached result
  std::pair<Instruction*, bool>& cachedResult = depGraphLocal[query];
  // If we have a _confirmed_ cached entry, return it
  if (!block && !start) {
    if (cachedResult.second)
      return cachedResult.first;
    else if (cachedResult.first && cachedResult.first != NonLocal)
      // If we have an unconfirmed cached entry, we can start our search from there
      QI = cachedResult.first;
  }
```

```
if (start)
  QI = start;
else if (!start && block)
  QI = block->end();

AliasAnalysis& AA = getAnalysis<AliasAnalysis>();
TargetData& TD = getAnalysis<TargetData>();

// Get the pointer value for which dependence will be determined
Value* dependee = 0;
uint64_t dependeeSize = 0;
bool queryIsVolatile = false;
if (StoreInst* S = dyn_cast<StoreInst>(query)) {
  dependee = S->getPointerOperand();
  dependeeSize = TD.getTypeStoreSize(S->getOperand(0)->getType());
  queryIsVolatile = S->isVolatile();
} else if (LoadInst* L = dyn_cast<LoadInst>(query)) {
  dependee = L->getPointerOperand();
  dependeeSize = TD.getTypeStoreSize(L->getType());
  queryIsVolatile = L->isVolatile();
} else if (VAArgInst* V = dyn_cast<VAArgInst>(query)) {
  dependee = V->getOperand(0);
  dependeeSize = TD.getTypeStoreSize(V->getType());
} else if (FreeInst* F = dyn_cast<FreeInst>(query)) {
  dependee = F->getPointerOperand();

  // FreeInsts erase the entire structure, not just a field
  dependeeSize = ~0UL;
} else if (CallSite::get(query).getInstruction() != 0)
  return getCallSiteDependency(CallSite::get(query), start, block);
else if (isa<AllocationInst>(query))
  return None;
else
  return None;

BasicBlock::iterator blockBegin = block ? block->begin()
                                        : query->getParent()->begin();

// Walk backwards through the basic block, looking for dependencies
while (QI != blockBegin) {
  --QI;

  // If this inst is a memory op, get the pointer it accessed
  Value* pointer = 0;
  uint64_t pointerSize = 0;
  if (StoreInst* S = dyn_cast<StoreInst>(QI)) {
    // All volatile loads/stores depend on each other
    if (queryIsVolatile && S->isVolatile()) {
      if (!start && !block) {
        cachedResult.first = S;
        cachedResult.second = true;
        reverseDep[S].insert(query);
      }
```

```
      return S;
    }

    pointer = S->getPointerOperand();
    pointerSize = TD.getTypeStoreSize(S->getOperand(0)->getType());
  } else if (LoadInst* L = dyn_cast<LoadInst>(QI)) {
    // All volatile loads/stores depend on each other
    if (queryIsVolatile && L->isVolatile()) {
      if (!start && !block) {
        cachedResult.first = L;
        cachedResult.second = true;
        reverseDep[L].insert(query);
      }

      return L;
    }

    pointer = L->getPointerOperand();
    pointerSize = TD.getTypeStoreSize(L->getType());
  } else if (AllocationInst* AI = dyn_cast<AllocationInst>(QI)) {
    pointer = AI;
    if (ConstantInt* C = dyn_cast<ConstantInt>(AI->getArraySize()))
      pointerSize = C->getZExtValue() * \
                    TD.getABITypeSize(AI->getAllocatedType());
    else
      pointerSize = ~0UL;
  } else if (VAArgInst* V = dyn_cast<VAArgInst>(QI)) {
    pointer = V->getOperand(0);
    pointerSize = TD.getTypeStoreSize(V->getType());
  } else if (FreeInst* F = dyn_cast<FreeInst>(QI)) {
    pointer = F->getPointerOperand();

    // FreeInsts erase the entire structure
    pointerSize = ~0UL;
  } else if (CallSite::get(QI).getInstruction() != 0) {
    // Call insts need special handling. Check if they can modify our pointer
    AliasAnalysis::ModRefResult MR = AA.getModRefInfo(CallSite::get(QI),
                                                      dependee, dependeeSize);

    if (MR != AliasAnalysis::NoModRef) {
      // Loads don't depend on read-only calls
      if (isa<LoadInst>(query) && MR == AliasAnalysis::Ref)
        continue;

      if (!start && !block) {
        cachedResult.first = QI;
        cachedResult.second = true;
        reverseDep[QI].insert(query);
      }

      return QI;
    } else {
```

```
        continue;
      }
    }

    // If we found a pointer, check if it could be the same as our pointer
    if (pointer) {
      AliasAnalysis::AliasResult R = AA.alias(pointer, pointerSize,
                                              dependee, dependeeSize);

      if (R != AliasAnalysis::NoAlias) {
        // May-alias loads don't depend on each other
        if (isa<LoadInst>(query) && isa<LoadInst>(QI) &&
            R == AliasAnalysis::MayAlias)
          continue;

        if (!start && !block) {
          cachedResult.first = QI;
          cachedResult.second = true;
          reverseDep[QI].insert(query);
        }

        return QI;
      }
    }
  }

  // If we found nothing, return the non-local flag
  if (!start && !block) {
    cachedResult.first = NonLocal;
    cachedResult.second = true;
    reverseDep[NonLocal].insert(query);
  }

  return NonLocal;
}

/// dropInstruction - Remove an instruction from the analysis, making
/// absolutely conservative assumptions when updating the cache. This is
/// useful, for example when an instruction is changed rather than removed.
void MemoryDependenceAnalysis::dropInstruction(Instruction* drop) {
  depMapType::iterator depGraphEntry = depGraphLocal.find(drop);
  if (depGraphEntry != depGraphLocal.end())
    reverseDep[depGraphEntry->second.first].erase(drop);

  // Drop dependency information for things that depended on this instr
  SmallPtrSet<Instruction*, 4>& set = reverseDep[drop];
  for (SmallPtrSet<Instruction*, 4>::iterator I = set.begin(), E = set.end();
       I != E; ++I)
    depGraphLocal.erase(*I);

  depGraphLocal.erase(drop);
  reverseDep.erase(drop);
```

```
  for (DenseMap<BasicBlock*, Value*>::iterator DI =
       depGraphNonLocal[drop].begin(), DE = depGraphNonLocal[drop].end();
       DI != DE; ++DI)
    if (DI->second != None)
      reverseDepNonLocal[DI->second].erase(drop);

  if (reverseDepNonLocal.count(drop)) {
    SmallPtrSet<Instruction*, 4>& set = reverseDepNonLocal[drop];
    for (SmallPtrSet<Instruction*, 4>::iterator I = set.begin(), E = set.end();
         I != E; ++I)
      for (DenseMap<BasicBlock*, Value*>::iterator DI =
           depGraphNonLocal[*I].begin(), DE = depGraphNonLocal[*I].end();
           DI != DE; ++DI)
        if (DI->second == drop)
          DI->second = Dirty;
  }

  reverseDepNonLocal.erase(drop);
  nonLocalDepMapType::iterator I = depGraphNonLocal.find(drop);
  if (I != depGraphNonLocal.end())
    depGraphNonLocal.erase(I);
}

/// removeInstruction - Remove an instruction from the dependence analysis,
/// updating the dependence of instructions that previously depended on it.
/// This method attempts to keep the cache coherent using the reverse map.
void MemoryDependenceAnalysis::removeInstruction(Instruction* rem) {
  // Figure out the new dep for things that currently depend on rem
  Instruction* newDep = NonLocal;

  for (DenseMap<BasicBlock*, Value*>::iterator DI =
       depGraphNonLocal[rem].begin(), DE = depGraphNonLocal[rem].end();
       DI != DE; ++DI)
    if (DI->second != None)
      reverseDepNonLocal[DI->second].erase(rem);

  depMapType::iterator depGraphEntry = depGraphLocal.find(rem);

  if (depGraphEntry != depGraphLocal.end()) {
    reverseDep[depGraphEntry->second.first].erase(rem);

    if (depGraphEntry->second.first != NonLocal &&
        depGraphEntry->second.first != None &&
        depGraphEntry->second.second) {
      // If we have dep info for rem, set them to it
      BasicBlock::iterator RI = depGraphEntry->second.first;
      RI++;
      newDep = RI;
    } else if ( (depGraphEntry->second.first == NonLocal ||
                 depGraphEntry->second.first == None ) &&
               depGraphEntry->second.second ) {
      // If we have a confirmed non-local flag, use it
      newDep = depGraphEntry->second.first;
```

```
    } else {
      // Otherwise, use the immediate successor of rem
      // NOTE: This is because, when getDependence is called, it will first
      // check the immediate predecessor of what is in the cache.
      BasicBlock::iterator RI = rem;
      RI++;
      newDep = RI;
    }
  } else {
    // Otherwise, use the immediate successor of rem
    // NOTE: This is because, when getDependence is called, it will first
    // check the immediate predecessor of what is in the cache.
    BasicBlock::iterator RI = rem;
    RI++;
    newDep = RI;
  }

