Compiler Optimization Notes

January 4, 2023

Contents

1 Local Optimizations

Local Optimizations never goes away because this is always a piece of what happens even when we talk about even more sophiscated types of optimizations.

First we will talk about how to represent the code within a function or procedure, that's using something called a flow graph which is made of basic blocks. Next we will contrast two different abstractions for doing local optimizations.

1.1 Basic Blocks/Flow graphs

1.1.1 Basic Blocks

A basic block is a sequence of instructions(3-address statements). There are some requirements for basic block:

- **Only the first instruction can be reached from outside the blcok.** The reason why this property is useful is that within a basic block, we just march instruction by instruction through the block, this simplies things at least within a basic block.
- **All the statements are executed consecutively if the first one is.**
- **The basic block must be maximal.** i.e., they cannot be made larger without violating conditions.

1.1.2 Flow graphs

Flow graph is a graph representation of the procedure. In flow graph, basic blocks are the nodes, and the edge for $B_i \to B_j$ stands for a path from node B_i to node B_j. So how will $B_i \to B_j$ happen? There are two possibilities:

- Either first instruction of B_j is the target of a goto at end of B_i.
- B_j physically follows B_i which doesn't end in an unconditional goto.

1.1.3 Partitioning into Basic Blocks

- Identify the leader of each basic block
  - First instruction
  - Any target of a jump
  - Any instruction immediately following a jump
- Basic block starts at leader and ends at instruction immediately before a leader(or the last instruction).

An example of flow graph is shown below:

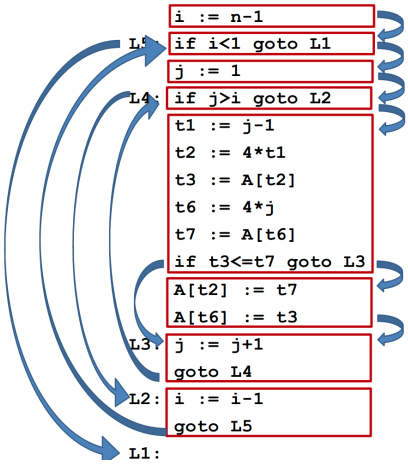


Figure 1: Example of a flow graph

1.1.4 Reachability of Basic Blocks

There is one thing interesting need to mention here. So the source code is below:

```
if x {
    ...
    return;
} else {
    ...
}
```

Listing 1: An example

The corresponding flow graph is shown in 2:

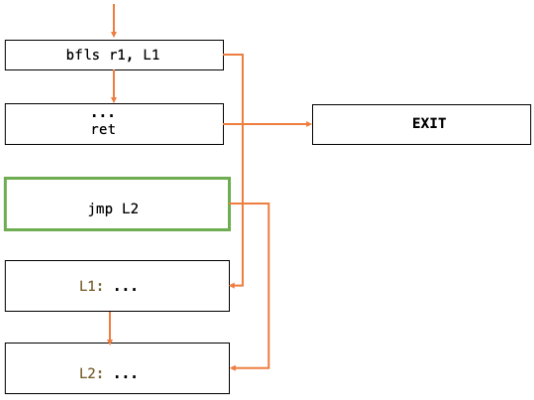


Figure 2: Example of a flow graph

We can see that the box in green is unreachable from the entry. So why is that interesting? Typically, after compiers construct the control flow graph, they will go through and remove any unreachable nodes. Just do depth first traversal of the graph from the entry node and mark all those visited nodes. So unmarked nodes will be deleted. This will help the compiler get a better optimization result.

So why do these unreachable nodes appear? The anwser is it is not the job of the front-end of the compiler to clean up the unreachable nodes.

1.2 Local optimizations

Local optimizations are those occur **within the basic blocks**.

1.2.1 common subexpression elimination

There're some types of local optimizations. One is called **common subexpression elimination**. Subexpressions are some arithmetic expressions that occur on the right hand of the instructions. The goal of this common subexpression elimination is to identify expressions that are guaranteed to produce identical values at runtime and arrange to only perform the associated computation once (when the first instance of the expression is encountered).

```
a = b + c;
d = b + c;
```

Listing 2: Subexpression example

In the example 2, `b + c` is so called coomon subexpression, we could replace the instruction containing common subexpression with an assign expression.

```
    a = b + c;
    d = a
```

Listing 3: code snippet applied common subexpression elimination to 2

You may wonder why this kind of redundancy can occure in code? Are we programmers stupid to do so? In fact, the redundancy most comes from the stage when compilers turn your source code. For example, **when you use arrays,** you need to do some arithmetic to generate the address of the array element you are accessing. So every time you referece the same array element, compiler will calculate the same address again. Similarly, if you **access offsets within fields**. Last example is **access to parameters** in the stack.

1.3 Abtraction 1:DAG

DAG is the acronym for Directed Acyclic Graph. The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. DAG is an efficient method for identifying common sub-expressions.[1]

The parse tree and DAG of the expression $a + a * (b + c) + (b + c) * d$ is shown in 3.

In DAG, some of the computation are reused. So we can generate optimizaed code based on DAG.

[1]copied from https://wildpartyofficial.com/what-is-dag-in-compiler-construction

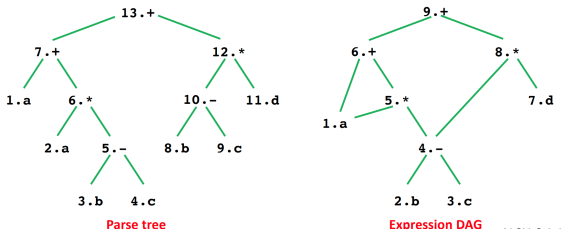


Figure 3: Example of a DAG

The optmized code for the DAG3 is:

```
t1 = b - c;
t2 = a * t1;
t3 = a + t2;
t4 = t1 * d;
t5 = t3 + t4;
```

Listing 4: code

1.3.1 How well do DAGs hold up across statements?

We have seen that DAGs can be useful in a long arithmetic expression. So how well do DAGs perform in sequence of instructions?

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

Listing 5: code

The corresponding DAG is shown in 4.
Based on the DAG4, one optimizaed code is 6

```
a = b+c;
d = a-d;
c = d+c;
```

Listing 6: code

6 is not correct. B need to be overwritten but not yet. So if using DAGs, you need to be very careful.

DAGs make sense if you just have one long expression, but once you have sequence of instructions overwriting variables , DAGs are less appealing because this abstraction doesn't really include the concept of time.

1.4 Abtraction 2:Value numbering

We have seen drawbacks of DAGs. One way to fix the problem is to attach variable name to latest value. Value numbering is such abstraction.

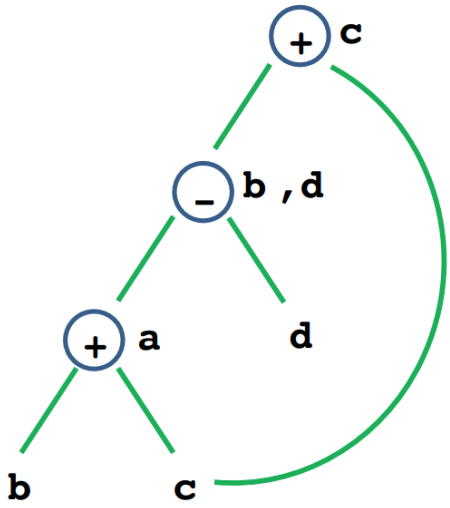


Figure 4: Example of a DAG

The idea behind value numbering is there is a mapping between variables(static) to values(dynamic). So common subexpression means same value number.

1.4.1 Algorithm

```
Data structure:
    VALUES = Table of
        expression /* [OP, valnum1, valnum2] */
        var /* name of variable currently holding expr */
For each instruction (dst = src1 OP src2) in execution order
    valnum1=var2value(src1); valnum2=var2value(src2)

    IF [OP, valnum1, valnum2] is in VALUES
        v = the index of expression
        Replace instruction with: dst = VALUES[v].var
    ELSE
        Add
            expression = [OP, valnum1, valnum2]
            var = tv
        to VALUES
        v = index of new entry; tv is new temporary for v
        Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                                  dst = tv
    set_var2value (dst, v)
```

Listing 7: code

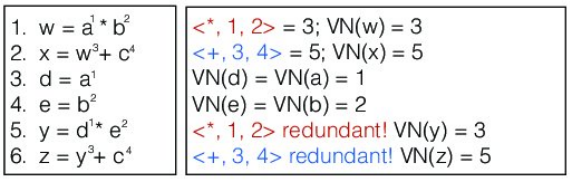


Figure 5: An example of value numbering.

1.4.2 Example

Figure 5 shows a concrete example of how VN identifies computation redundancies within a basic block. The VN processes each instruction statically. It obtains the previously computed symbolic value of each operand on the RHS, assigning a unique number on encountering a new operand. Then, it hashes the symbolic values assigned to operands together with the operator to obtain a symbolic value for the computation. If the computed symbolic value for a computation is already present in the table of previously computed values, then the current computation is redundant. In this basic block, computations on Line 5 and 6 are redundant since the computations are already computed by instruction on Line 1 and 2. [2]

[2] copied from https://www.researchgate.net/publication/283214075_Runtime_Value_Numbering_A_Profiling_Technique_to_Pinpoint_Redundant_Computations

2 Introduction to Data Flow Analysis

2.1 Motivation for Dataflow Analysis

Some optimizations[3] , however, require more "global" information. For example, consider the code 8

```
1   a = 1;
    b = 2;
3   c = 3;
    if (...) x = a + 5;
5   else x = b + 4;
    c = x + 1;
```

Listing 8: An example to illustrate some optimizations require more global information

In this example, the initial assignment to `c` (at line 3) is useless, and the expression `x + 1` can be simplified to 7, but it is less obvious how a compiler can discover these facts since they cannot be discovered by looking only at one or two consecutive statements. A more global analysis is needed so that the compiler knows at each point in the program:

- which variables are guaranteed to have constant values, and
- which variables will be used before being redefined.

To discover these kinds of properties, we use dataflow analysis.

2.1.1 What is Data Flow Analysis?

Local Optimizations only consider optimizations within a node in CFG. Data flow analysis will take edges into account, which means composing effects of basic blocks to derive information at basic block boundaries. Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program.

Typically, we will do local optimization for the first step to know what happens in a basic block, step 2 is to do data flow analysis. In the third step, we will go back and revisit the individual instructions inside of the blocks.

Data flow analysis is **flow-sensitive**, which means we take into account the effect of control flow. It is also a **intraprocedural analysis** which means the analysis is within a procedure.[4] Data-flow analysis computes its solutions over the paths in a control-flow graph. The well-known, meet-over-all-paths formulation produces safe, precise solutions for general dataflow problems. All paths-whether feasible or infeasible, heavily or rarely executed-contribute equally to a solution.

Here are some examples of intraprocedural optimizations:

- **constant propagation**. Constant propagation is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program

[3] based on https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html

[4] **Interprocedural analysis** uses calling relationships among procedures

as possible. Expressions whose operands are all constants can be evaluated at compile time and the results propagated further.

- **common subexpression elimination** CSE is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.
- **dead code elimination**. Actually, source code written by programmers doesn't contain a lot of dead code, dead code happens to occur partly because of how the front end translates code into the IR. Doing optimizations will also turn code into dead.

2.1.2 Static Program vs. Dynamic Execution

Program is statically finite, but there can be infinite many dynamic execution paths. On one hand, analysis need to be precise, so we will take into account as much dynamic execution as possible. On the other hand, analysis need to do the analysis quickly. For a compromise, the analysis result is **conservative** and what it does is for each point in the program, combines information of all the instances of the same program point.

2.1.3 Data Flow Analysis Schema

Before thinking about how to define a dataflow problem, note that there are two kinds of problems:

- Forward problems (like constant propagation) where the information at a node n summarizes what can happen on paths from "enter" to n. **So if we care about what happened in the past, it's a forward problem.**
- Backward problems (like live-variable analysis), where the information at a node n summarizes what can happen on paths from n to "exit". **So if we care about what will happen in the future, it's a backward problem.**

In what follows, we will assume that we're thinking about a forward problem unless otherwise specified.

