This talk is divided into 7 parts. Part 1 begins with a quick look at the SSA optimization framework and global value numbering. Part 2 describes a well known brute force algorithm for GVN. Part 3 modifies this to arrive at an efficient sparse algorithm. Part 4 unifies the sparse algorithm with a wide range of additional analyses. Parts 5-7 put it all together, present measurements and suggest conclusions.

The framework of an SSA based global optimizer can be represented as a pipeline of processing stages. The intermediate representation of a routine flows in at the top and is first translated to static single assignment form. The SSA form IR is now processed by a number of passes, one of which is global value numbering. The results of GVN are used to transform the IR. Finally the IR is translated out of SSA form, and optimized IR flows out of the pipeline.

These are the basic notions of value numbering. A value is a constant or an SSA variable. The values of a routine can be partitioned into congruence classes. Congruent values are guaranteed to be identical for any possible execution of the routine. Every congruence class has a representative value called a leader.

GVN is an analysis phase - it does not transform IR. Its input is the SSA form IR of a routine. It produces 4 outputs: the congruence classes of the routine, the values in every congruence class, the leader of every congruence class and the congruence class of every value.

GVN can be unified with analyses such as constant folding, algebraic simplification and unreachable code elimination. Unified algorithms produce better results and run faster than repeated invocations of their component analyses. The results of GVN are used to perform unreachable code elimination, constant propagation, copy propagation and redundancy elimination.

This is a well known brute force algorithm for GVN. Step 1. initially makes all the SSA variables of the routine have the indeterminate value $\top$. Step 2. clears a hash table to map expressions to values. Step 3. processes all the instructions of the routine in reverse post order. Only instructions that assign expressions to variables are considered. For an instruction which assigns the expression X op Y to V, let E be the expression Value-of(X) op Value-of(Y). A hash table lookup is performed on E. If the lookup is successful, its result is made the value of V. Otherwise the value of V is set to V itself, and the hash table is updated to map E onto V. Finally, step 4. repeats steps 2. and 3. until there are no more changes in the values.

Here is an example of the brute force algorithm in action. First all the variables are given the value $\top$. Then their definitions are processed in RPO.

The SSA variable I_1 is assigned the value 1. Since the expression 1 is not in the hash table, the value of I_1 is set to I_1. For J_1, the hash table already contains 1, so its value is also set to I_1.

I_2 is assigned the merge of the values of I_1 and I_3. I_3 has the value $\top$, so it is ignored. A ϕ-function with one argument reduces to that argument. So I_2 is assigned the value I_1. Similarly J_2 is also assigned the value I_1.

I_3 is defined as $I_2 + 1$, which becomes $I_1 + 1$, which is not in the hash table. So I_3 is assigned the value I_3. Since J_3 has the same definition, it is also assigned the same value. This ends the first pass.

The second pass does not change I_1 and J_1. The definition of I_2 is now $\phi(I_1, I_3)$, which is not in the hash table. So I_2 changes to the value I_2. J_2 also changes to I_2. I_3 and J_3 remain the same.

The third pass confirms these values. Hence J_3 is congruent to I_3. The algorithm has reached a fixed point in 3 passes.

The Brute Force algorithm is Taylor Simpsons 1996 hash based RPO algorithm. It achieves the same result as the 1988 partitioning algorithm due to Alpern, Wegman and Zadeck. Brute Force is an optimistic algorithm - it assumes that all values are initially congruent until it can prove otherwise. Only an optimistic algorithm can discover the congruence of I_3 and J_3 in the previous example. Brute Force takes $O(C)$ passes where C is the loop connectedness of the SSA def-use graph. The loop connectedness of a graph is the maximum number of back edges in any acyclic path.

