Combining Analyses, Combining Optimizations - Summary

1. INTRODUCTION

Cliff Click's thesis "Combining Analysis, Combining Optimizations" [Click and Cooper 1995] uses a structurally different intermediate representation to combine Dead Code Elimination with Constant Propagation. The combined analysis enables better performing JIT compiled code. This technique is currently being used in the Java Hotspot Server compiler.

The intermediate representation uses Static Single Assignment (SSA) form. Knowledge of SSA is assumed in this paper. Consider the following source code:

Source Program	SSA Program Results

`int x = 1;`
`do {`
 	
`  cond = (x != 1);`
`  if (cond) {`
`  x = 2;`
`  }`
 	
`} while ( read() );`
`return x;`
	`x0: 1;`
	`do {`
	`  x1: phi ( x0, x3 );`
	`  cond:  (x1 != 1);`
	`  if (cond) {`
	`  x2:  2;`
	`  }`
	
	`  x3: phi (x2, x1);`
	`} while ( read() );`
	`return x3;`

Table I. The example program source code and the transformation of the program into SSA Form.

We will use this SSA program as the running example for the rest of this paper.

2. A SEA OF NODES

Key to the analysis is an intermediate representation that unhinges control flow from data. Each SSA data value is represented as a node in a graph. A node produces a value. (eg, Add 1 + 2). A node points to its operands (the constants 1 and 2). There is no other extra data. The "Sea Of Nodes" refers to the nodes in a slightly modified Value Dependency Graph (VDG) [Weise et al. 1994]. The Value Dependency Graph has three main advantages:

(1) Find constants that are global - have only one node that represents a constant number 2.
(2) Find dead code - any data that isn't a data dependency is now dead code.
(3) Global code motion - insert nodes into the most optimal location in the CFG later.

Contrast this with most compilers that represent a program as a sequential ordering of instructions that must execute in a given order for the program's execution to be correct. Consider the example SSA program. Most compilers represent the code as shown in Figure 1.

There are two distinct items. The basic blocks and the actual data values themselves. Each data value is ordered by the basic block it is in. The Value Dependency Graph removes the basic block structure, leaving only the values. However, to actually compile and execute the program, the notion of control is required. The Value Dependency Graph solves this by adding explicit `CONTROL` nodes that can change control flow (if, break, return, etc). The targets of these branches are then modeled as data dependencies. Each `REGION` node corresponds to a basic block in the control flow graph. `IF` control nodes have three `REGION` data nodes as their operands. One for the true path and one for the false path. `JUMP` nodes point to one `REGION` node. Two special nodes are added that are unique to the Value Dependency Graph. The first is the `START` node,

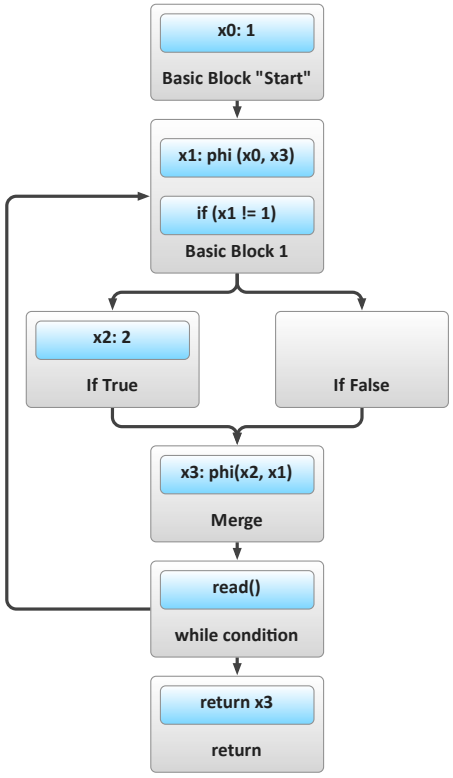

Fig. 1. The control flow graph of the example program. The control flow graph has an order in which instructions must execute. In addition, both data and control flow are represented.

which represents the root of the program. The second is the `STOP` node which represents the end of the graph. Nodes then point to their operands to create data dependencies. The example program represented as a Value Dependency Graph is shown in Figure 2.