Another way that many common dataflow problems can be categorized is as may problems or must problems. The solution to a "may" problem provides information about what may be true at each program point (e.g., for live-variables analysis, a variable is considered live after node n if its value may be used before being overwritten, while for constant propagation, the pair (x, v) holds before node n if x must have the value v at that point).

Now let's think about how to define a dataflow problem so that it's clear what the (best) solution should be. When we do dataflow analysis "by hand", we look at the CFG and think about:

- What information holds at the start of the program.
- When a node n has more than one incoming edge in the CFG, how to combine the incoming information (i.e., given the information that holds after each predecessor of n, how to combine that information to determine what holds before n).
- How the execution of each node changes the information.

This intuition leads to the following definition. An instance of a dataflow problem includes:

- a CFG,
- a domain D of "dataflow facts",
- a dataflow fact "init" (the information true at the start of the program for forward problems, or at the end of the program for backward problems),
- an operator $\wedge$ (used to combine incoming information from multiple predecessors),
- for each CFG node n, a dataflow function $f_n : D \rightarrow D$ (that defines the effect of executing n).

For constant propagation, an individual dataflow fact is a set of pairs of the form (var, val), so the domain of dataflow facts is the set of all such sets of pairs (the power set). For live-variable analysis, it is the power set of the set of variables in the program.

For both constant propagation and live-variable analysis, the "init" fact is the empty set (no variable starts with a constant value, and no variables are live at the end of the program).

For constant propagation, the combining operation $\wedge$ is set intersection. This is because if a node n has two predecessors, p1 and p2, then variable x has value v before node n iff it has value v after both p1 and p2. For live-variable analysis, $\wedge$ is set union: if a node n has two successors, s1 and s2, then the value of x after n may be used before being overwritten iff that holds either before s1 or before s2. In general, for "may" dataflow problems, $\wedge$ will be some union-like operator, while it will be an intersection-like operator for "must" problems.

For constant propagation, the dataflow function associated with a CFG node that does not assign to any variable (e.g., a predicate) is the identity function. For a node n that assigns to a variable x, there are two possibilities:

- 1. The right-hand side has a variable that is not constant. In this case, the function result is the same as its input except that if variable x was constant the before n, it is not constant after n.
- 2. All right-hand-side variables have constant values. In this case, the right-hand side of the assignment is evaluated producing consant-value c, and the dataflow-function result is the same as its input except that it includes the pair (x, c) for variable x (and excludes the pair for x, if any, that was in the input).

For live-variable analysis, the dataflow function for each node n has the form: $f_n(S) = Gen_n \cup (S - KILL_n)$, where $KILL_n$ is the set of variables defined at node n, and GEN_n is the set of variables used at node n. In other words, for a node that does not assign to any variable, the variables that are live before n are those that are live after n plus those that are used at n; for a node that assigns to variable x, the variables that are live before n are those that are live after n except x, plus those that are used at n (including x if it is used at n as well as being defined there).

An equivalent way of formulating the dataflow functions for live-variable analysis is: $f_n(S) = (S \cap NOT - KILL_n) \cup GEN_n$, where $NOT - KILL_n$ is the set of variables not defined at node n. The advantage of this formulation is that it permits the dataflow facts to be represented using bit vectors, and the dataflow functions to be implemented using simple bit-vector operations (and or).

It turns out that a number of interesting dataflow problems have dataflow functions of this same form, where GEN_n and $KILL_n$ are sets whose definition depends only on n, and the combining operator $\wedge$ is either union or intersection. These problems are called GEN/KILL problems, or bit-vector problems.

3 Reaching Definitions

The Reaching Definitions Problem is a data-flow problem used to answer the following questions: Which definitions of a variable X reach a given use of X in an expression? Is X used anywhere before it is defined? A definition$d$ reaches a point p if there exists path from the point immediately following d to p such that d is not killed(overwritten) along that path.

3.1 Iterative Algorithm

Here is the iterative algorithm.

Algorithm 1 Reaching Defintions:Iterative Algorithm

Input: control flow graph CFG = (N, E, Entry, Exit)

out[Entry] = $\emptyset$ ▷ Boundary condition
for `each basic block B other than Entry` **do**
  `out[B] =` $\emptyset$ ▷ Initialization for iterative algorithm
end for
while Changes to any out[] occur **do**
  for `each basic block B other than Entry` **do**
    $in[B] = \cup(out[p])$`, for all predecessors p of B`
    $out[B] = f_B(in[B])$ ▷ $out[B] = gen[B] \cup (in[B] - kill[B])$
  end for
end while

3.2 Worklist Algorithm

3.3 Example

Here comes an example of reaching definition.

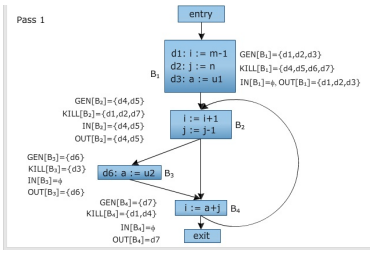


Figure 6: Pass 1

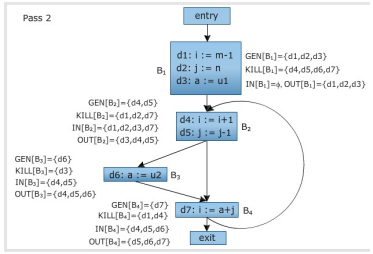


Figure 7: Pass 2

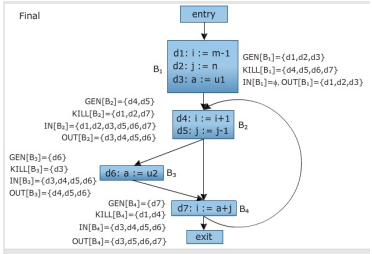


Figure 8: Pass 3

Algorithm 2 Reaching Defintions:Worklist Algorithm

Input: control flow graph CFG = (N, E, Entry, Exit)

```
out[Entry] = ∅                                          ▷ Boundary condition
ChangedNodes = N
for each basic block B other than Entry do
    out[B] = ∅                                          ▷ Initialization for iterative algorithm
end for
while ChangedNodes ≠ ∅ do
    Remove i from ChangedNodes
    in[B] = ∪(out[p]), for all predecessors p of B
    oldout = out[i]
    out[i] = f_i(in[i])                                 ▷ out[i] = gen[i] ∪ (in[i] − kill[i])
    if oldout ≠ out[i] then
        for all successors s of i do
            add s to ChangedNodes
        end for
    end if
end while
```

3.4 Summary

Direction	Forward
Domain	Sets of definitions
Meet operator	$\cup$
Top(T)	ϕ
Bottom	Universal Set
Boundary condition	OUT[ENTRY] = ϕ
Initialization for internal nodes	OUT[B] = ϕ
Finited escending chain?	✓
Transfer function	$f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b)$
Monotone&Distributive?	✓

4 Live Variabl Analysis

In compilers, live variable analysis (or simply liveness analysis) is a classic data-flow analysis to calculate the variables that are live at each point in the program. A variable is live at some point if it holds a value that may be needed in the future, or equivalently if its value may be read before the next time the variable is written to. [5]

4.1 Motivation

Programs may contain

- code which gets executed but which has no useful effect on the program's overall result;
- occurrences of variables being used before they are defined;[6]
- many variables which need to be allocated registers and/or memory locations for compilation.[7]

The concept of variable liveness is useful in dealing with all three of these situations.

Liveness analysis is highly used for **register allocation**(If variable `x` is live in a basic block b, it is a potential candidate for register allocation) and **dead code elimination**(If variable `x` is not live after an assignment `x =...`, then the assignment is redundant and can be deleted as dead code).

4.2 Problem formulation

Liveness is a data-flow property of variables: "Is the value of this variable needed?" We therefore usually consider liveness from an instruction's perspective: each instruction (or node of the flowgraph) has an associated set of live variables.

4.3 Semantic vs. syntactic

There are two kinds of variable liveness : Semantic liveness and Syntactic liveness.

A variable x is **semantically** live at a node n if there is some execution sequence starting at n whose (externally observable) behaviour can be affected by changing the value of x. Semantic liveness is concerned with the execution behaviour of the program.

A variable is **syntactically** live at a node if there is a path to the exit of the flow graph along which its value may be used before it is redefined. Syntactic liveness is concerned with properties of the syntactic structure of the program.

So what is the difference between Semantic liveness and Syntactic liveness? syntactic liveness is a computable approximation of semantic liveness.

Consider the example 9

[5] based on Wikipedia

[6] we can use liveness information to find undefined variables.

[7] Two variables can use the same register if they are never in use at the same time(i.e, never simultaneously live). Register allocation uses liveness information.

```
int t = x * y;
if ((x+1)*(x+1) == y) {
 t = 1;
}
if (x*x + 2*x + 1 != y) {
 t = 2;
}
return t;
```

Listing 9: An example to illustrate semantic syntatic

In fact, t is dead in node `int t = x * y;` because one of the conditions will be true, so on every execution path t is redefined before it is returned. The value assigned by the first instruction is never used.

But on read path from Figure 9 through the flowgraph, t is not redefined before it's used, so t is syntactically live at the first instruction.Note that this path never actually occurs during execution.

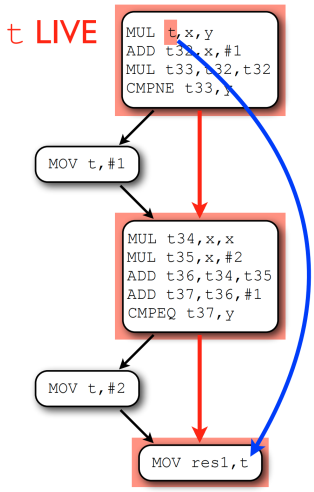


Figure 9: CFG for 9

4.4 Summary

Direction	Backward
Domain	Sets of variables
Meet operator	$\cup$
Top(T)	ϕ
Bottom	Universal Set
Boundary condition	IN[EXIT] = ϕ
Initialization for internal nodes	IN[B] = ϕ
Finited escending chain?	✓
Transfer function	$f_b(x) = \text{USE}_b \cup (x - \text{DEF}_b)$
Monotone&Distributive?	✓

4.5 Strongly Live Variables Analysis[1]

A variable is strongly live if

- it is used in a statement other than assignment statement, or (same as simple liveness)
- it is used in an assignment statement defining a variable that is strongly live

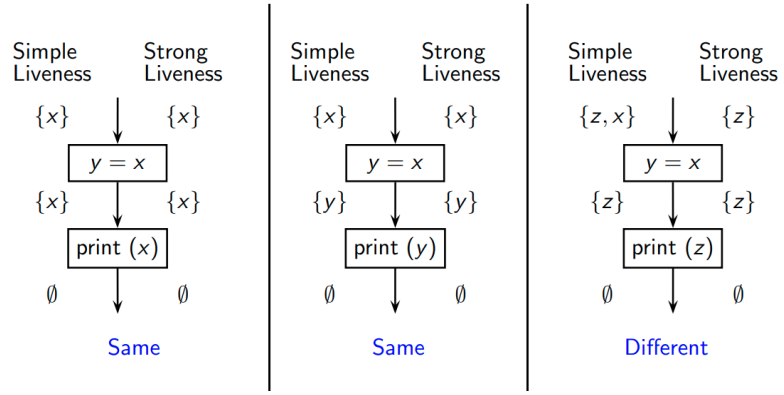


Figure 10: Understanding Strong Liveness

A variable is live at a program point if its current value is likely to be used later. We want to compute the smallest set of variables that are live. Simple liveness considers every use of a variable as useful. Strong liveness checks the liveness of the result before declaring the operands to be live. Strong liveness is more precise than simple liveness. The transfer function of Strongly Live Variables Analysis is shwon below:

$$f_n(X) = \begin{cases} (X - \{y\}) \cup (Opd(e) \cap \mathbb{V}ar) & n \text{ is } y = e, e \in \mathbb{E}pr, y \in X \\ X - \{y\} & n \text{ is input } (y) \\ X \cup \{y\} & n \text{ is use } (y) \\ X & \text{otherwise} \end{cases}$$

The first case means that If y is not strongly live, the assignment is skipped using the "otherwise" clause

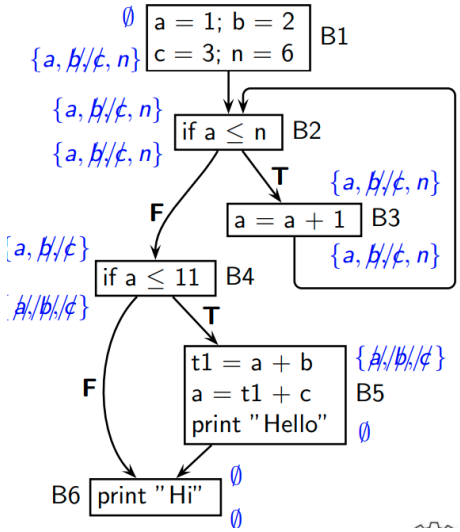


Figure 11: Simple Liveness VS. Strong Liveness.