Sparse value numbering is a simple modification to Brute Force. Step 1. performs the same initialization as Brute Force. Step 2. touches the instructions of the start block. Step 3. examines all the instructions of the routine in reverse post order. If an instruction is untouched, it is skipped. Otherwise it is wiped, and processed as in Brute Force. If the value it produces has changed, its consumers are touched. These are found from the SSA def-use chains of the routine. Finally, step 4. repeats step 3. until there are no more touched instructions.

Here is an example of sparse value numbering in action. Since it processes instructions in exactly the same way as Brute Force, the values computed by every pass are exactly the same as for Brute Force. Initially, the definitions of I_1 and J_1 are touched. These are processed, and their values change, so their consumers, the definitions of I_2 and J_2, are touched and processed. I_2 and J_2 also change, so the definitions of I_3 and J_3 are touched and processed. I_3 and J_3 also change, so the definitions of I_2 and J_2 are touched. Hence the first pass ends after processing 6 instructions, leaving the definitions of I_2 and J_2 touched.

The second pass skips the definitions of I_1 and J_1, and processes the definitions of I_2 and J_2. The change in I_2 and J_2 causes the definitions of I_3 and J_3 to be touched and processed. The change in I_3 and J_3 causes the definitions of I_2 and J_2 to be touched. So the second pass ends after processing 4 instructions, leaving the definitions of I_2 and J_2 touched.

The third pass processes only 2 instructions - the definitions of I_2 and J_2. Since they do not change, it does not touch any more instructions, and the algorithm terminates after having reached a fixed point.

Brute Force processed a total of 3 x 6 or 18 instructions. Sparse value numbering processed a total of 6 + 4 + 2 or 12 instructions. So it is ≈ 1.5 times faster than Brute Force for this example.

Sparse value numbering is faster than Brute Force because it does not process all the instructions in every pass. It does have to examine every instruction to see if it is touched, but this check is much faster than processing it. Unlike Brute Force, the sparse algorithm does not clear the hash table between passes. Because of this, when the leader of a congruence class is moved to a new congruence class, the old congruence class requires special treatment (unless it is empty). The definitions of the remaining members of the old class are touched, and one of them is chosen to be their new leader.

The previous example demonstrated that at the end of a pass of sparse value numbering, the instructions left touched are ϕ-instructions with one or more operands carried by back edges. For acyclic code, there are no such instructions, therefore sparse value numbering completes in one pass. For cyclic code, it must process these instructions. When the optimistic assumption is confirmed, the values of these instructions will not change, therefore sparse value numbering completes in almost one pass. When the optimistic assumption is rejected, sparse value numbering takes anywhere up to one less pass than Brute Force. Therefore sparse value numbering should be quite efficient. This is supported by measurements collected from our implementation of sparse value numbering when optimizing the SPEC CINT2000 C benchmarks. It takes $< 4\%$ of the total optimization time (this is with additional unified analyses), and 1.98 passes per routine on average. Finally, the speedup due to sparseness is 1.23 to 1.57.

Sparse value numbering can be strengthened by unifying it with a wide range of additional analyses. Algebraic transformations are the most straightforward to incorporate. Before looking up an expression in the hash table, it is subjected to algebraic transformations. Constant folding evaluates expressions with constant operands. The algebraic, associative and distributive laws are used to simplify expressions or restructure them in order to find more congruences. This requires forward propagation, which replaces an SSA variable with its defining expression. These unifications require two modifications to sparse value numbering. Firstly, if any value of a congruence class is defined as a constant, that constant should be made the leader of the class. Secondly, a value which is subject to forward propagation should be treated as changed when its defining expression changes.

This is an application of algebraic transformations. The first pass of sparse value numbering sets the value of I_1 to 1. Because I_3 is $\top$, I_2 reduces to 1. Constant folding evaluates I_3 to 1. The second pass processes the definition of I_2. Both I_1 and I_3 have the value 1, so I_2 becomes their merge, which is 1. Since there is no change in I_3, the algorithm terminates. It has taken almost one pass to find out that I_3 has the value 1.