  SmallPtrSet<Instruction*, 4>& set = reverseDep[rem];
  for (SmallPtrSet<Instruction*, 4>::iterator I = set.begin(), E = set.end();
       I != E; ++I) {
    // Insert the new dependencies
    // Mark it as unconfirmed as long as it is not the non-local flag
    depGraphLocal[*I] = std::make_pair(newDep, (newDep == NonLocal ||
                                                newDep == None));
  }

  depGraphLocal.erase(rem);
  reverseDep.erase(rem);

  if (reverseDepNonLocal.count(rem)) {
    SmallPtrSet<Instruction*, 4>& set = reverseDepNonLocal[rem];
    for (SmallPtrSet<Instruction*, 4>::iterator I = set.begin(), E = set.end();
         I != E; ++I)
      for (DenseMap<BasicBlock*, Value*>::iterator DI =
           depGraphNonLocal[*I].begin(), DE = depGraphNonLocal[*I].end();
           DI != DE; ++DI)
        if (DI->second == rem)
          DI->second = Dirty;

  }

  reverseDepNonLocal.erase(rem);
  nonLocalDepMapType::iterator I = depGraphNonLocal.find(rem);
  if (I != depGraphNonLocal.end())
    depGraphNonLocal.erase(I);

  getAnalysis<AliasAnalysis>().deleteValue(rem);
}
```

```
//===- DeadStoreElimination.cpp - Fast Dead Store Elimination ----------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file implements a trivial dead store elimination that only considers
// basic-block local redundant stores.
//
// FIXME: This should eventually be extended to be a post-dominator tree
// traversal.  Doing so would be pretty trivial.
//
//===----------------------------------------------------------------------===//

#define DEBUG_TYPE "dse"
#include "llvm/Transforms/Scalar.h"
#include "llvm/Constants.h"
#include "llvm/Function.h"
#include "llvm/Instructions.h"
#include "llvm/IntrinsicInst.h"
#include "llvm/Pass.h"
#include "llvm/ADT/SetVector.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/MemoryDependenceAnalysis.h"
#include "llvm/Target/TargetData.h"
#include "llvm/Transforms/Utils/Local.h"
#include "llvm/Support/Compiler.h"
using namespace llvm;

STATISTIC(NumFastStores, "Number of stores deleted");
STATISTIC(NumFastOther , "Number of other instrs removed");

namespace {
  struct VISIBILITY_HIDDEN DSE : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    DSE() : FunctionPass((intptr_t)&ID) {}

    virtual bool runOnFunction(Function &F) {
      bool Changed = false;
      for (Function::iterator I = F.begin(), E = F.end(); I != E; ++I)
        Changed |= runOnBasicBlock(*I);
      return Changed;
    }

    bool runOnBasicBlock(BasicBlock &BB);
    bool handleFreeWithNonTrivialDependency(FreeInst* F,
                                            Instruction* dependency,
```

```
                                    SetVector<Instruction*>& possiblyDead);
    bool handleEndBlock(BasicBlock& BB, SetVector<Instruction*>& possiblyDead);
    bool RemoveUndeadPointers(Value* pointer, uint64_t killPointerSize,
                              BasicBlock::iterator& BBI,
                              SmallPtrSet<Value*, 64>& deadPointers,
                              SetVector<Instruction*>& possiblyDead);
    void DeleteDeadInstructionChains(Instruction *I,
                                     SetVector<Instruction*> &DeadInsts);

    /// Find the base pointer that a pointer came from
    /// Because this is used to find pointers that originate
    /// from allocas, it is safe to ignore GEP indices, since
    /// either the store will be in the alloca, and thus dead,
    /// or beyond the end of the alloca, and thus undefined.
    void TranslatePointerBitCasts(Value*& v, bool zeroGepsOnly = false) {
      assert(isa<PointerType>(v->getType()) &&
             "Translating a non-pointer type?");
      while (true) {
        if (BitCastInst* C = dyn_cast<BitCastInst>(v))
          v = C->getOperand(0);
        else if (GetElementPtrInst* G = dyn_cast<GetElementPtrInst>(v))
          if (!zeroGepsOnly || G->hasAllZeroIndices()) {
            v = G->getOperand(0);
          } else {
            break;
          }
        else
          break;
      }
    }

    // getAnalysisUsage - We require post dominance frontiers (aka Control
    // Dependence Graph)
    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
      AU.setPreservesCFG();
      AU.addRequired<TargetData>();
      AU.addRequired<AliasAnalysis>();
      AU.addRequired<MemoryDependenceAnalysis>();
      AU.addPreserved<AliasAnalysis>();
      AU.addPreserved<MemoryDependenceAnalysis>();
    }
  };
  char DSE::ID = 0;
  RegisterPass<DSE> X("dse", "Dead Store Elimination");
}

FunctionPass *llvm::createDeadStoreEliminationPass() { return new DSE(); }

bool DSE::runOnBasicBlock(BasicBlock &BB) {
  MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();
  TargetData &TD = getAnalysis<TargetData>();

  // Record the last-seen store to this pointer
```

```
DenseMap<Value*, StoreInst*> lastStore;
// Record instructions possibly made dead by deleting a store
SetVector<Instruction*> possiblyDead;

bool MadeChange = false;

// Do a top-down walk on the BB
for (BasicBlock::iterator BBI = BB.begin(), BBE = BB.end();
     BBI != BBE; ++BBI) {
  // If we find a store or a free...
  if (!isa<StoreInst>(BBI) && !isa<FreeInst>(BBI))
    continue;

  Value* pointer = 0;
  if (StoreInst* S = dyn_cast<StoreInst>(BBI)) {
    if (!S->isVolatile())
      pointer = S->getPointerOperand();
    else
      continue;
  } else
    pointer = cast<FreeInst>(BBI)->getPointerOperand();

  TranslatePointerBitCasts(pointer, true);
  StoreInst*& last = lastStore[pointer];
  bool deletedStore = false;

  // ... to a pointer that has been stored to before...
  if (last) {
    Instruction* dep = MD.getDependency(BBI);

    // ... and no other memory dependencies are between them....
    while (dep != MemoryDependenceAnalysis::None &&
           dep != MemoryDependenceAnalysis::NonLocal &&
           isa<StoreInst>(dep)) {
      if (dep != last ||
           TD.getTypeStoreSize(last->getOperand(0)->getType()) >
           TD.getTypeStoreSize(BBI->getOperand(0)->getType())) {
        dep = MD.getDependency(BBI, dep);
        continue;
      }

      // Remove it!
      MD.removeInstruction(last);

      // DCE instructions only used to calculate that store
      if (Instruction* D = dyn_cast<Instruction>(last->getOperand(0)))
        possiblyDead.insert(D);
      if (Instruction* D = dyn_cast<Instruction>(last->getOperand(1)))
        possiblyDead.insert(D);

      last->eraseFromParent();
      NumFastStores++;
      deletedStore = true;
```

```
        MadeChange = true;

        break;
      }
    }

    // Handle frees whose dependencies are non-trivial.
    if (FreeInst* F = dyn_cast<FreeInst>(BBI)) {
      if (!deletedStore)
        MadeChange |= handleFreeWithNonTrivialDependency(F,
                                                        MD.getDependency(F),
                                                        possiblyDead);
      // No known stores after the free
      last = 0;
    } else {
      // Update our most-recent-store map.
      last = cast<StoreInst>(BBI);
    }
  }

  // If this block ends in a return, unwind, unreachable, and eventually
  // tailcall, then all allocas are dead at its end.
  if (BB.getTerminator()->getNumSuccessors() == 0)
    MadeChange |= handleEndBlock(BB, possiblyDead);

  // Do a trivial DCE
  while (!possiblyDead.empty()) {
    Instruction *I = possiblyDead.back();
    possiblyDead.pop_back();
    DeleteDeadInstructionChains(I, possiblyDead);
  }

  return MadeChange;
}

/// handleFreeWithNonTrivialDependency - Handle frees of entire structures whose
/// dependency is a store to a field of that structure
bool DSE::handleFreeWithNonTrivialDependency(FreeInst* F, Instruction* dep,
                                             SetVector<Instruction*>& possiblyDead) {
  TargetData &TD = getAnalysis<TargetData>();
  AliasAnalysis &AA = getAnalysis<AliasAnalysis>();
  MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();

  if (dep == MemoryDependenceAnalysis::None ||
      dep == MemoryDependenceAnalysis::NonLocal)
    return false;

  StoreInst* dependency = dyn_cast<StoreInst>(dep);
  if (!dependency)
    return false;
  else if (dependency->isVolatile())
    return false;
```

```
  Value* depPointer = dependency->getPointerOperand();
  const Type* depType = dependency->getOperand(0)->getType();
  unsigned depPointerSize = TD.getTypeStoreSize(depType);

  // Check for aliasing
  AliasAnalysis::AliasResult A = AA.alias(F->getPointerOperand(), ~0U,
                                          depPointer, depPointerSize);

  if (A == AliasAnalysis::MustAlias) {
    // Remove it!
    MD.removeInstruction(dependency);

    // DCE instructions only used to calculate that store
    if (Instruction* D = dyn_cast<Instruction>(dependency->getOperand(0)))
      possiblyDead.insert(D);
    if (Instruction* D = dyn_cast<Instruction>(dependency->getOperand(1)))
      possiblyDead.insert(D);

    dependency->eraseFromParent();
    NumFastStores++;
    return true;
  }

  return false;
}

/// handleEndBlock - Remove dead stores to stack-allocated locations in the
/// function end block.   Ex:
/// %A = alloca i32
/// ...
/// store i32 1, i32* %A
/// ret void
bool DSE::handleEndBlock(BasicBlock& BB,
                         SetVector<Instruction*>& possiblyDead) {
  TargetData &TD = getAnalysis<TargetData>();
  AliasAnalysis &AA = getAnalysis<AliasAnalysis>();
  MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();

  bool MadeChange = false;