5 Available Expressions Analysis

5.1 Motivation

Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program. The concept of expression availability is useful in dealing with this situation.

5.2 Backgroud Knowledge

Any given program contains a finite number of expressions (i.e. computations which potentially produce values),so we may talk about the set of all expressions of a program. Consider the program in 23.4.2

```
int z = x * y;
print s + t;
int w = u / v;
```

Listing 10: An simple example containing some expressions

This program contian expression `x*y,s+t,u/v`.

5.3 Problem Formulation

Availability is a data-flow property of expressions: "Has the value of this expression already been computed?" At each instruction, each expression in the programis either available or unavailable. So each instruction(or node of the flowgraph) has an associated set of available expression.

5.4 Semantic vs. Syntactic

An expression is *semantically* available at a node n if its value gets computed (and not subsequently invalidated) along every execution sequence ending at n.

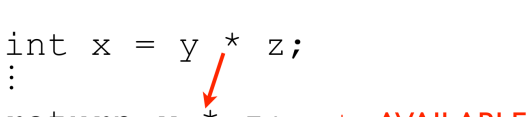


Figure 12: Available expression example

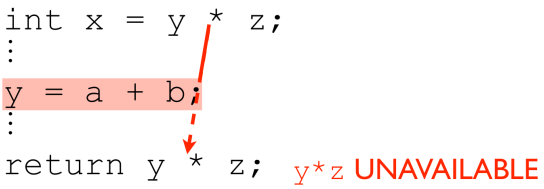


Figure 13: unavailable expression example

An expression is *syntactically* available at a node n if its value gets computed (and not subsequently invalidated) along every path from the entry of the flowgraph to n.

On the path in red from Figure15 through the flowgraph, $x+y$ is only computed once, so $x+y$ is syntactically unavailable at the last instruction.

Whereas with live variable analysis we found safety in assuming that more variables were live, here we find safety in assuming that fewer expressions are available. Because if an expression

```
if ((x+1)*(x+1) == y) {
  s = x + y;
}
if (x*x + 2*x + 1 != y) {
  t = x + y;
}
return x + y;  x+y AVAILABLE
```

Figure 14: x+y is semantically available

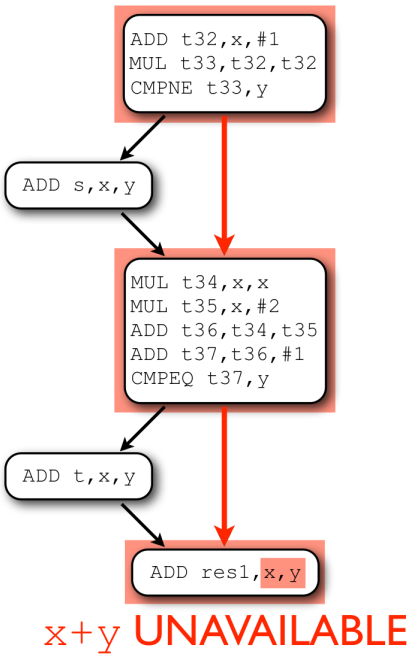


Figure 15: x+y is syntactically unavailable

is deemed to be available, we may do something dangerous (e.g. remove an instruction which recomputes its value). So sometimes safe means more, but sometimes means less.

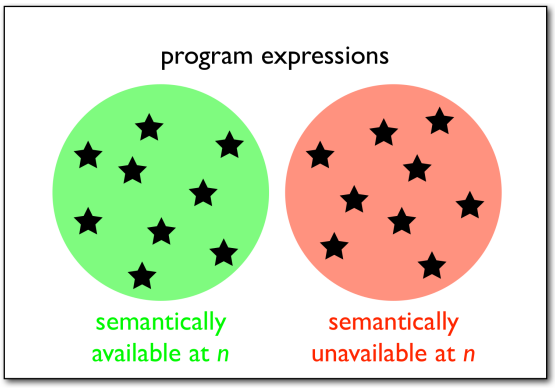


Figure 16: Semantic vs. syntactic

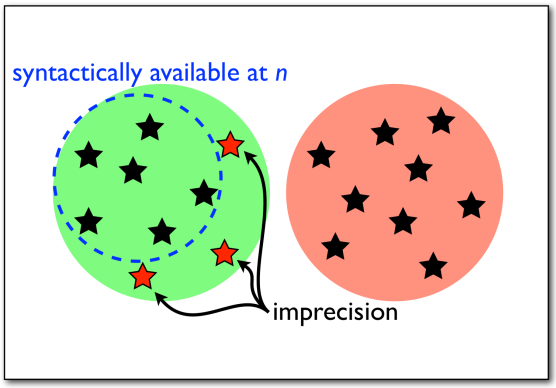


Figure 17: Semantic vs. syntactic

5.5 Summary

Direction	Forward
Domain	Sets of expressions
Meet operator	$\cap$
Top(T)	Universal Set
Bottom	ϕ
Boundary condition	OUT[ENTRY] = ϕ
Initialization for internal nodes	OUT[B] = T
Finited escending chain?	✓
Transfer function	$f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b)$
Monotone&Distributive?	✓
Kill_b	all E such that block b defines a variable in E
Gen_b	all E such that block b evaluates E and doesn't later kill it

6 Constant Propagation/Folding[2]

6.1 Problem Definition

Given a program, we would like to know for every point in the program (after every statement), which variables have constant values, and which do not. We say that a variable has a constant value at a certain point if every execution that reaches that point, gives that variable the same constant value.

6.2 Meet Operator

v1	v2	v1 ∧ v2
UNDEF	UNDEF	undef
	c_2	C2
	NAC	NAC
c_1	UNDEF	C1
	c_2	C1 if C1 == C2 NAC else
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

Figure 18: Meet operator for Constant Propagation.

6.3 Transfer Function

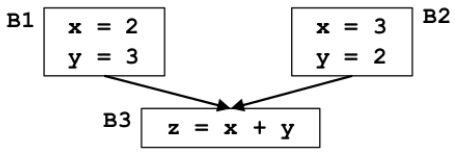
Let an assignment be of the form $x_3 = x_1 + x_2$, $+$ represents a generic operator.

IN[b,x_1]	IN[b,x_2]	OUT[b,x_3]
UNDEF	UNDEF	undef
	c_2	undef
	NAC	NAC
c_1	UNDEF	undef
	c_2	**C1 + C2**
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

Figure 19: Transfer Function.

6.4 Summary

Constant Propogation is not distributive[8]. The semi-lattice is shown in Figure 20.



	x	y	z
$f_1(T)$	2	3	UNDEF
$f_2(T)$	3	2	UNDEF
$f_1(T) \wedge f_2(T)$	NAC	NAC	UNDEF
$f_3(f_1(T) \wedge f_2(T))$	NAC	NAC	NAC
$f_3(f_1(T))$	2	3	5
$f_3(f_2(T))$	3	2	5
$f_3(f_1(T)) \wedge f_3(f_2(T))$	NAC	NAC	5

Not Distributive: $f_3(f_1(T) \wedge f_2(T)) < f_3(f_1(T)) \wedge f_3(f_2(T))$

Figure 20: Constant Propogation is not Distributive.

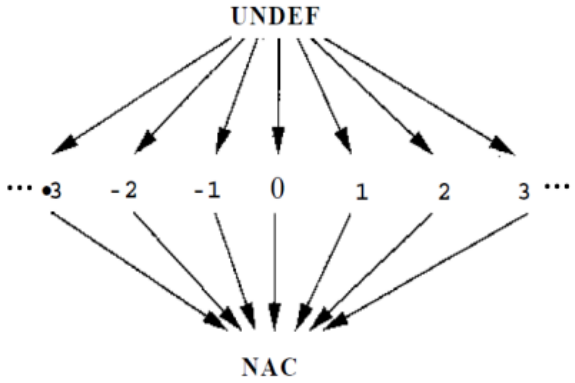


Figure 21: Constant Propogation's Semi-lattice Diagram.

[8]See Figure 21

Direction	Forward
Domain	Numbers
Top(T)	UNDEF
Bottom	NAC
Boundary condition	OUT[ENTRY] = UNDEF
Initialization for internal nodes	OUT[B] = UNDEF
Finited escending chain?	✓
Monotone ?	✓
Distributive?	X

7 Foundations of Data Flow Analysis

We saw a lot of examples of data flow analysis, eg. reaching definitions etc. Although there were differences between differeent types of data flow analysis, they did share number of things in common. Our goal is to develop a general purpose data flow analysis framework.

There are some questions that we want to answer about a framework that performs data flow analysis.

- Correctness: Do we get a correct answer?
- Precison: How good is the answer? [9]
- Covergence: Will the analysis terminate?
- Speed: How fast is the convergence?

7.1 A Unified Framework

Data flow problems are defined by

- Domain of values V(eg, variable names for liverness, the instruction numbers for reaching definitions)
- Meet operator $V \wedge V \to V$ to deal with the join nodes.
- Initial value. Once we have defined the meet operator, it will tell us how to initialize all of the non-entry or exits nodes and the boundary conditions for entry and exit nodes.
- A set of transfer functions $V \to V$ to define how information flows across basic blocks.

Why we bother to define such a framework?

- First, if meet operator, transfer function and the domains of values are specified in proper way, we will know about correctness, precision and so on.
- From practical engineering perspective, it allows us to reuse code.

7.2 Partial Order

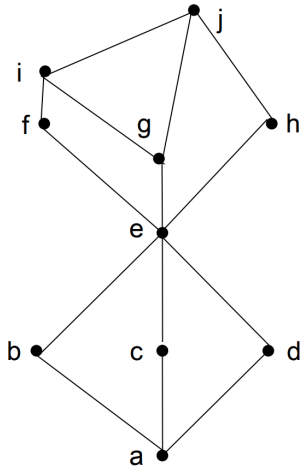
A relation R on a set S is called a **partial order** if it is

- **Transitivity** if $x \preceq y$ and $y \preceq z$ then $x \preceq z$
- **Antisymmetriy** if $x \preceq y$ and $y \preceq x$ then $x = x$
- **Reflexivity** $x \preceq x$

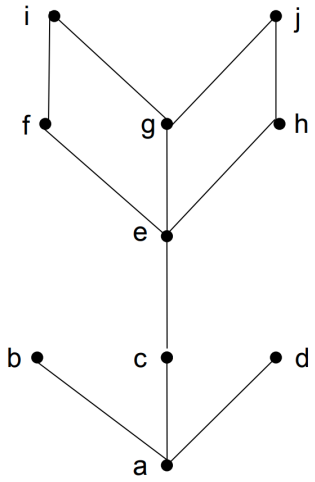
7.3 Lattice

A lattice is a partially ordered set in which every pair of elements has both a least upper bound (lub) and a greatest lower bound(glb).

[9] We want a safe solution but as precise as possible.



(a) This is a lattice example.



(b) This is not a lattice example becuasethe pair b,c deos not have a lub.