Unreachable code elimination is also straightforward to incorporate into sparse value numbering. Firstly, the start block of the routine is made reachable, and all the other blocks and edges are made unreachable. Secondly, touched but unreachable instructions are wiped but not processed. Thirdly, jump instructions are also examined. If an outedge cannot be followed, it remains unreachable. Otherwise it becomes and remains reachable. Once an edge becomes reachable, so do its target blocks. Finally, operands of ϕ-functions that are carried by unreachable edges are ignored.

This is an application of unreachable code elimination. Constant folding evaluates the predicate $I_1 \neq 0$ to true. So the edges E_1 and E_2 remain unreachable. Thus I_2 is ignored when evaluating the definition of I_3. Hence I_3 has the value 1.

The optimistic assumption is powerful but also expensive. A better tradeoff is the balanced assumption, which is pessimistic in congruence of values and optimistic in reachability. That is, it assumes that all values are non-congruent until proven otherwise, and all blocks (other than the start block) and edges are unreachable until proven otherwise. To perform balanced value numbering, every ϕ-function with one or more operands carried by back edges is treated as a unique value, and the algorithm is terminated after the first pass. Measurements collected when optimizing the SPEC CINT2000 C benchmarks indicate that balanced value numbering is as fast as pessimistic, and almost as strong as optimistic value numbering. It also runs 1.39 to 1.90 times faster than optimistic value numbering.

This example illustrates that values can be inferred in the absence of explicit definitions. The use of I_1 in block B_1 is dominated by edge E_1. E_1 is the false outedge from the conditional jump. So the predicate $J_1 \neq 0$ has the value false at edge E_1. Thus J_1 must have the value 0 at edge E_1 and block B_1. Furthermore I_1 is congruent to J_1. Thus I_1 must have the value 0 at edge E_1 and block B_1. Hence K_1 has the value 0.

The algorithm for value inference is straightforward to incorporate into sparse value numbering. Before looking up an expression in the hash table, value inference is performed on its operands. For each operand X of the expression, value inference starts from the block B containing the expression and goes up the dominator tree looking for an edge E such that E dominates B, E is the true outedge from a jump instruction with predicate $Y = Z$, and Y is congruent to X. If such an E is found, then X is inferred to have the value Z. Only a dominator tree approach can be completely unified with value numbering. The previous example cannot be handled by approaches that implement value inference by means of a pre-pass that inserts assignments.

When performing value inference, there are two approaches to determining dominance relationships. Accordingly there are two versions of the value numbering algorithm. The complete algorithm incrementally builds the reachable dominator tree - the dominator tree of the reachable subset of the CFG. The practical algorithm uses the dominator tree of the routine. The practical algorithm is simpler, but it cannot ignore unreachable code when determining dominators, and it cannot perform inferences along back edges. Furthermore sparse value numbering has to be modified to work for value inference. When the reachability or predicate of an edge $B_1 \rightarrow B_2$ changes, all potentially affected instructions are touched. The complete algorithm touches the instructions of all the blocks dominated by block B_2. The practical algorithm touches all instructions downstream in RPO of block B_2.

When value inference finds that one variable has the value of another, it can be repeated on the second variable. This makes its worst case time $O(E^2)$, where E is the number of edges in the CFG. However it can be made to run very efficiently by noting that it is applicable only to the operands of $=$ or $\neq$ predicates of jump instructions. These are initially marked as inferenceable, and a count of the inferenceable values is maintained for every congruence class. When a class acquires an inferenceable value, its count is incremented, and when it loses one, it is decremented. Value inference can now be restricted to values in classes whose counts are positive. Additionally, the results of value inference can be cached across multiple uses in a block. These optimizations make value inference run very efficiently in practice. Measurements collected when optimizing the SPEC CINT2000 C benchmarks indicate that it visits 0.91 blocks per instruction on average.