  // Pointers alloca'd in this function are dead in the end block
  SmallPtrSet<Value*, 64> deadPointers;

  // Find all of the alloca'd pointers in the entry block
  BasicBlock *Entry = BB.getParent()->begin();
  for (BasicBlock::iterator I = Entry->begin(), E = Entry->end(); I != E; ++I)
    if (AllocaInst *AI = dyn_cast<AllocaInst>(I))
      deadPointers.insert(AI);
  for (Function::arg_iterator AI = BB.getParent()->arg_begin(),
       AE = BB.getParent()->arg_end(); AI != AE; ++AI)
    if (AI->hasByValAttr())
      deadPointers.insert(AI);
```

```
// Scan the basic block backwards
for (BasicBlock::iterator BBI = BB.end(); BBI != BB.begin(); ){
  --BBI;

  // If we find a store whose pointer is dead...
  if (StoreInst* S = dyn_cast<StoreInst>(BBI)) {
    if (!S->isVolatile()) {
      Value* pointerOperand = S->getPointerOperand();
      // See through pointer-to-pointer bitcasts
      TranslatePointerBitCasts(pointerOperand);

      // Alloca'd pointers or byval arguments (which are functionally like
      // alloca's) are valid candidates for removal.
      if (deadPointers.count(pointerOperand)) {
        // Remove it!
        MD.removeInstruction(S);

        // DCE instructions only used to calculate that store
        if (Instruction* D = dyn_cast<Instruction>(S->getOperand(0)))
          possiblyDead.insert(D);
        if (Instruction* D = dyn_cast<Instruction>(S->getOperand(1)))
          possiblyDead.insert(D);

        BBI++;
        S->eraseFromParent();
        NumFastStores++;
        MadeChange = true;
      }
    }

    continue;

  // We can also remove memcpy's to local variables at the end of a function
  } else if (MemCpyInst* M = dyn_cast<MemCpyInst>(BBI)) {
    Value* dest = M->getDest();
    TranslatePointerBitCasts(dest);

    if (deadPointers.count(dest)) {
      MD.removeInstruction(M);

      // DCE instructions only used to calculate that memcpy
      if (Instruction* D = dyn_cast<Instruction>(M->getRawSource()))
        possiblyDead.insert(D);
      if (Instruction* D = dyn_cast<Instruction>(M->getLength()))
        possiblyDead.insert(D);
      if (Instruction* D = dyn_cast<Instruction>(M->getRawDest()))
        possiblyDead.insert(D);

      BBI++;
      M->eraseFromParent();
      NumFastOther++;
      MadeChange = true;
```

```
      continue;
    }

    // Because a memcpy is also a load, we can't skip it if we didn't remove it
  }

  Value* killPointer = 0;
  uint64_t killPointerSize = ~0UL;

  // If we encounter a use of the pointer, it is no longer considered dead
  if (LoadInst* L = dyn_cast<LoadInst>(BBI)) {
    // However, if this load is unused, we can go ahead and remove it, and
    // not have to worry about it making our pointer undead!
    if (L->use_empty()) {
      MD.removeInstruction(L);

      // DCE instructions only used to calculate that load
      if (Instruction* D = dyn_cast<Instruction>(L->getPointerOperand()))
        possiblyDead.insert(D);

      BBI++;
      L->eraseFromParent();
      NumFastOther++;
      MadeChange = true;
      possiblyDead.remove(L);

      continue;
    }

    killPointer = L->getPointerOperand();
  } else if (VAArgInst* V = dyn_cast<VAArgInst>(BBI)) {
    killPointer = V->getOperand(0);
  } else if (isa<MemCpyInst>(BBI) &&
             isa<ConstantInt>(cast<MemCpyInst>(BBI)->getLength())) {
    killPointer = cast<MemCpyInst>(BBI)->getSource();
    killPointerSize = cast<ConstantInt>(
                        cast<MemCpyInst>(BBI)->getLength())->getZExtValue();
  } else if (AllocaInst* A = dyn_cast<AllocaInst>(BBI)) {
    deadPointers.erase(A);

    // Dead alloca's can be DCE'd when we reach them
    if (A->use_empty()) {
      MD.removeInstruction(A);

      // DCE instructions only used to calculate that load
      if (Instruction* D = dyn_cast<Instruction>(A->getArraySize()))
        possiblyDead.insert(D);

      BBI++;
      A->eraseFromParent();
      NumFastOther++;
      MadeChange = true;
      possiblyDead.remove(A);
```

```
    }

    continue;
  } else if (CallSite::get(BBI).getInstruction() != 0) {
    // If this call does not access memory, it can't
    // be undeadifying any of our pointers.
    CallSite CS = CallSite::get(BBI);
    if (AA.doesNotAccessMemory(CS))
      continue;

    unsigned modRef = 0;
    unsigned other = 0;

    // Remove any pointers made undead by the call from the dead set
    std::vector<Value*> dead;
    for (SmallPtrSet<Value*, 64>::iterator I = deadPointers.begin(),
         E = deadPointers.end(); I != E; ++I) {
      // HACK: if we detect that our AA is imprecise, it's not
      // worth it to scan the rest of the deadPointers set.   Just
      // assume that the AA will return ModRef for everything, and
      // go ahead and bail.
      if (modRef >= 16 && other == 0) {
        deadPointers.clear();
        return MadeChange;
      }

      // Get size information for the alloca
      unsigned pointerSize = ~0U;
      if (AllocaInst* A = dyn_cast<AllocaInst>(*I)) {
        if (ConstantInt* C = dyn_cast<ConstantInt>(A->getArraySize()))
          pointerSize = C->getZExtValue() * \
                        TD.getABITypeSize(A->getAllocatedType());
      } else {
        const PointerType* PT = cast<PointerType>(
                                       cast<Argument>(*I)->getType());
        pointerSize = TD.getABITypeSize(PT->getElementType());
      }

      // See if the call site touches it
      AliasAnalysis::ModRefResult A = AA.getModRefInfo(CS, *I, pointerSize);

      if (A == AliasAnalysis::ModRef)
        modRef++;
      else
        other++;

      if (A == AliasAnalysis::ModRef || A == AliasAnalysis::Ref)
        dead.push_back(*I);
    }

    for (std::vector<Value*>::iterator I = dead.begin(), E = dead.end();
         I != E; ++I)
      deadPointers.erase(*I);
```

```
      continue;
    } else {
      // For any non-memory-affecting non-terminators, DCE them as we reach them
      Instruction *CI = BBI;
      if (!CI->isTerminator() && CI->use_empty() && !isa<FreeInst>(CI)) {

        // DCE instructions only used to calculate that load
        for (Instruction::op_iterator OI = CI->op_begin(), OE = CI->op_end();
             OI != OE; ++OI)
          if (Instruction* D = dyn_cast<Instruction>(OI))
            possiblyDead.insert(D);

        BBI++;
        CI->eraseFromParent();
        NumFastOther++;
        MadeChange = true;
        possiblyDead.remove(CI);

        continue;
      }
    }

    if (!killPointer)
      continue;

    TranslatePointerBitCasts(killPointer);

    // Deal with undead pointers
    MadeChange |= RemoveUndeadPointers(killPointer, killPointerSize, BBI,
                                       deadPointers, possiblyDead);
  }

  return MadeChange;
}

/// RemoveUndeadPointers - check for uses of a pointer that make it
/// undead when scanning for dead stores to alloca's.
bool DSE::RemoveUndeadPointers(Value* killPointer, uint64_t killPointerSize,
                               BasicBlock::iterator& BBI,
                               SmallPtrSet<Value*, 64>& deadPointers,
                               SetVector<Instruction*>& possiblyDead) {
  TargetData &TD = getAnalysis<TargetData>();
  AliasAnalysis &AA = getAnalysis<AliasAnalysis>();
  MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();

  // If the kill pointer can be easily reduced to an alloca,
  // don't bother doing extraneous AA queries
  if (deadPointers.count(killPointer)) {
    deadPointers.erase(killPointer);
    return false;
  } else if (isa<GlobalValue>(killPointer)) {
    // A global can't be in the dead pointer set
```

```
    return false;
  }

  bool MadeChange = false;

  std::vector<Value*> undead;

  for (SmallPtrSet<Value*, 64>::iterator I = deadPointers.begin(),
      E = deadPointers.end(); I != E; ++I) {
    // Get size information for the alloca
    unsigned pointerSize = ~0U;
    if (AllocaInst* A = dyn_cast<AllocaInst>(*I)) {
      if (ConstantInt* C = dyn_cast<ConstantInt>(A->getArraySize()))
        pointerSize = C->getZExtValue() * \
                      TD.getABITypeSize(A->getAllocatedType());
    } else {
      const PointerType* PT = cast<PointerType>(
                                                cast<Argument>(*I)->getType());
      pointerSize = TD.getABITypeSize(PT->getElementType());
    }

    // See if this pointer could alias it
    AliasAnalysis::AliasResult A = AA.alias(*I, pointerSize,
                                            killPointer, killPointerSize);

    // If it must-alias and a store, we can delete it
    if (isa<StoreInst>(BBI) && A == AliasAnalysis::MustAlias) {
      StoreInst* S = cast<StoreInst>(BBI);

      // Remove it!
      MD.removeInstruction(S);

      // DCE instructions only used to calculate that store
      if (Instruction* D = dyn_cast<Instruction>(S->getOperand(0)))
        possiblyDead.insert(D);
      if (Instruction* D = dyn_cast<Instruction>(S->getOperand(1)))
        possiblyDead.insert(D);