3. EXECUTION MODEL

The traditional compiler model executes a program by starting at the entry basic block, iteratively executing over all the instructions in the basic block, in the order of a list. Consider adding two numbers:

$$x = 1 + 2;$$

A basic block executes instructions in the following necessary order:

—*firstValue* = Constant int 1; // Create int 1
—*secondValue* = Constant int 2: // create int 2
—*additionValue* = *firstValue* + *secondValue*;

The Value Dependency Graph is executed recursively, starting at the `STOP` node and executes each value's data dependent operands. Execution continues via a depth first search on a value's operand, and once all operands of a value have been visited, the actual execution of the original instruction can commence.

—*additionValue* - Has two operands. Get the *firstValue* and *secondValue*.
—*firstValue* - Produce the number 1.
—*secondValue* - Produce the number 2.
—*additionValue* - Add the two operands.

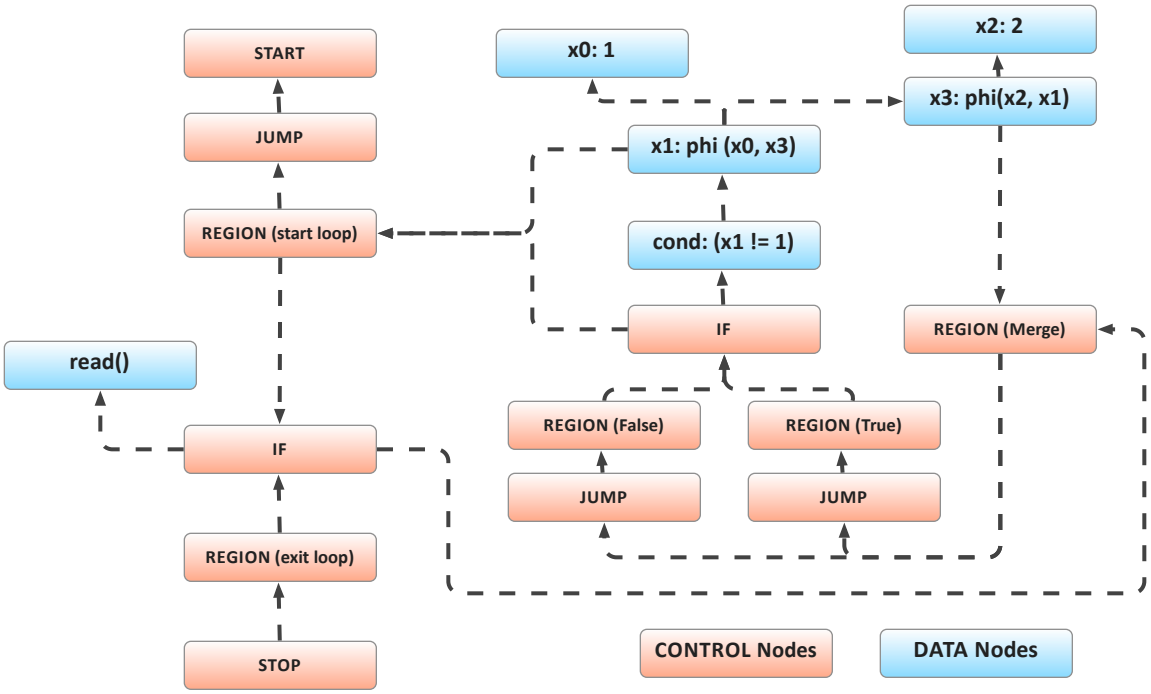

Fig. 2. The Value Dependency Graph. Special control nodes are inserted to represent control flow.

Another way of thinking about this is that the traditional compiler iterates over the CFG by visiting a value's definition prior to its uses. Execution begins at the `START` node until the `STOP` node is reached. The "Sea of Nodes" approach visits a values use, then its definition. Execution begins at the `STOP` node until the `START` node is reached.