Predicate inference is similar to value inference in concept and implementation. In this example, the predicate $J_1 = 0$ in block B_1 is dominated by edge E_1. The predicate $I_1 \neq 0$ has the value false at edge E_1. Moreover I_1 is congruent to J_1. Hence the predicate $J_1 = 0$ has the value true in block B_1. Predicate inference is also very efficient in practice. Measurements collected when optimizing the SPEC CINT2000 C benchmarks indicate that it visits 0.38 blocks per instruction on average.

It is not possible for value numbering to discover congruences between ϕ-functions in different blocks. This is because the ϕ-functions of classical SSA do not associate their operands with the conditions under which they arrive. Φ-predication overcomes this limitation by associating each operand of a ϕ-function with a predicate which is true when and only when control flow carries that operand to the ϕ-function.

The slide shows two ϕ-functions I_0 and I'_0, which are in different blocks. The problem is: when are I_0 and I'_0 congruent? This is answered by rewriting I_0 as: if P_1 then I_1 else if P_2 then I_2 else if .... The predicate P_1 is true when and only when control reaches block B_1 through E_1. All paths to block B_1 must go through its immediate dominator D_1. Hence it is sufficient for P_1 to be true when and only when control reaches B_1 along the path $D_1 \rightarrow \cdots \rightarrow E_1 \rightarrow B_1$. Similarly I'_0 is rewritten as: if P'_1 then I'_1 else if P'_2 then I'_2 else if .... Now, I_0 is congruent to I'_0 if and only if the I_i are congruent to the I'_i and the P_j are congruent to the P'_j.

It is convenient to define the predicate of a block B_1 as: $P_1 \vee P_2 \vee \ldots$. Now two ϕ-functions are congruent if their arguments are congruent and either their blocks are identical, or the predicates of their blocks are congruent. The predicate of a block should not be optimized to the value true; although it has the form of an expression, it is really an ordered list of predicates. The algorithm to compute the predicate of a block B_1 first determines its immediate dominator D_1. It then traverses all reachable paths from block D_1 to block B_1, combining the predicates of jump instructions encountered during traversal. This algorithm imposes two restrictions: block B_1 must postdominate block D_1, and back edges can not be traversed. If these restrictions are violated, the predicate of block B_1 is undefined and non-congruent to the predicate of any other block.

Here is an example of ϕ-predication in action. Blocks B_4 and B_7 contain ϕ-functions whose arguments are congruent. The predicate of block B_4 is computed by starting from its immediate dominator B_1, and traversing the two reachable paths from block B_1 to block B_4. The first path $B_1 \rightarrow B_2 \rightarrow B_4$ reaches B_4 with the predicate $K_1 \neq 0$. The second path $B_1 \rightarrow B_2 \rightarrow B_4$ reaches B_4 with the predicate $K_1 = 0$. Hence the predicate of block B_4 is $(K_1 \neq 0) \vee (K_1 = 0)$. The predicate of block B_7 is identical. Hence J_3 is congruent to I_3.

To incorporate ϕ-predication into sparse value numbering, it is necessary to compute the predicates of touched blocks only. The predicates of blocks are computed before their instructions are processed. When the reachability or predicate of an edge $B_1 \rightarrow B_2$ changes, all potentially affected blocks are touched. The complete algorithm touches all blocks that postdominate block B_2. The practical algorithm touches all blocks that are downstream in RPO of block B_2. Φ-predication is very efficient in practice. Measurements collected when optimizing the SPEC CINT2000 C benchmarks indicate that it visits 0.38 blocks per instruction on average.

Combining all these analyses produces an algorithm that unifies sparse value numbering with constant folding, algebraic simplification, unreachable code elimination, global reassociation, value inference, predicate inference, and ϕ-predication. For balanced value numbering, the worst case time complexity of the unified algorithm is $O(E^2(E+I))$, where E is the number of edges in the CFG and I is the number of instructions in the routine. For optimistic value numbering on an acyclic CFG, it is the same. For optimistic value numbering on an cyclic CFG, it is $O(CE^2(E+I))$, where C is the loop connectedness of the SSA def-use graph. In spite of these worst case time complexities, the unified algorithm is quite efficient in practice. Measurements collected when optimizing the SPEC CINT2000 C benchmarks indicate that it takes $< 4\%$ of the total optimization time.