Figure 22: Two examples

7.4 Complete Lattice

A lattice A is called a complete lattice if every subset S of A admits a glb and a lub in A.

7.5 Semi-Lattice

A semilattice (or upper semilattice) is a partially ordered set that has a least upper bound for any nonempty finite subset.

7.6 Meet Operator

Meet operator must hold the following properties:

- **commutative**: $x \wedge y = y \wedge x$. No ordering in the incoming edges.
- **idempotent**: $x \wedge x = x$
- **associative** : $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- there is a Top element T such that $x \wedge T = x$. Partly due to the way we initialize everything we need.

Meet Operator defines a partial ordering on values. This is important in ensuring the analysis converges. So what does it mean ? $x \preceq y$ if and only if $x \wedge y = x$. The $\preceq$ not means less or equal to or subset, but it really means lattice inclusion. So if $x \preceq y$, this means x is more conservative

Figure 23: Meet Operator

or constrained. In another word, x is lattice included in y. Partial ordering will also lead to some other properties

- **Transitivity** if $x \preceq y$ and $y \preceq z$ then $x \preceq z$
- **Antisymmetriy** if $x \preceq y$ and $y \preceq x$ then $x = z$
- **Reflexivity** $x \preceq x$

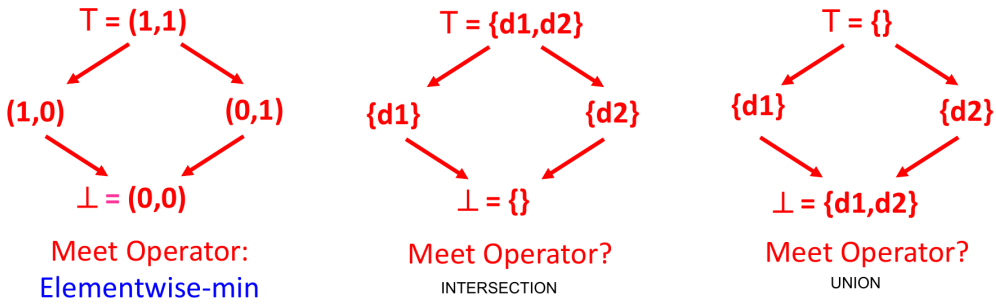


Figure 24: Different meet operator defines different lattice

For our data flow analysis, values and meet operator define a semi-lattice, which means $\top$ exits, but not necessarily $\bot$.

7.7 Descending Chain

The height of a lattice is the largest number of $\succ$ relations that will fit in a descending chain. eg. $x_0 \succ x_1 \succ x_2 \succ \ldots$

So, for reaching definitions, the height is the number of definitions.

Finite descending chain will ensure the convergence. If we don't finite descending chain, there is a possibility that the analysis will never terminate. But a infinite lattice still can have a finite descending chain. I want to note that infinite lattice doesn't always mean a non-convergence.

So consider the constant propogation, the infinite lattice has finite descending chain, so this can converge.

7.8 Transfer Functions

Tranfer function dicates how information propagates across a basic block. So what we need for our transfer function? **First**, it must have an identity function which means there exists an f such

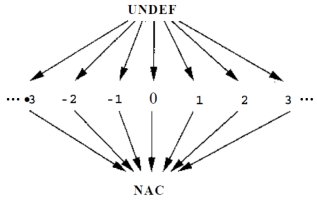


Figure 25: The lattice of constant propogation

that $f(x) = x$ for all x. For example, in Reaching Definitions and Liveness, when $Gen, KILL = \Phi$, this transfer function satisfies $f(x) = x$. **Second**, when we compose transfer functions, it must be consitent with the transfer function. So if $f_1, f_2 \in F$, the $f_1 \cdot f_2 \in F$.

For example,

$$\begin{aligned} f_1(x) &= G_1 \cup (x - K_1) \\ f_2(x) &= G_2 \cup (x - K_2) \\ f_2(f_1(x)) &= G_2 \cup [(G_1 \cup (x - K_1)) - K_2] \\ &= [G_2 \cup (G_1 - K_2)] \cup [x - (K_1 \cup K_2)] \\ G &= G_2 \cup (G_1 - K_2) \\ K &= K_1 \cup K_2 \end{aligned}$$

7.9 Monotonicity

A framework $(F, V, \wedge)$ is monotone if and only if $x \preceq y$ implies $f(x) \preceq f(y)$. This means that a "smaller(more conservative) or equal" input to the same function will always give a "smaller(more conservative) or equal" output.

Alternatively, $(F, V, \wedge)$ is monotone if and only if $f(x \wedge y) \preceq f(x) \wedge f(y)$. So merge input, then apply f is small(more conservative) or equal to apply the transfer function individually and then merge the result. Values are defined by semi-lattice, the meet operator only ever moves down the lattice from top towards the bottom. So we need to constrain the transfer function.

I will show you a unmonotone example.

Let top be 1 and bottom be 0 and the meet operator is $\cap$. $f(0) = 1, f(1) = 0$

Let's check whether reaching definitions is monotone.

Note that monotone framework does not mean $f(x) \preceq x$.

7.10 Distributivity

A framework $(F, V, \wedge)$ is distributive if and only if it

$$f(x \wedge y) = f(x) \wedge f(y)$$

Reaching definitions is distributive but constant propogation is not.

7.11 Meet-Over-Paths(MOP)[3]

A solution to an instance of a dataflow problem is a dataflow fact for each node of the given CFG. But what does it mean for a solution to be correct, and if there is more than one correct solution, how can we judge whether one is better than another?

Ideally, we would like the information at a node to reflect what might happen on all possible paths to that node. This ideal solution is called the meet over all paths (MOP) solution, and is discussed below. Unfortunately, it is not always possible to compute the MOP solution; we must sometimes settle for a solution that provides less precise information.

The MOP solution (for a forward problem) for each CFG node n is defined as follows:

- For every path "enter $\rightarrow \cdots \rightarrow$ n", compute the dataflow fact induced by that path (by applying the dataflow functions associated with the nodes on the path to the initial dataflow fact).
- Combine the computed facts (using the combining operator, $\wedge$).
- The result is the MOP solution for node n.

It is worth noting that even the MOP solution can be overly conservative (i.e., may include too much information for a "may" problem, and too little information for a "must" problem), because not all paths in the CFG are executable. For example, a program may include a predicate that always evaluates to false (e.g., a programmer may include a test as a debugging device – if the program is correct, then the test will always fail, but if the program contains an error then the test might succeed, reporting that error). Another way that non-executable paths can arise is when two predicates on the path are not independent (e.g., whenever the first evaluates to true then so does the second). These situations are illustrated in Figure 26.

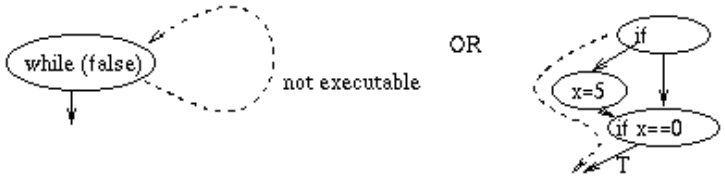


Figure 26: Examples to illustrate MOP is overly conservative.

So MOP = Perfect − Solution U Solution − to − Unexecuted − Paths

Unfortunately, since most programs include loops, they also have infinitely many paths, and thus it is not possible to compute the MOP solution to a dataflow problem by computing information for every path and combining that information. Fortunately, there are other ways to solve dataflow problems (given certain reasonable assumptions about the dataflow functions associated with the CFG nodes). As we shall see, if those functions are distributive, then the solution that we compute is identical to the MOP solution. If the functions are monotonic, then the solution may not be identical to the MOP solution, but is a conservative approximation.

7.12 Solving a Dataflow Problem by Solving a Set of Equations

The alternative to computing the MOP solution directly, is to solve a system of equations that essentially specify that local information must be consistent with the dataflow functions. In particular, we associate two dataflow facts with each node n:

- n.before: the information that holds before n executes, and
- n.after: the information that holds after n executes.

These n.befores and n.afters are the variables of our equations, which are defined as follows (two equations for each node n):

- n.before = $\wedge$(p1.after, p2.after, ...) where p1, p2, etc are n's predecessors in the CFG (and $\wedge$ is the combining operator for this dataflow problem).
- n.after = f_n(n.before)

In addition, we have one equation for the enter node:

- enter.after = init (recall that "init" is part of the specification of a dataflow problem)

These equations make intuitive sense: the dataflow information that holds before node n executes is the combination of the information that holds after each of n's predecessors executes, and the information that holds after n executes is the result of applying n's dataflow function to the information that holds before n executes.

One question is whether, in general, our system of equations will have a unique solution. The answer is that, in the presence of loops, there may be multiple solutions. For example, consider the simple program whose CFG is given in Figure 27:

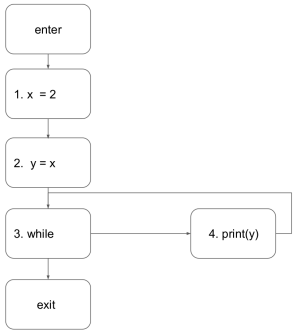


Figure 27: An example to illustrate the system of equations.

The equations for constant propagation are as follows:

`enter.after` = Φ

`1.before` = `enter.after`

`1.after` = `1.before` $-(x, *)$ union $(x, 2)$

`2.before` = `1.after`

`2.after` = if (x, c) is in `2.before` then `2.before` $-(y, *)$ union (y, c), else `2.before` $-(y, *)$

`3.before` = $\wedge$ (`2.after, 4.after`)
`3.after` = `3.before`
`4.before` = `3.after`
`4.after` = `4.before`

Because of the cycle in the example CFG, the equations for `3.before, 3.after, 4.before, and 4.after` are mutually recursive, which leads to the four solutions shown below (differing on those four values).

Variable	Solution 1	Solution 2	Solution 3	Solution 4
1.before	{}	{}	{}	{}
1.after	$\{(x,2)\}$	$\{(x,2)\}$	$\{(x,2)\}$	$\{(x,2)\}$
2.before	$\{(x,2)\}$	$\{(x,2)\}$	$\{(x,2)\}$	$\{(x,2)\}$
2.after	$\{(x,2)(y,2)\}$	$\{(x,2)(y,2)\}$	$\{(x,2)(y,2)\}$	$\{(x,2)(y,2)\}$
3.before	{}	$\{(x,2)\}$	$\{(y,2)\}$	$\{(x,2)(y,2)\}$
3.after	{}	$\{(x,2)\}$	$\{(y,2)\}$	$\{(x,2)(y,2)\}$
4.before	{}	$\{(x,2)\}$	$\{(y,2)\}$	$\{(x,2)(y,2)\}$
4.after	{}	$\{(x,2)\}$	$\{(y,2)\}$	$\{(x,2)(y,2)\}$

The solution we want is solution 4, which includes the most constant information. In general, for a "must" problem the desired solution will be the largest one, while for a "may" problem the desired solution will be the smallest one.

7.13 Iterative Algorithms

Many different algorithms have been designed for solving a dataflow problem's system of equations. Most can be classified as either iterative algorithms or elimination algorithms.

Most of the iterative algorithms are variations on the following algorithm (this version is for forward problems shown in Algorithm 3). It uses a new value T (called "top"). T has the property that, for all dataflow facts d, $T \wedge d = d$. Also, for all dataflow functions, $f_n(T) = T$. (When we consider the lattice model for dataflow analysis we will see that this initial value is the top element of the lattice.)