      BBI++;
      S->eraseFromParent();
      NumFastStores++;
      MadeChange = true;

      continue;

      // Otherwise, it is undead
      } else if (A != AliasAnalysis::NoAlias)
        undead.push_back(*I);
  }

  for (std::vector<Value*>::iterator I = undead.begin(), E = undead.end();
       I != E; ++I)
      deadPointers.erase(*I);
```

```
  return MadeChange;
}

/// DeleteDeadInstructionChains - takes an instruction and a setvector of
/// dead instructions.  If I is dead, it is erased, and its operands are
/// checked for deadness.  If they are dead, they are added to the dead
/// setvector.
void DSE::DeleteDeadInstructionChains(Instruction *I,
                                      SetVector<Instruction*> &DeadInsts) {
  // Instruction must be dead.
  if (!I->use_empty() || !isInstructionTriviallyDead(I)) return;

  // Let the memory dependence know
  getAnalysis<MemoryDependenceAnalysis>().removeInstruction(I);

  // See if this made any operands dead. We do it this way in case the
  // instruction uses the same operand twice.  We don't want to delete a
  // value then reference it.
  for (unsigned i = 0, e = I->getNumOperands(); i != e; ++i) {
    if (I->getOperand(i)->hasOneUse())
      if (Instruction* Op = dyn_cast<Instruction>(I->getOperand(i)))
        DeadInsts.insert(Op);     // Attempt to nuke it later.

    I->setOperand(i, 0);        // Drop from the operand list.
  }

  I->eraseFromParent();
  ++NumFastOther;
}
```

```
//===- GVN.cpp - Eliminate redundant values and loads ————===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===———————————————————————————————-===//
//
// This pass performs global value numbering to eliminate fully redundant
// instructions.  It also performs simple dead load elimination.
//
//===———————————————————————————————-===//

#define DEBUG_TYPE "gvn"

#include "llvm/Transforms/Scalar.h"
#include "llvm/BasicBlock.h"
#include "llvm/Constants.h"
#include "llvm/DerivedTypes.h"
#include "llvm/Function.h"
#include "llvm/IntrinsicInst.h"
#include "llvm/Instructions.h"
#include "llvm/ParameterAttributes.h"
#include "llvm/Value.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Analysis/Dominators.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/MemoryDependenceAnalysis.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/Compiler.h"
#include "llvm/Target/TargetData.h"
using namespace llvm;

//===———————————————————————————————-===//
//                         ValueTable Class
//===———————————————————————————————-===//

/// This class holds the mapping between values and value numbers. It is used
/// as an efficient mechanism to determine the expression-wise equivalence of
/// two values.
namespace {
  struct VISIBILITY_HIDDEN Expression {
    enum ExpressionOpcode { ADD, SUB, MUL, UDIV, SDIV, FDIV, UREM, SREM,
                            FREM, SHL, LSHR, ASHR, AND, OR, XOR, ICMPEQ,
                            ICMPNE, ICMPUGT, ICMPUGE, ICMPULT, ICMPULE,
                            ICMPSGT, ICMPSGE, ICMPSLT, ICMPSLE, FCMPOEQ,
```

```
                        FCMPOGT, FCMPOGE, FCMPOLT, FCMPOLE, FCMPONE,
                        FCMPORD, FCMPUNO, FCMPUEQ, FCMPUGT, FCMPUGE,
                        FCMPULT, FCMPULE, FCMPUNE, EXTRACT, INSERT,
                        SHUFFLE, SELECT, TRUNC, ZEXT, SEXT, FPTOUI,
                        FPTOSI, UITOFP, SITOFP, FPTRUNC, FPEXT,
                        PTRTOINT, INTTOPTR, BITCAST, GEP, CALL, EMPTY,
                        TOMBSTONE };

ExpressionOpcode opcode;
const Type* type;
uint32_t firstVN;
uint32_t secondVN;
uint32_t thirdVN;
SmallVector<uint32_t, 4> varargs;
Value* function;

Expression() { }
Expression(ExpressionOpcode o) : opcode(o) { }

bool operator==(const Expression &other) const {
  if (opcode != other.opcode)
    return false;
  else if (opcode == EMPTY || opcode == TOMBSTONE)
    return true;
  else if (type != other.type)
    return false;
  else if (function != other.function)
    return false;
  else if (firstVN != other.firstVN)
    return false;
  else if (secondVN != other.secondVN)
    return false;
  else if (thirdVN != other.thirdVN)
    return false;
  else {
    if (varargs.size() != other.varargs.size())
      return false;

    for (size_t i = 0; i < varargs.size(); ++i)
      if (varargs[i] != other.varargs[i])
        return false;

    return true;
  }
}

bool operator!=(const Expression &other) const {
  if (opcode != other.opcode)
    return true;
  else if (opcode == EMPTY || opcode == TOMBSTONE)
    return false;
  else if (type != other.type)
    return true;
```

```
      else if (function != other.function)
        return true;
      else if (firstVN != other.firstVN)
        return true;
      else if (secondVN != other.secondVN)
        return true;
      else if (thirdVN != other.thirdVN)
        return true;
      else {
        if (varargs.size() != other.varargs.size())
          return true;

        for (size_t i = 0; i < varargs.size(); ++i)
          if (varargs[i] != other.varargs[i])
            return true;

          return false;
      }
    }
  };

  class VISIBILITY_HIDDEN ValueTable {
    private:
      DenseMap<Value*, uint32_t> valueNumbering;
      DenseMap<Expression, uint32_t> expressionNumbering;
      AliasAnalysis* AA;

      uint32_t nextValueNumber;

      Expression::ExpressionOpcode getOpcode(BinaryOperator* BO);
      Expression::ExpressionOpcode getOpcode(CmpInst* C);
      Expression::ExpressionOpcode getOpcode(CastInst* C);
      Expression create_expression(BinaryOperator* BO);
      Expression create_expression(CmpInst* C);
      Expression create_expression(ShuffleVectorInst* V);
      Expression create_expression(ExtractElementInst* C);
      Expression create_expression(InsertElementInst* V);
      Expression create_expression(SelectInst* V);
      Expression create_expression(CastInst* C);
      Expression create_expression(GetElementPtrInst* G);
      Expression create_expression(CallInst* C);
    public:
      ValueTable() : nextValueNumber(1) { }
      uint32_t lookup_or_add(Value* V);
      uint32_t lookup(Value* V) const;
      void add(Value* V, uint32_t num);
      void clear();
      void erase(Value* v);
      unsigned size();
      void setAliasAnalysis(AliasAnalysis* A) { AA = A; }
      uint32_t hash_operand(Value* v);
  };
}
```

```
namespace llvm {
template <> struct DenseMapInfo<Expression> {
  static inline Expression getEmptyKey() {
    return Expression(Expression::EMPTY);
  }

  static inline Expression getTombstoneKey() {
    return Expression(Expression::TOMBSTONE);
  }

  static unsigned getHashValue(const Expression e) {
    unsigned hash = e.opcode;

    hash = e.firstVN + hash * 37;
    hash = e.secondVN + hash * 37;
    hash = e.thirdVN + hash * 37;

    hash = ((unsigned)((uintptr_t)e.type >> 4) ^
            (unsigned)((uintptr_t)e.type >> 9)) +
           hash * 37;

    for (SmallVector<uint32_t, 4>::const_iterator I = e.varargs.begin(),
         E = e.varargs.end(); I != E; ++I)
      hash = *I + hash * 37;

    hash = ((unsigned)((uintptr_t)e.function >> 4) ^
            (unsigned)((uintptr_t)e.function >> 9)) +
           hash * 37;

    return hash;
  }
  static bool isEqual(const Expression &LHS, const Expression &RHS) {
    return LHS == RHS;
  }
  static bool isPod() { return true; }
};
}

//===----------------------------------------------------------------------===//
//                     ValueTable Internal Functions
//===----------------------------------------------------------------------===//
Expression::ExpressionOpcode
                             ValueTable::getOpcode(BinaryOperator* BO) {
  switch(BO->getOpcode()) {
    case Instruction::Add:
      return Expression::ADD;
    case Instruction::Sub:
      return Expression::SUB;
    case Instruction::Mul:
      return Expression::MUL;
    case Instruction::UDiv:
      return Expression::UDIV;
```

```
    case Instruction::SDiv:
      return Expression::SDIV;
    case Instruction::FDiv:
      return Expression::FDIV;
    case Instruction::URem:
      return Expression::UREM;
    case Instruction::SRem:
      return Expression::SREM;
    case Instruction::FRem:
      return Expression::FREM;
    case Instruction::Shl:
      return Expression::SHL;
    case Instruction::LShr:
      return Expression::LSHR;
    case Instruction::AShr:
      return Expression::ASHR;
    case Instruction::And:
      return Expression::AND;
    case Instruction::Or:
      return Expression::OR;
    case Instruction::Xor:
      return Expression::XOR;

    // THIS SHOULD NEVER HAPPEN
    default:
      assert(0 && "Binary operator with unknown opcode?");
      return Expression::ADD;
  }
}