4. THE SCHEDULER

The scheduler takes as an input the Value Dependency Graph and outputs the nodes in a traditional CFG order. As a side effect, global code motion is a built in optimization. The scheduler is only required if the VDG needs to be serialized, or have some kind of order. Thus, to JIT compile the graph, the graph must be scheduled. You can interpret the nodes via the execution model before this section. The scheduler has two functions:

(1) Rebuild the control flow graph, including the basic block structure.
(2) Insert each node in the Value Dependency Graph into a basic block.

The only rule the scheduler must adhere to is that all definitions of a value must dominate those value's uses. Cliff Click exemplifies two scheduling algorithms:

(1) Schedule Early - Greedily insert a value's definition into the first basic block that dominates all those value's uses where all input operands are available.
(2) Schedule Late - Insert a value's definition into the last basic block that dominates all of the value's uses.

Scheduling Early reduces code size because a value is guaranteed to only exist in one basic block. However, this increases the live range of the value, increasing register pressure. Scheduling late alleviates register pressure by copying a value to multiple basic blocks or pushing a value as late as possible in the CFG. The thesis does not provide any evaluation on which scheduling algorithm is performs better. Once nodes have

SSA Program	Constant Propagation Equations and Results

x0: 1;
do {
x1: phi (x0, x3);
cond: (x1 != 1);
if (cond) {
x2: 2;
}
 	
x3: phi (x2, x1);
} while (read());
return x3;
	x0 = 1
	
	x1 = x0 MEET x3 (Unknown)
	cond = (x1 != 1) (Unknown)
	
	x2 = 2;
	
	x3 = x1 MEET x2 (Unknown)
	read = Unknown

Table II. The SSA program and constant propagation equations. Each data value has an equation associated with it. The Phi value becomes the MEET operator, with each operand of the Phi being an input to the MEET operator.

Const Prop	**Undefined**	**Constant**	**Unknown**	Details
Undefined	Undefined	Constant 1	Undefined	Undefined MEET Undefined = Undefined
constant	Constant 0	(c0 == c1) ? co : Unknown	Unknown	Two Constants MEET, if equal, output constant.
Unknown	Undefined	Unknown	Unknown	Unknown MEET Undefined = undefined.

Table III. The definition for the MEET operator. Columns and rows are the operands to the MEET operator. MEET results here are defined by the identity (A == A) function.

been scheduled, the program can be code generated via other well known techniques that work on a control flow graph.

5. PUTTING IT ALL TOGETHER

The overarching process is as follows:

(1) Construct SSA during parsing of the application program.
(2) Build the Value Dependency Graph, inserting control flow operations as `REGION`, `START`, `STOP`, `JUMP` data nodes where necessary.
(3) Perform constant propagation and dead code elimination.
(4) Rebuild the control flow graph with the scheduler.
(5) Reinsert nodes from the Value Dependency Graph into basic blocks in the new control flow graph.
(6) Generate native machine code for the program with any CFG based technique.

6. CLIFF CLICK'S THESIS

Cliff Click uses the sea of node approach to efficiently do constant propagation and dead code elimination in a combined manner. The technique discovers more dead code and constants than previous techniques.

6.1 Constant Propagation

Consider the example SSA program. Each data value is analyzed, regardless of control flow, to find any SSA value that is constant. Each data value in the graph is assigned an equation. The equation dictates how each data value is evaluated. In the case of constant propagation, data values in the VDG that are constant have a constant equation which represents the literal constant value. Phi instructions are a combination of the results of their operands. The equations used for each data value and results for the constant propagation analysis in the example SSA program are shown in Table [II].

SSA Program	Unreachable Code Equations

x0: 1;
do {
x1: phi (x0, x3);
cond: (x1 != 1);
if (cond) {
x2: 2;
}
 	
x3: phi (x2, x1);
} while (read());
return x3;
	s0 = Reachable
	
	s1 = s0 OR (pred() != FALSE)
	s2 = s1; // If s1 is reachable, s2 is as well
	s3 = s2; // If s2 is reachable, s3 is as well
	s4 = s3 AND (cond != constant FALSE)
	
	s5 = s4 OR (s3 AND (cond != constant TRUE))
	s6 = s5
	s7 = s6 AND (read() != TRUE)

Table IV. The unreachable code equations. There is one equation per statement in the program.