7.14 Kildall's Lattice Framework for Dataflow Analysis

Recall that our informal definition of a dataflow problem included:

- a domain D of "dataflow facts",
- a dataflow fact "init" (the information true at the start of the program),
- an operator $\wedge$ (used to combine incoming information from multiple predecessors),
- for each CFG node n, a dataflow function $f_n : D \to D$ (that defines the effect of executing n)

and that our goal is to solve a given instance of the problem by computing before and after sets for each node of the control-flow graph. A problem is that, with no additional information

Algorithm 3 Iterative Algorithms

```
Step 1 (initialize n.afters):
Set enter.after = init. Set all other n.after to T.
Step 2 (initialize worklist):
Initialize a worklist to contain all CFG nodes except enter and exit.
Step 3 (iterate):
while the worklist is not empty: do
    Remove a node n from the worklist.
    Compute n.before by combining all p.after such that p is a predecessor of n in the CFG.
    Compute tmp = f_n( n.before )
    if tmp != n.after then
        Set n.after = tmp
        Put all of n's successors on the worklist
    end if
end while
```

about the domain D, the operator $\wedge$, and the dataflow functions f_n, we can't say, in general, whether a particular algorithm for computing the before and after sets works correctly (e.g., does the algorithm always halt? does it compute the MOP solution? if not, how does the computed solution relate to the MOP solution?).

Kildall addressed this issue by putting some additional requirements on D, $\wedge$, and f_n. In particular he required that:

- D be a complete lattice L such that for any instance of the dataflow problem, L has no infinite descending chains.
- $\wedge$ be the lattice's meet operator.
- All f_n be distributive.

He also required (essentially) that the iterative algorithm initialize n.after (for all nodes n other than the enter node) to the lattice's "top" value. (Kildall's algorithm is slightly different from the iterative algorithm presented here, but computes the same result.)

Given these properties, Kildall showed that:

- The iterative algorithm always terminates.
- The computed solution is the MOP solution.

It is interesting to note that, while his theorems are correct, the example dataflow problem that he uses (constant propagation) does not satisfy his requirements; in particular, the dataflow functions for constant propagation are not distributive (though they are monotonic). This means that the solution computed by the iterative algorithm for constant propagation will not, in general be the MOP solution. An example to illustrate this is shown in Figure 28

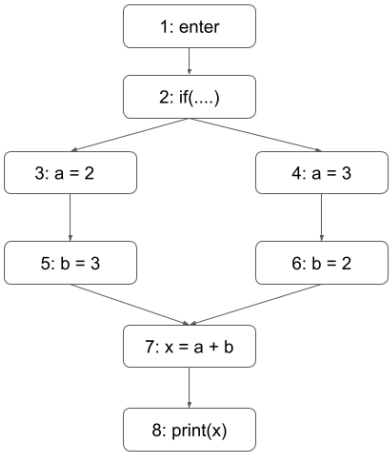


Figure 28: An example to illustrate that the solution computed by the iterative algorithm for constant propagation not equal to MOP.

The MOP solution for the final `print` statement includes the pair $(x, 5)$, since x is assigned the value 5 on both paths to that statement. However, the greatest solution to the set of equations for this program (the result computed using the iterative algorithm) finds that x is not constant at the print statement. This is because the equations require that `n.before` be the meet of `m.after` for all predecessors `m`; in particular, they require that the before set for `node 7 (x = a + b)` has empty, since the after sets of the two predecessors have $(a, 2), (b, 3)$, and $(a, 3), (b, 2)$, respectively, and the intersection of those two sets is empty. Given that value for `7.before`, the equations require that `7.after` (and `8.before`) say that x is not constant. We can only discover that x is constant after `node 7` if both a and b are constant before `node 7`.

In 1977, a paper by Kam and Ullman[4] extended Kildall's results to show that, given monotonic dataflow functions:

- The solutions to the set of equations form a lattice.
- The solution computed by the iterative algorithm is the greatest solution (using the lattice ordering).
- If the functions are monotonic but not distributive, then the solution computed by the iterative algorithm may be less than the MOP solution (using the lattice ordering).

7.15 Speed of convergence

Preorder means visit parents before children, it isalso called **reverse postorder(RPO)**. Postorder means visit children before parent. RPO visits as many predecessors possible before visiting a node so in case of forward data flow problems (like Dominator computation) this would help to converge faster.

Algorithm 4 Depth-First Iterative Algorithm

```
Step 1: depth-first post order
function MAIN()
    count = 1;
    Visit(root);
end function
function VISIT(N)
    for each successor s that has not been visited do
        PostOrder(n) = count;
        count = count+1;
    end for
end function
Step 2: reverse order
for each node i do
    rPostOrder(i)= NumNodes - PostOrder(i)
end for
Step 3: Depth-First Iterate
out[entry] = init_value
for all nodes i do
    out[i] = T
end for
Change = True
while Change do
    Change = False
    for each node i in rPostOrder do
        in[i] = ∧(out[p]), for all predecessors p of i
        oldout = out[i]
        out[i] = f_i (in[i])
        if oldout ≠ out[i] then
            Change = True
        end if
    end for
end while
```

The number of passes are determined by number of back edges in the path essentially the nesting depth of the graph

$$\text{Number of iterations} = \text{number of back edges in any acyclic path} + 2$$

7.16 Summary

Let's answer the questions we asked before.

- If D is a semi-lattice L which has finite descending chain, transfer function is monotone, then the iterative algorithm will converge.
- If dataflow framework is monotone, then if the algorithm converges, $IN[B] \preceq MOP[B]$
- If dataflow framework is distributive, then if the algorithm converges, $IN[B] = MOP[B]$

8 Introduction to Static Single Assignment

Many dataflow analysis need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the use sites of variables defined there, and a list of pointers to all definition sites of the variables used there. An improvement on the idea of *def-use chains* is *static single-assignment form*, or *SSA form*, an intermediate representation in which each variable has only one definition in the program text. SSA is very useful for many optimizations such as Loop-Invariant Code Motion and Copy Propagation.

8.1 Definition-Use and Use-Definition Chains

Use-Definition (UD) Chains

For a given definition of a variable X, what are all of its uses?

Definition-Use (DU) Chains

For a given use of a variable X, what are all of the reaching definitions of X?

Unfortunately, it is expensive to use UD and DU chains, because if we have N defs, and M uses, the space complexity is $O(NM)$. An example is in Figure 29

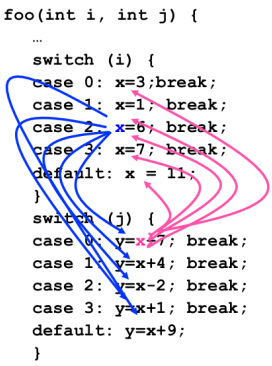


Figure 29: If a variable has N uses and M definitions (which occupy about $N+M$ instructions in a program), it takes space (and time) proportional to NM to represent def-use chains – a quadratic blowup.

8.2 Static Single Assignment(SSA)

Static Single Assignment

Static Single Assignment is an IR where every variable is assigned a value at most once in the program text.

the Φ function

Φ merges multiple definitions along multiple control paths into a single definition.
At a basic block with p predecessors, there are p arguments to the Φ functions.

$$x_{\text{new}} \leftarrow \Phi\left(\mathbf{x}_1, \mathbf{x}_1, \mathbf{x}_1, \ldots, \mathbf{x}_{\text{p}}\right)$$

8.2.1 Why SSA is useful?

Useful for Dataflow Analysis Dataflow analysis and optimization algorithms can be made simpler when each variable has only one definition.

Less space and time complexity If a variable has N uses and M definitions (which occupy about N + M instructions in a program), it takes space (and time) proportional to N · M to represent def-use chains – a quadratic blowup. For almost all realistic programs, the size of the SSA form is linear in the size of the original program.

Simplify some algorithms Uses and defs of variables in SSA form relate in a useful way to the dominator structure of the control-flow graph, which simplifies algorithms such as interference-graph construction.

Eliminate needless relationship Unrelated uses of the same variable in the source program become different variables in SSA form, eliminating needless relationships shown in 8.2.1.

```
1 for i <- 1 to N
    do A[i] <- 0
3 for i <- 1 to M
    do s <- s + B[i]
```

8.3 How to represent SSA?

In straight-line code, such as within a basic block, it is easy to see that each instruction can define a fresh new variable instead of redefining an old one shown in Figure 30

But when two control-flow paths merge together, it is not obvious how to have only one assignment for each variable. To solve this problem we introduce a notational fiction, called a Φ function. Figure 31 shows that we can combine a1 (defined in block 1) and a2 (defined in block 3) using the function $a3 \leftarrow \Phi(a1, a2)$.

$$
\begin{array}{lcl}
a & \leftarrow & x+y \\
b & \leftarrow & a-1 \\
a & \leftarrow & y+b \\
b & \leftarrow & x \cdot 4 \\
a & \leftarrow & a+b
\end{array}
\qquad
\begin{array}{lcl}
a_1 & \leftarrow & x+y \\
b_1 & \leftarrow & a_1-1 \\
a_2 & \leftarrow & y+b_1 \\
b_2 & \leftarrow & x \cdot 4 \\
a_3 & \leftarrow & a_2+b_2
\end{array}
$$

(a) A straight-line program. (b) The program in single-assignment form.

Figure 30: SSA for straight-line code

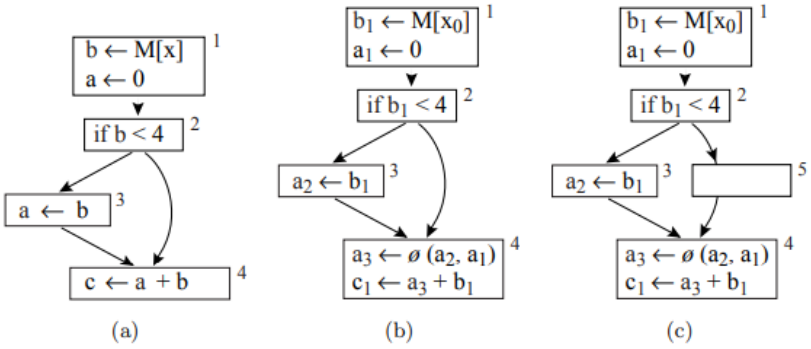
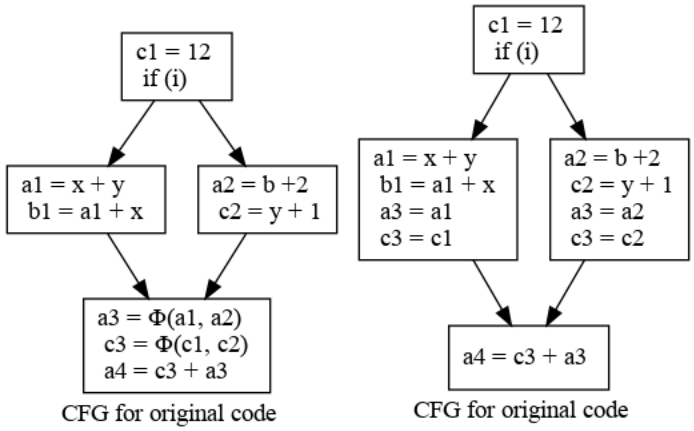


Figure 31: (a) A program with a control-flow join; (b) the program transformed to single-assignment form; (c) edge-split SSA form.

Unlike ordinary mathematical functions, Φ(a1, a2) yields a1 if control reaches block 4 along the edge $2 \to 4$, and yields a2 if control comes in on edge $3 \to 4$.

8.3.1 How does the ϕ-function know which edge was taken?

If we must execute the program, or translate it to executable form, we can "implement" the Φ-function using a move instruction on each incoming edge as shown in Figure 32. However, in many cases, we simply need the connection of uses to definitions and don't need to "execute" the Φ-functions during optimization. In these cases, we can ignore the question of which value to produce.



(a) Original code

(b) Code after moving instruction.

Figure 32: Implementing Φ-function

8.4 Converting to SSA form

The algorithm for converting a program to SSA form is roughly as follows:

- 1. adds Φ functions for the variables, and then
- 2. renames all the definitions and uses of variables using subscripts.

8.4.1 Trivial SSA

Trivial SSA form is based on a simple observation: Φ functions are only needed for variables that are "live" after the Φ function.

- Each assignment generates a fresh variable.
- At each join point insert Φ for all live variables.

Trivial SSA will generate some useless Φ functions. An example is shown in Figure 33. So a Φ-function is not needed for every variable at each point.