Expression::ExpressionOpcode ValueTable::getOpcode(CmpInst* C) {
  if (C->getOpcode() == Instruction::ICmp) {
    switch (C->getPredicate()) {
      case ICmpInst::ICMP_EQ:
        return Expression::ICMPEQ;
      case ICmpInst::ICMP_NE:
        return Expression::ICMPNE;
      case ICmpInst::ICMP_UGT:
        return Expression::ICMPUGT;
      case ICmpInst::ICMP_UGE:
        return Expression::ICMPUGE;
      case ICmpInst::ICMP_ULT:
        return Expression::ICMPULT;
      case ICmpInst::ICMP_ULE:
        return Expression::ICMPULE;
      case ICmpInst::ICMP_SGT:
        return Expression::ICMPSGT;
      case ICmpInst::ICMP_SGE:
        return Expression::ICMPSGE;
      case ICmpInst::ICMP_SLT:
        return Expression::ICMPSLT;
      case ICmpInst::ICMP_SLE:
        return Expression::ICMPSLE;
```

```
      // THIS SHOULD NEVER HAPPEN
      default:
        assert(0 && "Comparison with unknown predicate?");
        return Expression::ICMPEQ;
    }
  } else {
    switch (C->getPredicate()) {
      case FCmpInst::FCMP_OEQ:
        return Expression::FCMPOEQ;
      case FCmpInst::FCMP_OGT:
        return Expression::FCMPOGT;
      case FCmpInst::FCMP_OGE:
        return Expression::FCMPOGE;
      case FCmpInst::FCMP_OLT:
        return Expression::FCMPOLT;
      case FCmpInst::FCMP_OLE:
        return Expression::FCMPOLE;
      case FCmpInst::FCMP_ONE:
        return Expression::FCMPONE;
      case FCmpInst::FCMP_ORD:
        return Expression::FCMPORD;
      case FCmpInst::FCMP_UNO:
        return Expression::FCMPUNO;
      case FCmpInst::FCMP_UEQ:
        return Expression::FCMPUEQ;
      case FCmpInst::FCMP_UGT:
        return Expression::FCMPUGT;
      case FCmpInst::FCMP_UGE:
        return Expression::FCMPUGE;
      case FCmpInst::FCMP_ULT:
        return Expression::FCMPULT;
      case FCmpInst::FCMP_ULE:
        return Expression::FCMPULE;
      case FCmpInst::FCMP_UNE:
        return Expression::FCMPUNE;

      // THIS SHOULD NEVER HAPPEN
      default:
        assert(0 && "Comparison with unknown predicate?");
        return Expression::FCMPOEQ;
    }
  }
}

Expression::ExpressionOpcode
                              ValueTable::getOpcode(CastInst* C) {
  switch(C->getOpcode()) {
    case Instruction::Trunc:
      return Expression::TRUNC;
    case Instruction::ZExt:
      return Expression::ZEXT;
    case Instruction::SExt:
```

```
    return Expression::SEXT;
  case Instruction::FPToUI:
    return Expression::FPTOUI;
  case Instruction::FPToSI:
    return Expression::FPTOSI;
  case Instruction::UIToFP:
    return Expression::UITOFP;
  case Instruction::SIToFP:
    return Expression::SITOFP;
  case Instruction::FPTrunc:
    return Expression::FPTRUNC;
  case Instruction::FPExt:
    return Expression::FPEXT;
  case Instruction::PtrToInt:
    return Expression::PTRTOINT;
  case Instruction::IntToPtr:
    return Expression::INTTOPTR;
  case Instruction::BitCast:
    return Expression::BITCAST;

  // THIS SHOULD NEVER HAPPEN
  default:
    assert(0 && "Cast operator with unknown opcode?");
    return Expression::BITCAST;
 }
}

uint32_t ValueTable::hash_operand(Value* v) {
 if (CallInst* CI = dyn_cast<CallInst>(v))
  if (!AA->doesNotAccessMemory(CI))
    return nextValueNumber++;

 return lookup_or_add(v);
}

Expression ValueTable::create_expression(CallInst* C) {
 Expression e;

 e.type = C->getType();
 e.firstVN = 0;
 e.secondVN = 0;
 e.thirdVN = 0;
 e.function = C->getCalledFunction();
 e.opcode = Expression::CALL;

 for (CallInst::op_iterator I = C->op_begin()+1, E = C->op_end();
      I != E; ++I)
  e.varargs.push_back(hash_operand(*I));

 return e;
}

Expression ValueTable::create_expression(BinaryOperator* BO) {
```

```
  Expression e;

  e.firstVN = hash_operand(BO->getOperand(0));
  e.secondVN = hash_operand(BO->getOperand(1));
  e.thirdVN = 0;
  e.function = 0;
  e.type = BO->getType();
  e.opcode = getOpcode(BO);

  return e;
}

Expression ValueTable::create_expression(CmpInst* C) {
  Expression e;

  e.firstVN = hash_operand(C->getOperand(0));
  e.secondVN = hash_operand(C->getOperand(1));
  e.thirdVN = 0;
  e.function = 0;
  e.type = C->getType();
  e.opcode = getOpcode(C);

  return e;
}

Expression ValueTable::create_expression(CastInst* C) {
  Expression e;

  e.firstVN = hash_operand(C->getOperand(0));
  e.secondVN = 0;
  e.thirdVN = 0;
  e.function = 0;
  e.type = C->getType();
  e.opcode = getOpcode(C);

  return e;
}

Expression ValueTable::create_expression(ShuffleVectorInst* S) {
  Expression e;

  e.firstVN = hash_operand(S->getOperand(0));
  e.secondVN = hash_operand(S->getOperand(1));
  e.thirdVN = hash_operand(S->getOperand(2));
  e.function = 0;
  e.type = S->getType();
  e.opcode = Expression::SHUFFLE;

  return e;
}

Expression ValueTable::create_expression(ExtractElementInst* E) {
  Expression e;
```

```
  e.firstVN = hash_operand(E->getOperand(0));
  e.secondVN = hash_operand(E->getOperand(1));
  e.thirdVN = 0;
  e.function = 0;
  e.type = E->getType();
  e.opcode = Expression::EXTRACT;

  return e;
}

Expression ValueTable::create_expression(InsertElementInst* I) {
  Expression e;

  e.firstVN = hash_operand(I->getOperand(0));
  e.secondVN = hash_operand(I->getOperand(1));
  e.thirdVN = hash_operand(I->getOperand(2));
  e.function = 0;
  e.type = I->getType();
  e.opcode = Expression::INSERT;

  return e;
}

Expression ValueTable::create_expression(SelectInst* I) {
  Expression e;

  e.firstVN = hash_operand(I->getCondition());
  e.secondVN = hash_operand(I->getTrueValue());
  e.thirdVN = hash_operand(I->getFalseValue());
  e.function = 0;
  e.type = I->getType();
  e.opcode = Expression::SELECT;

  return e;
}

Expression ValueTable::create_expression(GetElementPtrInst* G) {
  Expression e;

  e.firstVN = hash_operand(G->getPointerOperand());
  e.secondVN = 0;
  e.thirdVN = 0;
  e.function = 0;
  e.type = G->getType();
  e.opcode = Expression::GEP;

  for (GetElementPtrInst::op_iterator I = G->idx_begin(), E = G->idx_end();
       I != E; ++I)
    e.varargs.push_back(hash_operand(*I));

  return e;
}
```

```
//===----------------------------------------------------------------------===//
//                     ValueTable External Functions
//===----------------------------------------------------------------------===//

/// lookup_or_add - Returns the value number for the specified value, assigning
/// it a new number if it did not have one before.
uint32_t ValueTable::lookup_or_add(Value* V) {
  DenseMap<Value*, uint32_t>::iterator VI = valueNumbering.find(V);
  if (VI != valueNumbering.end())
    return VI->second;

  if (CallInst* C = dyn_cast<CallInst>(V)) {
    if (AA->onlyReadsMemory(C)) { // includes doesNotAccessMemory
      Expression e = create_expression(C);

      DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
      if (EI != expressionNumbering.end()) {
        valueNumbering.insert(std::make_pair(V, EI->second));
        return EI->second;
      } else {
        expressionNumbering.insert(std::make_pair(e, nextValueNumber));
        valueNumbering.insert(std::make_pair(V, nextValueNumber));

        return nextValueNumber++;
      }
    } else {
      valueNumbering.insert(std::make_pair(V, nextValueNumber));
      return nextValueNumber++;
    }
  } else if (BinaryOperator* BO = dyn_cast<BinaryOperator>(V)) {
    Expression e = create_expression(BO);

    DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
    if (EI != expressionNumbering.end()) {
      valueNumbering.insert(std::make_pair(V, EI->second));
      return EI->second;
    } else {
      expressionNumbering.insert(std::make_pair(e, nextValueNumber));
      valueNumbering.insert(std::make_pair(V, nextValueNumber));

      return nextValueNumber++;
    }
  } else if (CmpInst* C = dyn_cast<CmpInst>(V)) {
    Expression e = create_expression(C);

    DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
    if (EI != expressionNumbering.end()) {
      valueNumbering.insert(std::make_pair(V, EI->second));
      return EI->second;
    } else {
      expressionNumbering.insert(std::make_pair(e, nextValueNumber));
      valueNumbering.insert(std::make_pair(V, nextValueNumber));
```

```
    return nextValueNumber++;
  }
} else if (ShuffleVectorInst* U = dyn_cast<ShuffleVectorInst>(V)) {
  Expression e = create_expression(U);

  DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
  if (EI != expressionNumbering.end()) {
    valueNumbering.insert(std::make_pair(V, EI->second));
    return EI->second;
  } else {
    expressionNumbering.insert(std::make_pair(e, nextValueNumber));
    valueNumbering.insert(std::make_pair(V, nextValueNumber));

    return nextValueNumber++;
  }
} else if (ExtractElementInst* U = dyn_cast<ExtractElementInst>(V)) {
  Expression e = create_expression(U);

  DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
  if (EI != expressionNumbering.end()) {
    valueNumbering.insert(std::make_pair(V, EI->second));
    return EI->second;
  } else {
    expressionNumbering.insert(std::make_pair(e, nextValueNumber));
    valueNumbering.insert(std::make_pair(V, nextValueNumber));

    return nextValueNumber++;
  }
} else if (InsertElementInst* U = dyn_cast<InsertElementInst>(V)) {
  Expression e = create_expression(U);

  DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
  if (EI != expressionNumbering.end()) {
    valueNumbering.insert(std::make_pair(V, EI->second));
    return EI->second;
  } else {
    expressionNumbering.insert(std::make_pair(e, nextValueNumber));
    valueNumbering.insert(std::make_pair(V, nextValueNumber));

    return nextValueNumber++;
  }
} else if (SelectInst* U = dyn_cast<SelectInst>(V)) {
  Expression e = create_expression(U);

  DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
  if (EI != expressionNumbering.end()) {
    valueNumbering.insert(std::make_pair(V, EI->second));
    return EI->second;
  } else {
    expressionNumbering.insert(std::make_pair(e, nextValueNumber));
    valueNumbering.insert(std::make_pair(V, nextValueNumber));
```

```
      return nextValueNumber++;
    }
  } else if (CastInst* U = dyn_cast<CastInst>(V)) {
    Expression e = create_expression(U);

    DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
    if (EI != expressionNumbering.end()) {
      valueNumbering.insert(std::make_pair(V, EI->second));
      return EI->second;
    } else {
      expressionNumbering.insert(std::make_pair(e, nextValueNumber));
      valueNumbering.insert(std::make_pair(V, nextValueNumber));

      return nextValueNumber++;
    }
  } else if (GetElementPtrInst* U = dyn_cast<GetElementPtrInst>(V)) {
    Expression e = create_expression(U);

    DenseMap<Expression, uint32_t>::iterator EI = expressionNumbering.find(e);
    if (EI != expressionNumbering.end()) {
      valueNumbering.insert(std::make_pair(V, EI->second));
      return EI->second;
    } else {
      expressionNumbering.insert(std::make_pair(e, nextValueNumber));
      valueNumbering.insert(std::make_pair(V, nextValueNumber));

      return nextValueNumber++;
    }
  } else {
    valueNumbering.insert(std::make_pair(V, nextValueNumber));
    return nextValueNumber++;
  }
}

/// lookup - Returns the value number of the specified value. Fails if
/// the value has not yet been numbered.
uint32_t ValueTable::lookup(Value* V) const {
  DenseMap<Value*, uint32_t>::iterator VI = valueNumbering.find(V);
  if (VI != valueNumbering.end())
    return VI->second;
  else
    assert(0 && "Value not numbered?");

  return 0;
}

/// clear - Remove all entries from the ValueTable
void ValueTable::clear() {
  valueNumbering.clear();
  expressionNumbering.clear();
  nextValueNumber = 1;
}
```

```
/// erase - Remove a value from the value numbering
void ValueTable::erase(Value* V) {
  valueNumbering.erase(V);
}

//===----------------------------------------------------------------------===//
//                         ValueNumberedSet Class
//===----------------------------------------------------------------------===//
namespace {
class ValueNumberedSet {
  private:
    SmallPtrSet<Value*, 8> contents;
    BitVector numbers;
  public:
    ValueNumberedSet() { numbers.resize(1); }
    ValueNumberedSet(const ValueNumberedSet& other) {
      numbers = other.numbers;
      contents = other.contents;
    }

    typedef SmallPtrSet<Value*, 8>::iterator iterator;

    iterator begin() { return contents.begin(); }
    iterator end() { return contents.end(); }

    bool insert(Value* v) { return contents.insert(v); }
    void insert(iterator I, iterator E) { contents.insert(I, E); }
    void erase(Value* v) { contents.erase(v); }
    unsigned count(Value* v) { return contents.count(v); }
    size_t size() { return contents.size(); }

    void set(unsigned i) {
      if (i >= numbers.size())
        numbers.resize(i+1);

      numbers.set(i);
    }

    void operator=(const ValueNumberedSet& other) {
      contents = other.contents;
      numbers = other.numbers;
    }

    void reset(unsigned i) {
      if (i < numbers.size())
        numbers.reset(i);
    }

    bool test(unsigned i) {
      if (i >= numbers.size())
        return false;

      return numbers.test(i);
```

```
    }

    void clear() {
      contents.clear();
      numbers.clear();
    }
};
}

//===--------------------------------------------------------------------===//
//                         GVN Pass
//===--------------------------------------------------------------------===//

namespace {

  class VISIBILITY_HIDDEN GVN : public FunctionPass {
    bool runOnFunction(Function &F);
  public:
    static char ID; // Pass identification, replacement for typeid
    GVN() : FunctionPass((intptr_t)&ID) { }

  private:
    ValueTable VN;

    DenseMap<BasicBlock*, ValueNumberedSet> availableOut;

    typedef DenseMap<Value*, SmallPtrSet<Instruction*, 4> > PhiMapType;
    PhiMapType phiMap;


    // This transformation requires dominator postdominator info
    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
      AU.setPreservesCFG();
      AU.addRequired<DominatorTree>();
      AU.addRequired<MemoryDependenceAnalysis>();
      AU.addRequired<AliasAnalysis>();
      AU.addRequired<TargetData>();
      AU.addPreserved<AliasAnalysis>();
      AU.addPreserved<MemoryDependenceAnalysis>();
      AU.addPreserved<TargetData>();
    }

    // Helper fuctions
    // FIXME: eliminate or document these better
    Value* find_leader(ValueNumberedSet& vals, uint32_t v) ;
    void val_insert(ValueNumberedSet& s, Value* v);
    bool processLoad(LoadInst* L,
                     DenseMap<Value*, LoadInst*>& lastLoad,
                     SmallVector<Instruction*, 4>& toErase);
    bool processInstruction(Instruction* I,
                            ValueNumberedSet& currAvail,
                            DenseMap<Value*, LoadInst*>& lastSeenLoad,
                            SmallVector<Instruction*, 4>& toErase);
```

```
    bool processNonLocalLoad(LoadInst* L,
                             SmallVector<Instruction*, 4>& toErase);
    bool processMemCpy(MemCpyInst* M, MemCpyInst* MDep,
                       SmallVector<Instruction*, 4>& toErase);
    bool performReturnSlotOptzn(MemCpyInst* cpy, CallInst* C,
                                SmallVector<Instruction*, 4>& toErase);
    Value *GetValueForBlock(BasicBlock *BB, LoadInst* orig,
                            DenseMap<BasicBlock*, Value*> &Phis,
                            bool top_level = false);
    void dump(DenseMap<BasicBlock*, Value*>& d);
    bool iterateOnFunction(Function &F);
    Value* CollapsePhi(PHINode* p);
    bool isSafeReplacement(PHINode* p, Instruction* inst);
    bool valueHasOnlyOneUseAfter(Value* val, MemCpyInst* use,
                                 Instruction* cutoff);
  };

  char GVN::ID = 0;

}

// createGVNPass - The public interface to this file...
FunctionPass *llvm::createGVNPass() { return new GVN(); }

static RegisterPass<GVN> X("gvn",
                           "Global Value Numbering");

STATISTIC(NumGVNInstr, "Number of instructions deleted");
STATISTIC(NumGVNLoad, "Number of loads deleted");

/// find_leader - Given a set and a value number, return the first
/// element of the set with that value number, or 0 if no such element
/// is present
Value* GVN::find_leader(ValueNumberedSet& vals, uint32_t v) {
  if (!vals.test(v))
    return 0;

  for (ValueNumberedSet::iterator I = vals.begin(), E = vals.end();
       I != E; ++I)
    if (v == VN.lookup(*I))
      return *I;

  assert(0 && "No leader found, but present bit is set?");
  return 0;
}

/// val_insert - Insert a value into a set only if there is not a value
/// with the same value number already in the set
void GVN::val_insert(ValueNumberedSet& s, Value* v) {
  uint32_t num = VN.lookup(v);
  if (!s.test(num))
    s.insert(v);
}
```

```
void GVN::dump(DenseMap<BasicBlock*, Value*>& d) {
  printf("{\n");
  for (DenseMap<BasicBlock*, Value*>::iterator I = d.begin(),
       E = d.end(); I != E; ++I) {
    if (I->second == MemoryDependenceAnalysis::None)
      printf("None\n");
    else
      I->second->dump();
  }
  printf("}\n");
}

Value* GVN::CollapsePhi(PHINode* p) {
  DominatorTree &DT = getAnalysis<DominatorTree>();
  Value* constVal = p->hasConstantValue();

  if (constVal) {
    if (Instruction* inst = dyn_cast<Instruction>(constVal)) {
      if (DT.dominates(inst, p))
        if (isSafeReplacement(p, inst))
          return inst;
    } else {
      return constVal;
    }
  }

  return 0;
}

bool GVN::isSafeReplacement(PHINode* p, Instruction* inst) {
  if (!isa<PHINode>(inst))
    return true;

  for (Instruction::use_iterator UI = p->use_begin(), E = p->use_end();
       UI != E; ++UI)
    if (PHINode* use_phi = dyn_cast<PHINode>(UI))
      if (use_phi->getParent() == inst->getParent())
        return false;

  return true;
}

/// GetValueForBlock - Get the value to use within the specified basic block.
/// available values are in Phis.
Value *GVN::GetValueForBlock(BasicBlock *BB, LoadInst* orig,
                             DenseMap<BasicBlock*, Value*> &Phis,
                             bool top_level) {