We have implemented the practical version of the unified algorithm in the SSA based high level interprocedural optimizer component of an internal version of the PA-RISC C compiler on HP-UX. Our implementation does not exploit the distributive law, but it is strong enough to handle the previous example. We now present measurements of the efficiency and strength of the unified algorithm collected when optimizing the SPEC CINT2000 C benchmarks. The unified algorithm takes $< 4\%$ of the total optimization time. So it is quite efficient in practice. It runs 1.23 to 1.57 times faster when sparseness is enabled, and 1.15 to 1.32 times faster when global reassociation, value inference, predicate inference and ϕ-predication are disabled. So the speedup due to sparseness more than offsets the cost of incorporating these additional analyses. It runs 1.39 to 1.90 times faster with balanced value numbering. So we can makes the most efficient use of compile time by disabling the additional analyses and performing balanced value numbering on infrequently executed routines. Finally, it takes 1.98 passes per routine on average, and the average instructions visits 0.91, 0.38 and 0.16 blocks for value inference, predicate inference and ϕ-predication respectively. This shows that both sparse value numbering and the additional analyses can be implemented very efficiently.

This is a comparison of the unified algorithm with Cliff Click's strongest algorithm - a unification of optimistic value numbering, constant folding, algebraic simplification and unreachable code elimination. This is the $O(N^2)$ algorithm from his thesis; it is not the weaker algorithm given in his PLDI'95 paper. We emulated Click's algorithm by disabling all the analyses except the four above. The unified algorithm finds 1 to 100 more unreachable values in 0.6% of the routines, 1 to 102 more unreachable and constant values in 4.4% of the routines, and 1 to 88 less congruence classes in 4.7% of the routines. Unreachable values are also counted as constants, to account for the case when a constant value is found to be unreachable. Note that fewer congruence classes is better because it implies more congruences. The unified algorithm also finds 1 to 4 more congruence classes in 0.1% of the routines. This is because of value inference. Although it usually finds more congruences in practice, this cannot be guaranteed. An example of where it loses in SPEC is when an expression E is computed within the if- and else- branches of a conditional statement. If value inference modifies the value of one of the operands of E, it will break the congruence between the two occurrences of E.

This is a comparison of the unified algorithm with Wegman and Zadeck's sparse conditional constant propagation algorithm. To emulate Wegman and Zadeck's algorithm we enabled optimistic value numbering, constant folding, algebraic simplification and unreachable code elimination, and disabled congruence finding between non-constant expressions. We did this by replacing non-constant expressions with their result values before hash table lookups. The unified algorithm finds 1 to 100 more unreachable values in 0.6% of the routines, and 1 to 102 more unreachable and constant values in 4.6% of the routines.

Finally, here is a comparison of the unified algorithm performing optimistic against balanced value numbering. Optimisic value numbering finds 1 to 203 more unreachable values in 0.03% of the routines, 1 to 207 more unreachable and constant values in 0.2% of the routines, and 1 to 115 less congruence classes in 0.3% of the routines. So balanced value numbering is almost as strong as optimistic value numbering.

The conclusions of this work are: First, sparse value numbering is practical and efficient. Second, balanced value numbering is a good tradeoff between compilation time and optimization strength. Third, sparse value numbering can be unified with a wide range of additional analyses. Fourth, the unified algorithm offers modest improvements over existing methods. Finally, the unified algorithm does not require any modifications to classical SSA. Thank you to Laurie Hendren for helping to prepare and presenting this slide set. Questions or comments regarding this work may please be sent to the author at kg@india.hp.com.