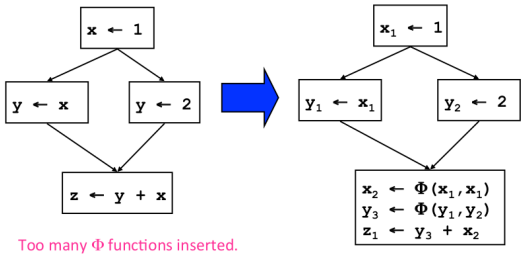


Figure 33: $x2 \leftarrow \Phi(x1, x1)$ is useless because $x2$ is equal to $x1$.

8.4.2 Minimal SSA

Minimal SSA is an updated version compared to trivial SSA.

- Each assignment generates a fresh variable.
- At each join point insert Φ for all live variables with multiple outstanding defs.

8.4.3 Path-convergence criterion

There should be a Φ-function for variable a at node z of the flow graph exactly when all of the following are true:

- 1. There is a block x containing a definition of a,
- 2. There is a block y (with $y \neq x$) containing a definition of a,
- 3. There is a nonempty path P_{xz} of edges from x to z,
- 4. There is a nonempty path P_{yz} of edges from y to z,
- 5. Paths P_{xz} and P_{yz} do not have any node in common other than z, and
- 6. The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.

We consider the start node to contain an implicit definition of every variable, either because the variable may be a formal parameter or to represent the notion of a ← uninitialized without special cases. A Φ-function itself counts as a definition of a, so the path-convergence criterion must be considered as a set of equations to be satisfied. As usual, we can solve them by iteration as shown in 5.

Algorithm 5 Iterated path-convergence criterion

while there are nodes x, y, z satisfying conditions 1–5 and
z does not contain a Φ-function for a **do**
    insert a ← Φ(a, a, . . . , a) at node Z
end while

8.4.4 Dominance property of SSA form

The iterated path-convergence algorithm for placing Φ-functions is not practical, since it would be very costly to examine every triple of nodes x, y, z, and every path leading from x and y. A much more efficient algorithm using the dominator tree of the flow graph as shown in Figure 34.

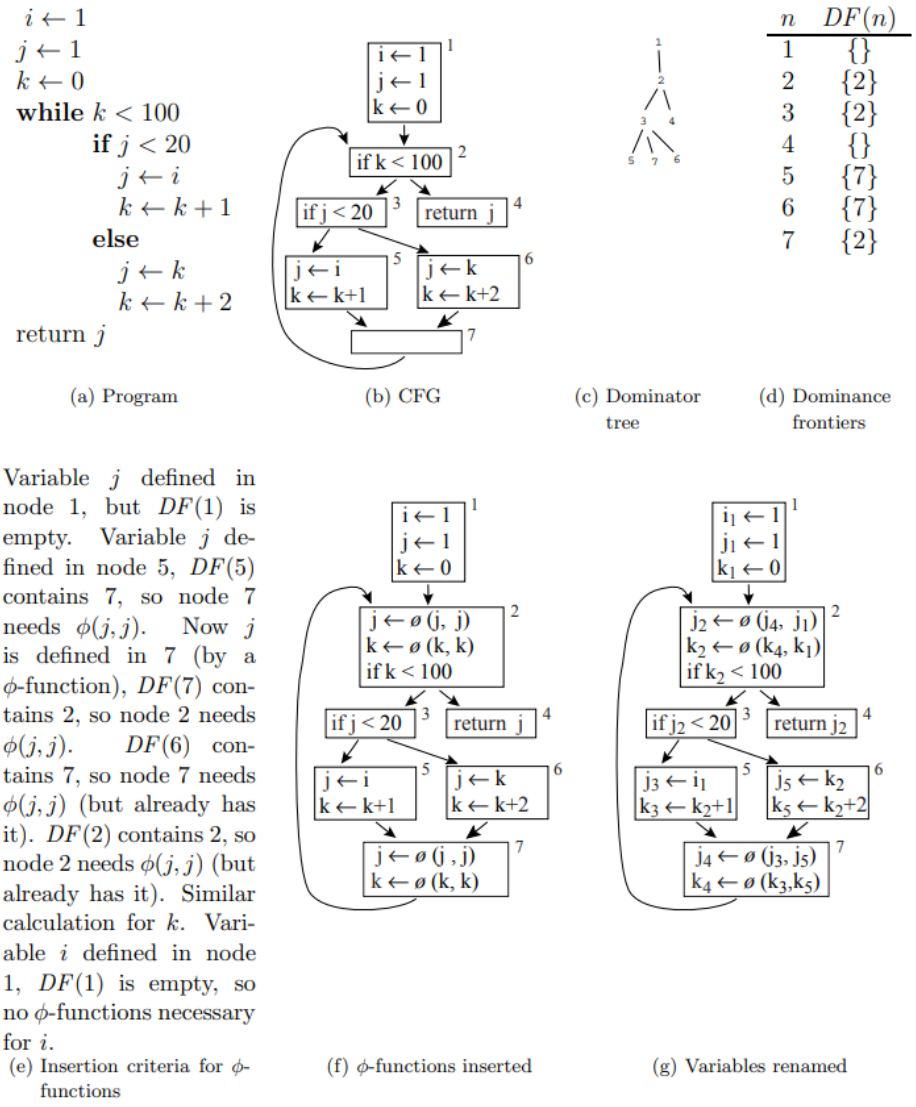


Figure 34: Conversion of a program to static single-assignment form. Node 7 is a postbody node, inserted to make sure there is only one loop edge; such nodes are not strictly necessary but are sometimes helpful.

Strictly dominance

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first.

Dominance

x dominates w (x dom w) iff x sdom w or x = w.

$$\mathrm{Dom}(n) = \begin{cases} \{n\} & \text{if } n = n_0 \\ \{n\} \cup \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p) \right) & \text{if } n \neq n_0 \end{cases}$$

Dominance tree

x sdom w iff x is a proper ancestor of w.

Dominance Frontier

The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

$$F(x) = \{w | \texttt{x dom pred(w) AND !(x sdom w) } \}$$

An essential property of static single assignment form is that definitions dominate uses; more specifically,

- If x is the ith argument of a Φ-function in block n, then the definition of x dominates the ith predecessor of n.
- If x is used in a non-Φ statement in block n, then the definition of x dominates n

Dominance Property of SSA

In SSA,

- If x i is used in $x \leftarrow \Phi(..., x_i, ...)$, then $BB(x_i)$ dominates ith predecessor of $BB(\Phi)$
- If x is used in $y \leftarrow ...x...$,then BB(x) dominates BB(y)

Dominance frontier criterion. Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a Φ-function for a.

Iterated dominance frontier. Since a Φ-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need Φ-functions.

Theorem. The iterated dominance frontier criterion and the iterated path convergence criterion specify exactly the same set of nodes at which to put Φ-functions

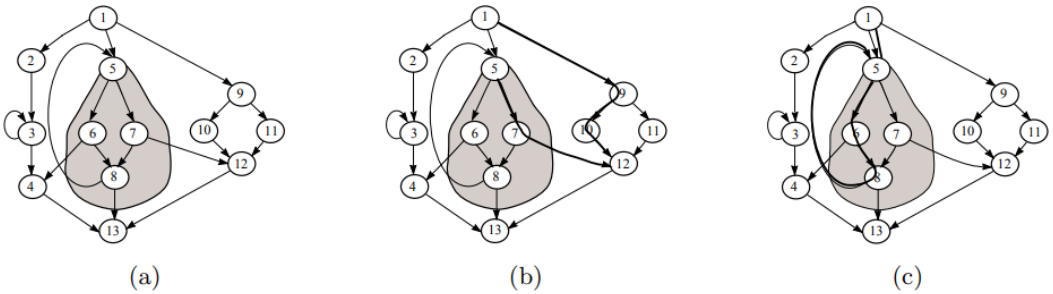


Figure 35: Node 5 dominates all the nodes in the grey area. (a) Dominance frontier of node 5 includes the nodes (4, 5, 12, 13) that are targets of edges crossing from the region dominated by 5 (grey area including node 5) to the region not strictly dominated by 5 (white area including node 5). (b) Any node in the dominance frontier of n is also a point of convergence of nonintersecting paths, one from n and one from the root node. (c) Another example of converging paths $P_{1,5}$ and $P_{5,5}$.

Proof

The sketch of a proof that shows if w is in the dominance frontier of a definition, then it must be a point of convergence.
Suppose there is a definition of variable a at some node n (such as node 5 in Figure 35b), and node w (such as node 12 in Figure 35b) is in the dominance frontier of n. The root node implicitly contains a definition of every variable, including a. There is a path P_{rw} from the root node (node 1 in Figure 35) to w that does not go through n or through any node that n dominates; and there is a path P_{nw} from n to w that goes only through dominated nodes. These paths have w as their first point of convergence.

8.5 Computing the dominance frontier

To insert all the necessary Φ-functions, for every node n in the flow graph we need DF[n], its dominance frontier. Given the dominator tree, we can efficiently compute the dominance frontiers of all the nodes of the flow graph in one pass. We define two auxiliary sets

- $DF_{local}[n]$ The successors of n that are not strictly dominated by n;
- $DF_{up}[n]$ Nodes in the dominance frontier of n that are not dominated by n's immediate dominator.

The dominance frontier of n can be computed from $DF_{local}[n]$ and $DF_{up}[n]$

$$DF[n] = DF_{\text{local}}\,[n] \quad \cup \bigcup_{c \in \text{ children } [n]} DF_{\text{up}}\,[c]$$

where children[n] are the nodes whose immediate dominator (idom) is n.

To compute $DF_{local}[n]$ 6 more easily (using immediate dominators instead of dominators), we use the following theorem: $DF_{local}[n]$ = the set of those successors of n whose immediate dominator is not n. The following computeDF function should be called on the root of the dominator tree (the start node of the flow graph). It walks the tree computing DF[n] for every node n: it computes $DF_{local}[n]$ by examining the successors of n, then combines $DF_{local}[n]$ and (for each child c) $DF_{up}[n]$.a

Algorithm 6 computeDF

```
S ← {}
for each node y in succ[n] do                    ▷ This loop computes DF_local[n]
    if idom(y) ≠ n then
        S ← S ∪ {y}
    end if
end for
for each child c of n in the dominator tree do
    computeDF[c]
    for each element w of DF[c] do               ▷ This loop computes DF_up[n]
        if n does not dominate w then
            S ← S ∪ {w}
        end if
    end for
end for
```

This algorithm is quite efficient. It does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. Although there are pathological graphs in which most of the nodes have very large dominance frontiers, in most cases the total size of all the DFs is approximately linear in the size of the graph, so this algorithm runs in "practically" linear time.

8.6 Inserting Φ-functions

Starting with a program not in SSA form, we need to insert just enough Φ-functions to satisfy the iterated dominance frontier criterion. To avoid re-examining nodes where no Φ-function has been inserted, we use a work-list algorithm.

Algorithm7 starts with a set V of variables, a graph G of controlflow nodes – each node is a basic block of statements – and for each node n a set A_{orig}[n] of variables defined in node n. The algorithm computes $A_\Phi[a]$, the set of nodes that must have Φ-functions for variable a. Sometimes a node may contain both an ordinary definition and a Φ-function for the same variable; for example, in Figure 35b, a $\in A_{orig}[2]$ and $2 \in A_\Phi[a]$.

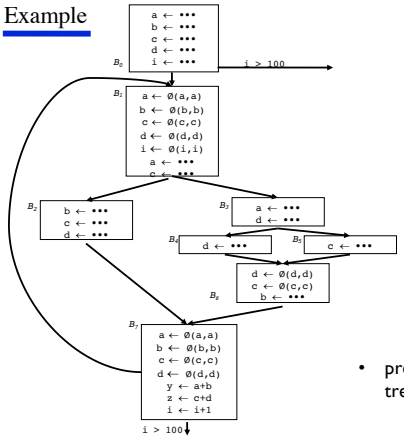
This algorithm does a constant amount of work (a) for each node and edge in the control-flow graph, (b) for each statement in the program, (c) for each element of every dominance frontier, and (d) for each inserted Φ-function. For a program of size N, the amounts a and b are proportional to N, c is usually approximately linear in N. The number of inserted Φ-functions (d) could be N^2 in the worst case, but empirical measurement has shown that it is usually proportional to N. So in practice, Algorithm 7 runs in approximately linear time.