  // If we have already computed this value, return the previously computed val.
  DenseMap<BasicBlock*, Value*>::iterator V = Phis.find(BB);
  if (V != Phis.end() && !top_level) return V->second;
```

```
  BasicBlock* singlePred = BB->getSinglePredecessor();
  if (singlePred) {
    Value *ret = GetValueForBlock(singlePred, orig, Phis);
    Phis[BB] = ret;
    return ret;
  }
  // Otherwise, the idom is the loop, so we need to insert a PHI node. Do so
  // now, then get values to fill in the incoming values for the PHI.
  PHINode *PN = new PHINode(orig->getType(), orig->getName()+".rle",
                            BB->begin());
  PN->reserveOperandSpace(std::distance(pred_begin(BB), pred_end(BB)));

  if (Phis.count(BB) == 0)
    Phis.insert(std::make_pair(BB, PN));

  // Fill in the incoming values for the block.
  for (pred_iterator PI = pred_begin(BB), E = pred_end(BB); PI != E; ++PI) {
    Value* val = GetValueForBlock(*PI, orig, Phis);

    PN->addIncoming(val, *PI);
  }
  AliasAnalysis& AA = getAnalysis<AliasAnalysis>();
  AA.copyValue(orig, PN);

  // Attempt to collapse PHI nodes that are trivially redundant
  Value* v = CollapsePhi(PN);
  if (v) {
    MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();

    MD.removeInstruction(PN);
    PN->replaceAllUsesWith(v);

    for (DenseMap<BasicBlock*, Value*>::iterator I = Phis.begin(),
         E = Phis.end(); I != E; ++I)
      if (I->second == PN)
        I->second = v;

    PN->eraseFromParent();

    Phis[BB] = v;

    return v;
  }

  // Cache our phi construction results
  phiMap[orig->getPointerOperand()].insert(PN);
  return PN;
}

/// processNonLocalLoad - Attempt to eliminate a load whose dependencies are
/// non-local by performing PHI construction.
bool GVN::processNonLocalLoad(LoadInst* L,
                              SmallVector<Instruction*, 4>& toErase) {
```

```
MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();

// Find the non-local dependencies of the load
DenseMap<BasicBlock*, Value*> deps;
MD.getNonLocalDependency(L, deps);

DenseMap<BasicBlock*, Value*> repl;

// Filter out useless results (non-locals, etc)
for (DenseMap<BasicBlock*, Value*>::iterator I = deps.begin(), E = deps.end();
     I != E; ++I)
  if (I->second == MemoryDependenceAnalysis::None) {
    return false;
  } else if (I->second == MemoryDependenceAnalysis::NonLocal) {
    continue;
  } else if (StoreInst* S = dyn_cast<StoreInst>(I->second)) {
    if (S->getPointerOperand() == L->getPointerOperand())
      repl[I->first] = S->getOperand(0);
    else
      return false;
  } else if (LoadInst* LD = dyn_cast<LoadInst>(I->second)) {
    if (LD->getPointerOperand() == L->getPointerOperand())
      repl[I->first] = LD;
    else
      return false;
  } else {
    return false;
  }

// Use cached PHI construction information from previous runs
SmallPtrSet<Instruction*, 4>& p = phiMap[L->getPointerOperand()];
for (SmallPtrSet<Instruction*, 4>::iterator I = p.begin(), E = p.end();
     I != E; ++I) {
  if ((*I)->getParent() == L->getParent()) {
    MD.removeInstruction(L);
    L->replaceAllUsesWith(*I);
    toErase.push_back(L);
    NumGVNLoad++;

    return true;
  } else {
    repl.insert(std::make_pair((*I)->getParent(), *I));
  }
}

// Perform PHI construction
SmallPtrSet<BasicBlock*, 4> visited;
Value* v = GetValueForBlock(L->getParent(), L, repl, true);

MD.removeInstruction(L);
L->replaceAllUsesWith(v);
toErase.push_back(L);
NumGVNLoad++;
```

```
  return true;
}

/// processLoad - Attempt to eliminate a load, first by eliminating it
/// locally, and then attempting non-local elimination if that fails.
bool GVN::processLoad(LoadInst* L,
                      DenseMap<Value*, LoadInst*>& lastLoad,
                      SmallVector<Instruction*, 4>& toErase) {
  if (L->isVolatile()) {
    lastLoad[L->getPointerOperand()] = L;
    return false;
  }

  Value* pointer = L->getPointerOperand();
  LoadInst*& last = lastLoad[pointer];

  // ... to a pointer that has been loaded from before...
  MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();
  bool removedNonLocal = false;
  Instruction* dep = MD.getDependency(L);
  if (dep == MemoryDependenceAnalysis::NonLocal &&
      L->getParent() != &L->getParent()->getParent()->getEntryBlock()) {
    removedNonLocal = processNonLocalLoad(L, toErase);

    if (!removedNonLocal)
      last = L;

    return removedNonLocal;
  }


  bool deletedLoad = false;

  // Walk up the dependency chain until we either find
  // a dependency we can use, or we can't walk any further
  while (dep != MemoryDependenceAnalysis::None &&
         dep != MemoryDependenceAnalysis::NonLocal &&
         (isa<LoadInst>(dep) || isa<StoreInst>(dep))) {
    // ... that depends on a store ...
    if (StoreInst* S = dyn_cast<StoreInst>(dep)) {
      if (S->getPointerOperand() == pointer) {
        // Remove it!
        MD.removeInstruction(L);

        L->replaceAllUsesWith(S->getOperand(0));
        toErase.push_back(L);
        deletedLoad = true;
        NumGVNLoad++;
      }

      // Whether we removed it or not, we can't
      // go any further
```

```
      break;
    } else if (!last) {
      // If we don't depend on a store, and we haven't
      // been loaded before, bail.
      break;
    } else if (dep == last) {
      // Remove it!
      MD.removeInstruction(L);

      L->replaceAllUsesWith(last);
      toErase.push_back(L);
      deletedLoad = true;
      NumGVNLoad++;

      break;
    } else {
      dep = MD.getDependency(L, dep);
    }
  }

  if (dep != MemoryDependenceAnalysis::None &&
      dep != MemoryDependenceAnalysis::NonLocal &&
      isa<AllocationInst>(dep)) {
    // Check that this load is actually from the
    // allocation we found
    Value* v = L->getOperand(0);
    while (true) {
      if (BitCastInst *BC = dyn_cast<BitCastInst>(v))
        v = BC->getOperand(0);
      else if (GetElementPtrInst *GEP = dyn_cast<GetElementPtrInst>(v))
        v = GEP->getOperand(0);
      else
        break;
    }
    if (v == dep) {
      // If this load depends directly on an allocation, there isn't
      // anything stored there; therefore, we can optimize this load
      // to undef.
      MD.removeInstruction(L);

      L->replaceAllUsesWith(UndefValue::get(L->getType()));
      toErase.push_back(L);
      deletedLoad = true;
      NumGVNLoad++;
    }
  }

  if (!deletedLoad)
    last = L;

  return deletedLoad;
}
```

```
/// valueHasOnlyOneUse - Returns true if a value has only one use after the
/// cutoff that is in the current same block and is the same as the use
/// parameter.
bool GVN::valueHasOnlyOneUseAfter(Value* val, MemCpyInst* use,
                                  Instruction* cutoff) {
  DominatorTree& DT = getAnalysis<DominatorTree>();

  SmallVector<User*, 8> useList(val->use_begin(), val->use_end());
  while (!useList.empty()) {
    User* UI = useList.back();


    if (isa<GetElementPtrInst>(UI) || isa<BitCastInst>(UI)) {
      useList.pop_back();
      for (User::use_iterator I = UI->use_begin(), E = UI->use_end();
           I != E; ++I)
        useList.push_back(*I);
    } else if (UI == use) {
      useList.pop_back();
    } else if (Instruction* inst = dyn_cast<Instruction>(UI)) {
      if (inst->getParent() == use->getParent() &&
          (inst == cutoff || !DT.dominates(cutoff, inst))) {
        useList.pop_back();
      } else
        return false;
    } else
      return false;
  }

  return true;
}

/// performReturnSlotOptzn - takes a memcpy and a call that it depends on,
/// and checks for the possibility of a return slot optimization by having
/// the call write its result directly into the callees return parameter
/// rather than using memcpy
bool GVN::performReturnSlotOptzn(MemCpyInst* cpy, CallInst* C,
                                 SmallVector<Instruction*, 4>& toErase) {
  // Deliberately get the source and destination with bitcasts stripped away,
  // because we'll need to do type comparisons based on the underlying type.
  Value* cpyDest = cpy->getDest();
  Value* cpySrc = cpy->getSource();
  CallSite CS = CallSite::get(C);

  // Since this is a return slot optimization, we need to make sure that
  // the value being copied is, in fact, in a return slot.    We also need to
  // check that the return slot parameter is marked noalias, so that we can
  // be sure that changing it will not cause unexpected behavior changes due
  // to it being accessed through a global or another parameter.
  if (CS.arg_size() == 0 ||
      cpySrc != CS.getArgument(0) ||
      !CS.paramHasAttr(1, ParamAttr::NoAlias | ParamAttr::StructRet))
    return false;
```