We must make a distinction here between Undefined and Unknown. Undefined means that the SSA value isn't understood yet, but that value can later become constant. Unknown means that nothing at all is known about the value, nor can we ever in any phase prove anything about the value. Since it is impossible to reason about the `read()` method, it is assigned a value Unknown.

MEET is a function that merges the constant propagation results between two operands. The MEET operator is dependent on a specific constant propagation function. For example, consider the identity function. (A == A). The MEET operator reflects the result of the identify function. The analysis can have multiple MEET definitions (A * 1, A + 0 = A, A - 0, etc), but the actual algorithm is the same. The result of this analysis is that only x0, and x2 are constant. x1, x3, and cond are unknown values. The full results for the MEET function defined for *IDENTITY* operation is shown in Table [III]. The MEET function takes in two arguments and outputs the same constant if the two arguments are the same constant value.

6.2 Dead Code Elimination

Dead Code Elimination finds code that is unreachable. This phase like the constant propagation analysis, is performed independent of any control flow information. The result of dead code elimination is that each statement is noted as either REACHABLE or UNREACHABLE. Reachable is code that is alive. Unreachable is dead code. Any code that can be determined to be unreachable is deleted.

The first statement of any program is automatically marked as REACHABLE. If all the operands of a statement are REACHABLE, then the statement is also reachable. Each statement is given an equation that determines whether or not the given statement is reachable. The only time data is marked as unreachable is if a control flow's condition is constant. The equations used to evaluate each statement and the results of the dead code elimination analysis for the program are in Table [IV]. Because *cond* and *read()* are not literal constants, all the statements in the program are reachable via this analysis.

6.3 Combining the Two

Const Prop	**UNREACHABLE**	**REACHABLE**	Details
Undefined	Undefined	Undefined	If Undefined, output Undefined
constant	Undefined	constant	If constant and unreachable, output undefined. Else constant and reachable, output constant.
Unknown	Undefined	Unknown	If Unknown and Unreachable, output undefined. Else Unknown.

Table V. Using reachability analysis results in the constant propagation analysis to output new constant propagation results.

When we combine constant propagation with dead code elimination, we find that *cond* is a constant false.

The main insight of Cliff Click's thesis is that constant propagation and dead code elimination can occur at the same time by modifying the corresponding equations. This is done by including the results of the dead code eliminator with the constant propagation equations and vice versa. Each equation is modified to have two inputs { x, s } where x is an SSA value for constant propagation and s is a statement for dead code elimination. Each equation is solved using a combined analysis. The result is a new output { x', s' }. This new result is then used as input for the same equations. This is performed iteratively until a fixed point is reached. The constant propagation values that rely on the reachability analysis to output new constant propagation results ({x, s} $->$ x') shown in Table [V]:

All inputs that are unreachable return Undefined because the optimization passes are optimistic. A later optimization may mark a specific branch as reachable. If the constant propagation phase marks the value as Unknown at this point, the value could not "upgrade" to a constant value.

Branch Path	**Undefined**	**False**	**True**	**Unknown**	Details
False Path	Unreachable	Reachable	Unreachable	Reachable	If Undefined, false path is unreachable. If False constant, False path is reachable, etc
True Path	Unreachable	Unreachable	Reachable	Reachable	If Undefined, true path is reachable. If False constant, true path is unreachable, etc

Table VI. Using constant propagation results in the reachability analysis to output new reachability analysis results.

The reachability analysis uses the constant propagation results to output new reachability analysis results ({x, s} $->$ s') shown in Table [VI]. All SSA values that are undefined are labeled Unreachable by default because the optimization is optimistic. It assumes all code is dead until proven reachable. Since this is a branch condition, the constant propagation results are mapped to either True or False. The semantics of what constant values return true/false depend on the programming language.