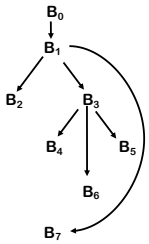
8.7 Renaming the variables

After the Φ-functions are placed, we can walk the dominator tree, renaming the different definitions (including Φ-functions) of variable a to a1, a2, a3 and so on. Rename each use of a to use the closest definition d of a that is above a in the dominator tree. Algorithm renames all uses and definitions of variables, after the Φ-functions have been inserted by Algorithm 8. In traversing the dominator tree, the algorithm "remembers" for each variable the most recently defined version of each variable, on a separate stack for each variable. Although the algorithm follows the structure of the dominator tree – not the flow graph – at each node in the tree it examines all outgoing flow edges, to see if there are any Φ-functions whose operands need to be properly numbered.

8.7.1 Example

Example

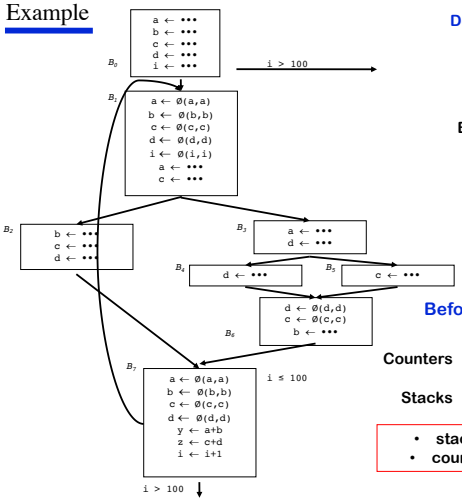




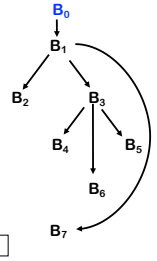
Dominance Tree

- pre-order traversal of the dominator tree: {0,1,2,3,4,5,6,7}

Example



Dominance Tree

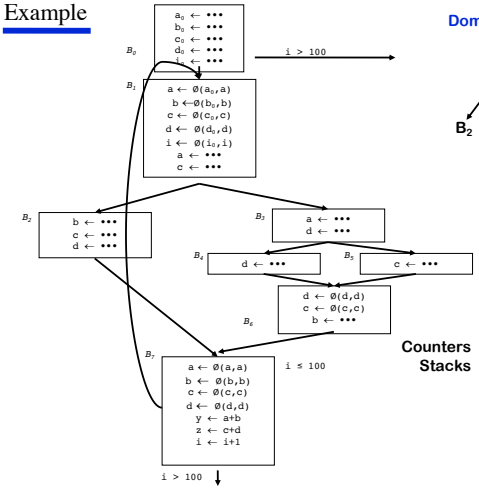


Before processing B_0

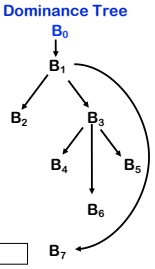
	a	b	c	d	i
Counters	0	0	0	0	0
Stacks					

- stacks for uses of variables
- counters for new definitions

Example



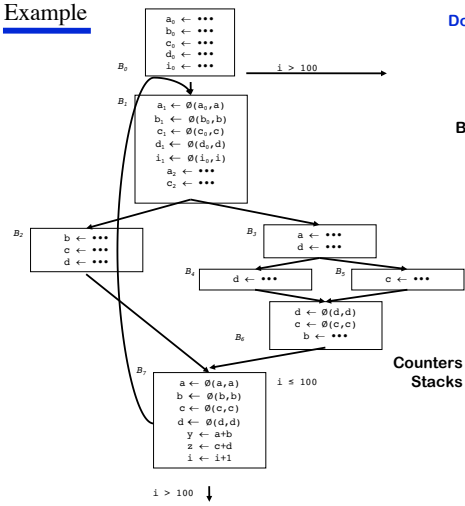
Dominance Tree



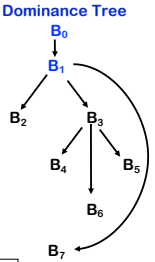
End of B_0

	a	b	c	d	i
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

Example



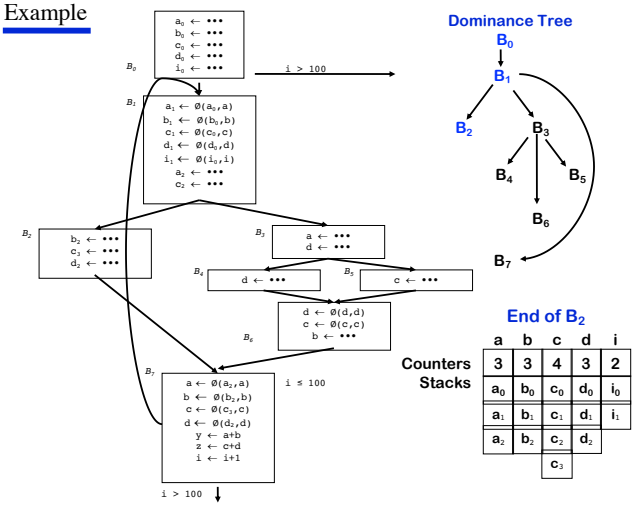
Dominance Tree



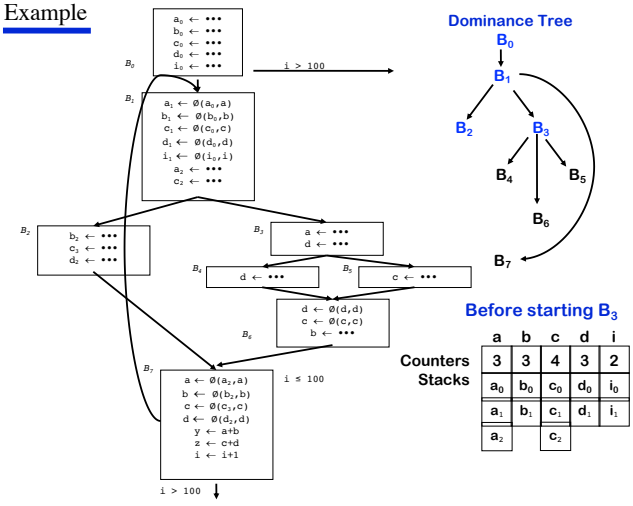
End of B_1

	a	b	c	d	i
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

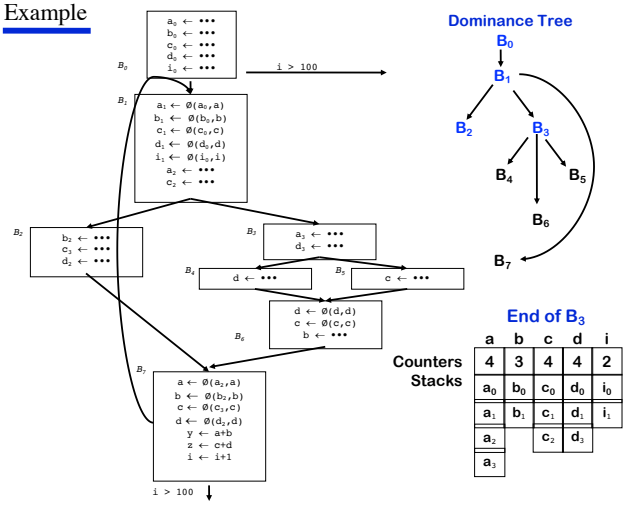
Example



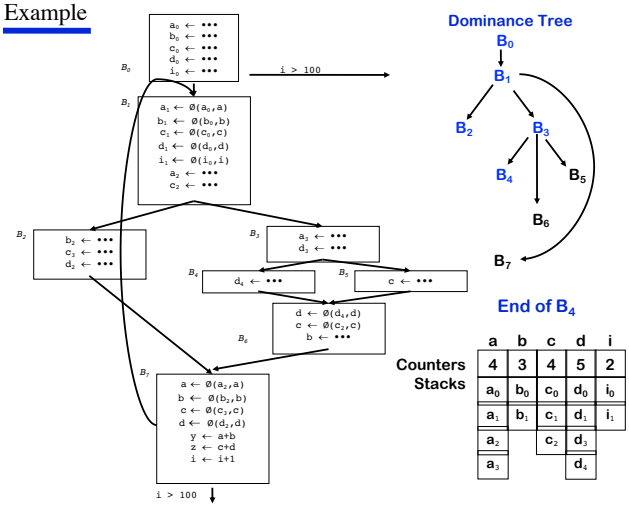
Example



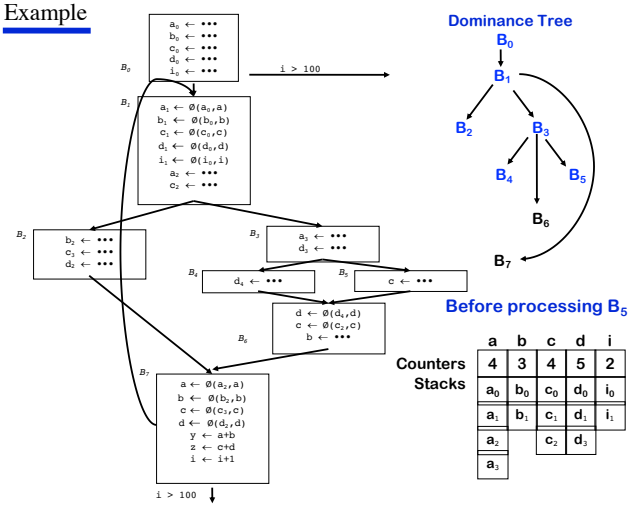
Example



Example



Example

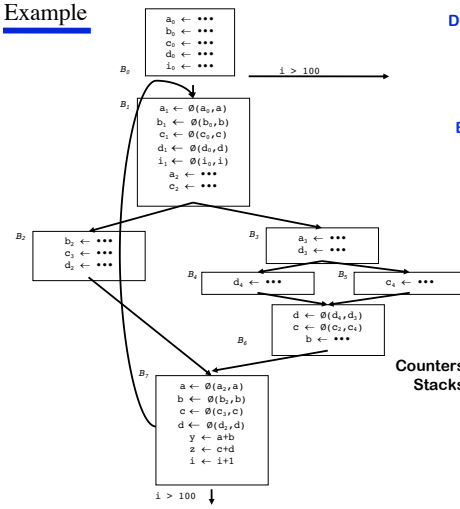


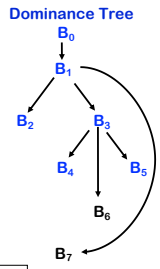
Before processing B_5

Counters
Stacks

a	b	c	d	i
4	3	4	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



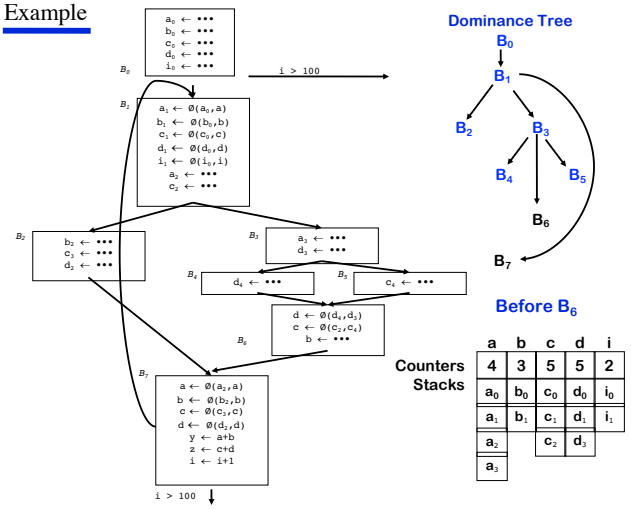


End of B_5

Counters
Stacks

a	b	c	d	i
4	3	5	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3		c_4		

Example

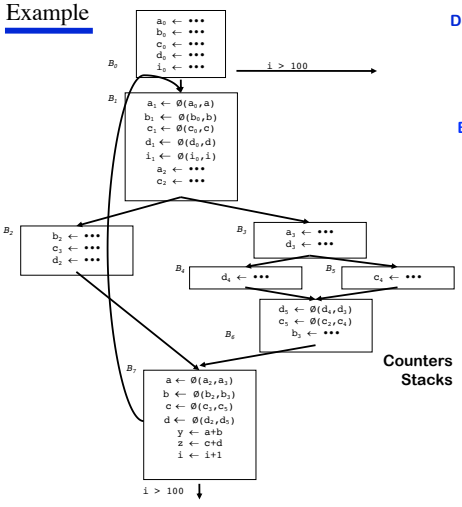


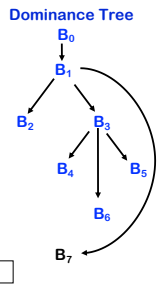
Before B_6

Counters
Stacks

a	b	c	d	i
4	3	5	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



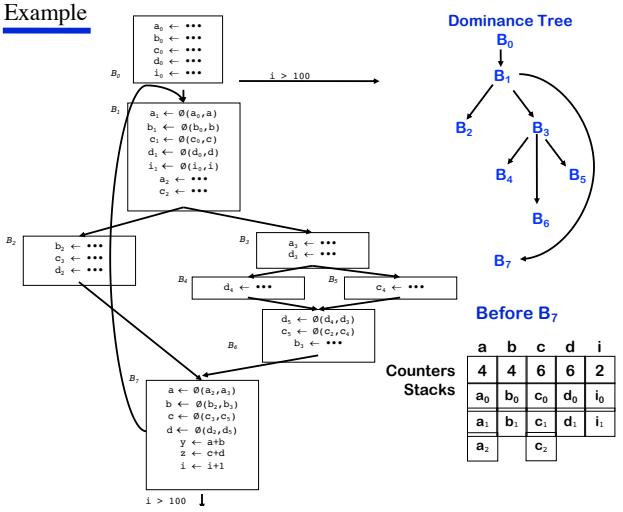


End of B_6

Counters
Stacks

a	b	c	d	i
4	4	6	6	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_3	c_2	d_3	
a_3		c_5	d_5	

Example

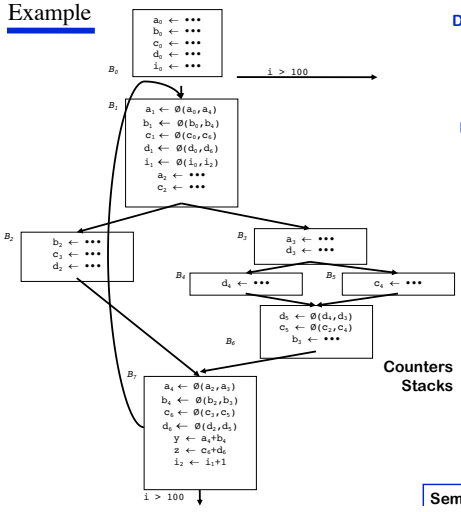


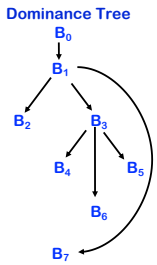
Before B_7

Counters / Stacks

a	b	c	d	i
4	4	6	6	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2		

Example





End of B_7

Counters / Stacks

a	b	c	d	i
5	5	7	7	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_4	c_2	d_6	i_2
a_4		c_6		

Semi-pruned SSA, done!

Algorithm 7 Place-Φ-Functions

```
for each node n do
    for each variable a in A_orig[n] do
        defsites[a] ← defsites[a] ∪{n}
    end for
end for
for each variable a do
    W ← defsites[a]
    while W not empty do
        remove some node n from W
        for each y in DF[n] do
            if y ∉ A_Φ[a] then
                insert the statement a ← Φ(a, a, . . . , a) at the top of block y, where the Φ-function
has as many arguments as y has predecessors
                A_Φ[a] ← A_Φ[a] ∪ {y}
                if a ∉ A_orig[y] then
                    W ← W ∪ {y}
                end if
            end if
        end for
    end while
end for
```

8.8 Edge Splitting

Some analyses and transformations including reverse transformation from SSA back into a normal form are simpler if there is never a controlflow edge that leads from a node with multiple successors to a node with multiple predecessors. To give the graph this unique successor or predecessor property, we perform the following transformation: For each control-flow edge a ← b such that a has more than one successor and b has more than one predecessor, we create a new, empty controlflow node z, and replace the a ← b edge with an a ← z edge and a z ← b edge.

An SSA graph with this property is in edge-split SSA form. Figure 31 illustrates edge splitting. Edge splitting may be done before or after insertion of Φ-functions.

Algorithm 8 Renaming variables.

```
Initialization:
for each variable a do
    Count[a] ← 0
    Stack[a] ← empty
    push 0 onto Stack[a]
end for
Rename(n)
for each statement S in block n do
    if S is not a Φ-function then
        for each use of some variable x in S do
            i ← top(Stack[x])
            replace the use of x with x_i in S
        end for
    end if
    for each definition of some variable a in S do
        Count[a] ← Count[a]+1
        i ← Count[a]
        push i onto Stack[a]
        replace definition of a with definition of a_i in S
    end for
end for
for each successor Y of block n, do