```
// Since we're changing the parameter to the callsite, we need to make sure
// that what would be the new parameter dominates the callsite.
DominatorTree& DT = getAnalysis<DominatorTree>();
if (Instruction* cpyDestInst = dyn_cast<Instruction>(cpyDest))
  if (!DT.dominates(cpyDestInst, C))
    return false;

// Check that something sneaky is not happening involving casting
// return slot types around.
if (CS.getArgument(0)->getType() != cpyDest->getType())
  return false;
// sret -> pointer
const PointerType* PT = cast<PointerType>(cpyDest->getType());

// We can only perform the transformation if the size of the memcpy
// is constant and equal to the size of the structure.
ConstantInt* cpyLength = dyn_cast<ConstantInt>(cpy->getLength());
if (!cpyLength)
  return false;

TargetData& TD = getAnalysis<TargetData>();
if (TD.getTypeStoreSize(PT->getElementType()) != cpyLength->getZExtValue())
  return false;

// For safety, we must ensure that the output parameter of the call only has
// a single use, the memcpy. Otherwise this can introduce an invalid
// transformation.
if (!valueHasOnlyOneUseAfter(CS.getArgument(0), cpy, C))
  return false;

// We only perform the transformation if it will be profitable.
if (!valueHasOnlyOneUseAfter(cpyDest, cpy, C))
  return false;

// In addition to knowing that the call does not access the return slot
// in some unexpected manner, which we derive from the noalias attribute,
// we also need to know that it does not sneakily modify the destination
// slot in the caller.    We don't have parameter attributes to go by
// for this one, so we just rely on AA to figure it out for us.
AliasAnalysis& AA = getAnalysis<AliasAnalysis>();
if (AA.getModRefInfo(C, cpy->getRawDest(), cpyLength->getZExtValue()) !=
    AliasAnalysis::NoModRef)
  return false;

// If all the checks have passed, then we're alright to do the transformation.
CS.setArgument(0, cpyDest);

// Drop any cached information about the call, because we may have changed
// its dependence information by changing its parameter.
MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();
MD.dropInstruction(C);
```

```
  // Remove the memcpy
  MD.removeInstruction(cpy);
  toErase.push_back(cpy);

  return true;
}

/// processMemCpy - perform simplication of memcpy's. If we have memcpy A which
/// copies X to Y, and memcpy B which copies Y to Z, then we can rewrite B to be
/// a memcpy from X to Z (or potentially a memmove, depending on circumstances).
///   This allows later passes to remove the first memcpy altogether.
bool GVN::processMemCpy(MemCpyInst* M, MemCpyInst* MDep,
                        SmallVector<Instruction*, 4>& toErase) {
  // We can only transforms memcpy's where the dest of one is the source of the
  // other
  if (M->getSource() != MDep->getDest())
    return false;

  // Second, the length of the memcpy's must be the same, or the preceeding one
  // must be larger than the following one.
  ConstantInt* C1 = dyn_cast<ConstantInt>(MDep->getLength());
  ConstantInt* C2 = dyn_cast<ConstantInt>(M->getLength());
  if (!C1 || !C2)
    return false;

  uint64_t DepSize = C1->getValue().getZExtValue();
  uint64_t CpySize = C2->getValue().getZExtValue();

  if (DepSize < CpySize)
    return false;

  // Finally, we have to make sure that the dest of the second does not
  // alias the source of the first
  AliasAnalysis& AA = getAnalysis<AliasAnalysis>();
  if (AA.alias(M->getRawDest(), CpySize, MDep->getRawSource(), DepSize) !=
      AliasAnalysis::NoAlias)
    return false;
  else if (AA.alias(M->getRawDest(), CpySize, M->getRawSource(), CpySize) !=
           AliasAnalysis::NoAlias)
    return false;
  else if (AA.alias(MDep->getRawDest(), DepSize, MDep->getRawSource(), DepSize)
           != AliasAnalysis::NoAlias)
    return false;

  // If all checks passed, then we can transform these memcpy's
  Function* MemCpyFun = Intrinsic::getDeclaration(
                                 M->getParent()->getParent()->getParent(),
                                 M->getIntrinsicID());

  std::vector<Value*> args;
  args.push_back(M->getRawDest());
  args.push_back(MDep->getRawSource());
  args.push_back(M->getLength());
```

```
  args.push_back(M->getAlignment());

  CallInst* C = new CallInst(MemCpyFun, args.begin(), args.end(), "", M);

  MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();
  if (MD.getDependency(C) == MDep) {
    MD.dropInstruction(M);
    toErase.push_back(M);
    return true;
  } else {
    MD.removeInstruction(C);
    toErase.push_back(C);
    return false;
  }
}

/// processInstruction - When calculating availability, handle an instruction
/// by inserting it into the appropriate sets
bool GVN::processInstruction(Instruction* I,
                                ValueNumberedSet& currAvail,
                                DenseMap<Value*, LoadInst*>& lastSeenLoad,
                                SmallVector<Instruction*, 4>& toErase) {
  if (LoadInst* L = dyn_cast<LoadInst>(I)) {
    return processLoad(L, lastSeenLoad, toErase);
  } else if (MemCpyInst* M = dyn_cast<MemCpyInst>(I)) {
    MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();

    // The are two possible optimizations we can do for memcpy:
    //   a) memcpy-memcpy xform which exposes redundance for DSE
    //   b) call-memcpy xform for sret return slot optimization
    Instruction* dep = MD.getDependency(M);
    if (dep == MemoryDependenceAnalysis::None ||
        dep == MemoryDependenceAnalysis::NonLocal)
      return false;
    if (MemCpyInst *MemCpy = dyn_cast<MemCpyInst>(dep))
      return processMemCpy(M, MemCpy, toErase);
    if (CallInst* C = dyn_cast<CallInst>(dep))
      return performReturnSlotOptzn(M, C, toErase);
    return false;
  }

  unsigned num = VN.lookup_or_add(I);

  // Collapse PHI nodes
  if (PHINode* p = dyn_cast<PHINode>(I)) {
    Value* constVal = CollapsePhi(p);

    if (constVal) {
      for (PhiMapType::iterator PI = phiMap.begin(), PE = phiMap.end();
           PI != PE; ++PI)
        if (PI->second.count(p))
          PI->second.erase(p);
```

```
      p->replaceAllUsesWith(constVal);
      toErase.push_back(p);
    }
  // Perform value-number based elimination
  } else if (currAvail.test(num)) {
    Value* repl = find_leader(currAvail, num);

    if (CallInst* CI = dyn_cast<CallInst>(I)) {
      AliasAnalysis& AA = getAnalysis<AliasAnalysis>();
      if (!AA.doesNotAccessMemory(CI)) {
        MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();
        if (cast<Instruction>(repl)->getParent() != CI->getParent() ||
            MD.getDependency(CI) != MD.getDependency(cast<CallInst>(repl))) {
          // There must be an intervening may-alias store, so nothing from
          // this point on will be able to be replaced with the preceding call
          currAvail.erase(repl);
          currAvail.insert(I);

          return false;
        }
      }
    }

    // Remove it!
    MemoryDependenceAnalysis& MD = getAnalysis<MemoryDependenceAnalysis>();
    MD.removeInstruction(I);

    VN.erase(I);
    I->replaceAllUsesWith(repl);
    toErase.push_back(I);
    return true;
  } else if (!I->isTerminator()) {
    currAvail.set(num);
    currAvail.insert(I);
  }

  return false;
}

// GVN::runOnFunction - This is the main transformation entry point for a
// function.
//
bool GVN::runOnFunction(Function& F) {
  VN.setAliasAnalysis(&getAnalysis<AliasAnalysis>());

  bool changed = false;
  bool shouldContinue = true;

  while (shouldContinue) {
    shouldContinue = iterateOnFunction(F);
    changed |= shouldContinue;
  }
```

```
  return changed;
}


// GVN::iterateOnFunction - Executes one iteration of GVN
bool GVN::iterateOnFunction(Function &F) {
  // Clean out global sets from any previous functions
  VN.clear();
  availableOut.clear();
  phiMap.clear();

  bool changed_function = false;

  DominatorTree &DT = getAnalysis<DominatorTree>();

  SmallVector<Instruction*, 4> toErase;

  // Top-down walk of the dominator tree
  for (df_iterator<DomTreeNode*> DI = df_begin(DT.getRootNode()),
         E = df_end(DT.getRootNode()); DI != E; ++DI) {

    // Get the set to update for this block
    ValueNumberedSet& currAvail = availableOut[DI->getBlock()];
    DenseMap<Value*, LoadInst*> lastSeenLoad;

    BasicBlock* BB = DI->getBlock();

    // A block inherits AVAIL_OUT from its dominator
    if (DI->getIDom() != 0)
      currAvail = availableOut[DI->getIDom()->getBlock()];

    for (BasicBlock::iterator BI = BB->begin(), BE = BB->end();
         BI != BE; ) {
      changed_function |= processInstruction(BI, currAvail,
                                             lastSeenLoad, toErase);

      NumGVNInstr += toErase.size();

      // Avoid iterator invalidation
      ++BI;

      for (SmallVector<Instruction*, 4>::iterator I = toErase.begin(),
           E = toErase.end(); I != E; ++I) {
        (*I)->eraseFromParent();
      }

      toErase.clear();
    }
  }

  return changed_function;
}
```