Note that the same set of inputs { CONSTANT PROPAGATION, REACHABILITY } are used in each function. Once we have the results for each function, the new set of results are used in the next iteration. No intermediate value is ever used as inputs for the same iteration. Like SSA Phi instructions, both functions must conceptually occur at the same time. (eg. if we have an input {a, b} that after a function outputs to {x, y}, we should never use {a, y} or {x, b}).

The modified example now has two equations associated with each value and statement. 1) The constant propagation equations. 2) The Unreachable Code equations. The following example combines the equations for each data value used in the constant propagation analysis and statement equations used in the dead code elimination examples. It also shows how to use the reachability analysis results as input for each constant propagation value for a new constant propagation result ({x, s} $->$ x').

Time	**s0**	**x0**	**s1**	**x1**	**s2**	**cond**	**s3**	**s4**	**x2**	**s5**	**x3**	**s6**	**read()**	**s7**
0	U	T	U	T	U	T	U	U	T	U	T	U	T	U
1	**R**	T	U	T	U	T	U	U	T	U	T	U	T	U
2	R	**1**	U	T	U	T	U	U	T	U	T	U	T	U
3	R	1	**R**	T	U	T	U	U	T	U	T	U	T	U
4	R	1	R	**1**	U	T	U	U	T	U	T	U	T	U
5	R	1	R	1	**R**	T	U	U	T	U	T	U	T	U
6	R	1	R	1	R	**CF**	U	U	T	U	T	U	T	U
7	R	1	R	1	R	CF	**R**	U	T	U	T	U	T	U
8	R	1	R	1	R	CF	R	**U**	T	U	T	U	T	U
9	R	1	R	1	R	CF	R	U	T	**R**	T	U	T	U
10	R	1	R	1	R	CF	R	U	T	R	**1**	U	T	U
11	R	1	R	1	R	CF	R	U	T	R	1	**R**	T	U
12	R	1	R	1	R	CF	R	U	T	R	1	R	**B**	U
13	R	1	R	1	R	CF	R	U	T	R	1	R	B	**R**
14	R	1	R	1	R	CF	R	U	T	R	1	R	B	R

Table VII. T = Undefined. U = Unreachable. R = Reachable. CF = constant false. B = Unknown. S = statement for dead code elimination. X = variable for constant propagation. Changes at each iteration are marked as **bold**.

SSA Program	Combined Equations (Use Constant prop For Reachability Analysis)
`x0: 1;`	`s0 = Reachable` `x0 = [s0 -> 1] // Arrow denotes feed S0 into 1`
`do {`	`// Using functions above`
`x1: phi ( x0, x3 );`	`s1 = s0 OR ( pred() != FALSE )` `x1 = [s1 -> (x0 MEET x3)]`
`cond: (x1 != 1);`	`s2 = s1;` `cond = [s2 -> (x1 != 1)`
`if (cond) {`	`s3 = s2;`
`x2: 2;`	`s4 = s3 AND (cond != constant FALSE)` `x2 = [s4 -> 2];`
`}`	
`x3: phi (x2, x1);`	`s5 = s4 OR (s3 AND (cond != constant TRUE))` `x3 = [s5 -> (x1 MEET x2)]`
`} while ( read() );`	`s6 = s5` `read() = Unknown`
`return x3;`	`s7 = s6 AND ( read() != TRUE )`

First, the constant propagation or dead code elimination equation is solved in full. Next, the constant propagation results are used as inputs to the dead code eliminator using the functions listed in Table [VI] [V], and vice versa. We then can proceed to solve the combined equation to actually find dead code and global constants. Each equation does not change during the analysis, only the inputs to the equation. Each dead code elimination and constant propagation variable are first inserted into a list. Any inputs that change during one iteration are reinserted into the list. When the list is empty, the algorithm is finished. The following iterations are explained in detail:

—Time 0: All dead code variables are initialized to Unreachable (U). All constant propagation variables are initialized to Undefined (T).

—Time 1: Statement s1 is set to Reachable.

—Time 2: Variable X0 is set to constant 1. (Constant & Reachable map to constant).

—Time 3: Statement S1 is set to Reachable. (Reachable & Undefined = Reachable).

—Time 4: Variable x1 is found to be constant 1. {(x3 = undefined), (x0 = 1), (Undefined MEET 1 = 1), (Constant & Reachable (s1 = R at T = 3) = maps to constant)}.

—Time 6: Variable cond set to constant false. {(s2 = Reachable), (x1 = const 1), (Constant & Reachable (s2 = R at T = 5) = map to constant) }.

—Time 8: Statement S4 is left as Unreachable. {(s3 = Unreachable), (cond = constant false), (Unreachable & Constant false = true path unreachable)}. If statement now found as dead code.

—Time 10: Variable x3 marked as constant 1. {(x1 = 1), (x2 = const 2, but on a dead code branch because s4 is unreachable), (x1 MEET x2 = 1), (Constant & Reachable (s5 at T = 9) = constant)}.

In the current example, we actually find that the value "cond" is always false. Therefore, the if statement is dead code due to constant propagation. Running dead code or constant propagation independently does not reach this conclusion. Chapter 4 of Cliff Click's thesis then details how to perform this algorithm more efficiently. The iterative algorithm shown above runs in O(n^2) where n is the number of nodes in the sea of nodes. The reduced execution time is O(n log n).

6.4 Intermediate Representation Semantics

The "sea of nodes" approach is strictly a structural definition of an intermediate representation. The actual semantics of each instruction is left up to the compiler developer. The only overriding design concept is that the instruction set is grouped by the number of operands each instruction has, rather than a logical grouping of the semantics of each instruction. This is because of the innate feature of the combined analysis approach. For two optimizations to be combined, their functions have to have the same number of inputs and output the same number of values.

7. USAGE AND PERFORMANCE ANECDOTES & CONCLUSION

The main advantage of the combined analysis is that global data flow analysis is more effective due to the lack of order imposed on the data. The analysis examines data independent of any control flow order, find constants globally, and then reorder the program such that the results of the analysis are still correct. The results are up to 25% faster. On average, speedups of 10% to 20% are achieved. In Java, the Hotspot Server Compiler, which uses this technique, is on average, 10% to 20% faster than the Java Client Compiler [Kotzmann et al. 2008]. The Java Client Compiler uses an SSA based intermediate representation with a control flow graph.

However, the Java server compiler is about 10x slower in terms of start up time than the client compiler. In addition, there is a cost to the VM developer. The Value Dependency Graph is difficult to debug as there is no logical mapping from application source code to intermediate representation like there is in a CFG. This makes it difficult to find bugs anywhere in the intermediate representation. In Java, the other main issue is that exception edges dominate the Value Dependency Graph since almost any Java instruction can throw an exception. Thus, like all engineering decisions, there must be a careful balance between JIT compiled runtime performance and the costs of startup and development time.

Pros:

(1) Generates better JIT compiled code than a CFG based techniques.

(2) Finds more constants and dead code due to the intermixing of the optimization results.

(3) Removes the "phase" problems of other compilers. (eg one optimization needs to occur prior to another phase).

(4) Global code motion is built into the design.

Cons:

(1) Expensive start up costs, which may override any JIT compiled code performance concerns. (In Java Hotspot Server Compiler).

(2) The Scheduler is often a performance bottleneck. (In Java Hotspot Server Compiler).

(3) Difficult to debug. Usual bugs are a lost node due to a misaligned pointer. No easy logical mapping between source program and IR due to lack of control structure.

(4) Exception edges dominate the sea of nodes.

REFERENCES

Click, C. and Cooper, K. 1995. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17,* 2, 181–196.

Kotzmann, T., Wimmer, C., Mossenbock, H., Rodriguez, T., Russell, K., and Cox, D. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO) 5,* 1, 1–32.

Weise, D., Crew, R., Ernst, M., and Steensgaard, B. 1994. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 297–310.