    Suppose n is the jth predecessor of Y
    for each Φ-function in Y do
        suppose the jth operand of the Φ-function is a
        i ← top(Stack[a])
        replace the jth operand with a_i
    end for
end for
for each child X of n do
    Rename(X)
end for
for each definition of some variable a in the original S do
    pop Stack[a]
end for
```

9 SSA-Style optimizations

9.1 Constant Propagation

notes

- If v ← c , replace all uses of v with c
- If v ← Φ (c,c,c) (each input is the same constant), replace all uses of v with c

Algorithm 9 SSA-CP

```
W ← list of all defs
while !W.isEmpty() do
    Stmt S ← W.removeOne()
    if (S has form v ← c) or (S has form v ← Φ(c,...,c)) then
        delete S
        for each stmt U that uses v do
            replace v with c in U
            W.add(U)
        end for
    end if
end while
```

9.2 Conditional Constant Propagation

In this section, we want to talk something about Conditional Constant Propagation. Before we talk about it, we need to know something about the history of Constant Propagation. There are four algorithms for determining constants. They are described in order of increasing power; each algorithm finds at least the constants found by the previous algorithm. These algorithms are among the simplest, fastest, and most powerful global constant propagation algorithms known. The first three algorithms are reformulations of the work of others; the fourth is new and contains the best features of each of the previous three. Figure 36 shows the relationship among the four algorithms.

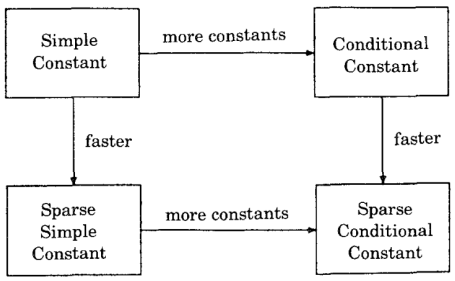


Figure 36: Relationship among the four constant propagation algorithms

The first algorithm, Simple Constant (SC), was developed by Kildall[5]. Kildall was among the first to describe the constant propagation problem and to give an algorithmic solution.

The second algorithm, Sparse Simple Constant (SSC), is an easily understood reformulation of an algorithm developed by Reif and Lewis[6]. This algorithm uses a data structure called the static single assignment graph (SSA graph). The SSA graph is a variant of the global value graph of Reif and Lewis, which in turn is based on the p-graph of Shapiro and Saint. The SSA graph allows this algorithm to find a class of constants equivalent to those of SC, yet the algorithm is faster than SC by a factor proportional to the number of variables in the program. Indeed, the speedup can be proportional to the product of the number of variables in the program and the number of edges in the program flow graph. It is unfortunate that this algorithm was not recognized for many years, since it works in time linear in the size of the SSA graph.

The third algorithm, Conditional Constant (CC), is a variant of Wegbreit's Algorithm[7]. CC discovers all constants that can be found by evaluating all conditional branches with all constant operands, but it uses the same input data structures and is asymptotically as slow as SC. The attraction of CC is that it propagates the values in such a way that when conditional branches are found to have a constant conditional expression, the search for constants can ignore parts of the program that are never executed. The algorithm does unreachable code elimination in combination with constant propagation. The first benefit of this approach is that the algorithm may run faster than SC, since it need not evaluate the sections of the program that are never executed. A second benefit is that values created in the unreachable areas cannot possibly kill potential constants, and thus CC can find more constants than can SC.

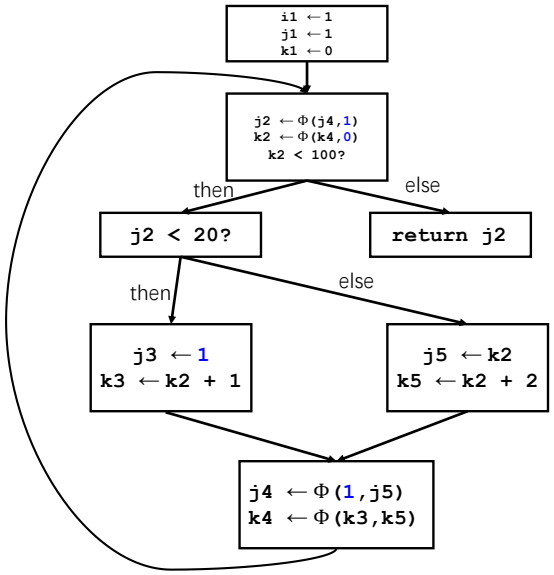
The fourth algorithm, Sparse Conditional Constant (SCC) finds the same class of constants as CC, yet has the same speedup over CC as SSC has over SC.

Wegman and Zadeck's Sparse Conditional Constant (SCC) algorithm was used to find constant expressions, constant conditions, and unreachable code [8]. The output of the SCC algorithm is an association of variables to one of $\{\bot, c, \top\}$, where $\bot$ marks a variable that can hold different values at different times, and $\top$ means the variable is not executed. In addition, every flow-graph node (corresponding to a quadruple) is marked as executable or non-executable. We then walk the flow-graph, eliminating dead-code (quadruples marked non-executable), replacing constant variables with their values, and changing constant conditional branches to goto statements.

notes

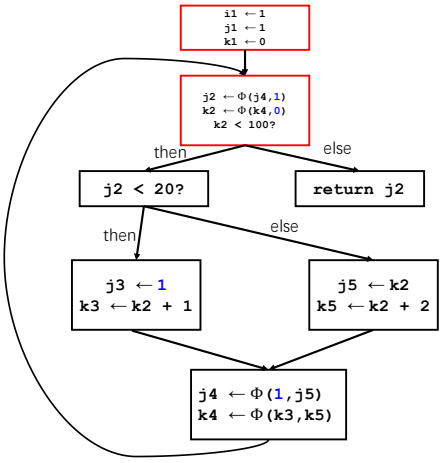
- Assume all blocks unexecuted until proven otherwise
- Assume all variables are not executed (only with proof of assignment of a non-constant value do we assume not constant)

9.2.1 Example



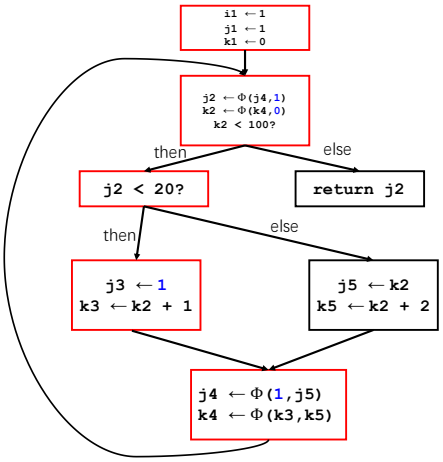
variable	value
i1	T
j1	T
j2	T
j3	T
j4	T
j5	T
k1	T
k2	T
k3	T
k4	T
k5	T

Figure 37: Original code. The black block is marked as unexecuted



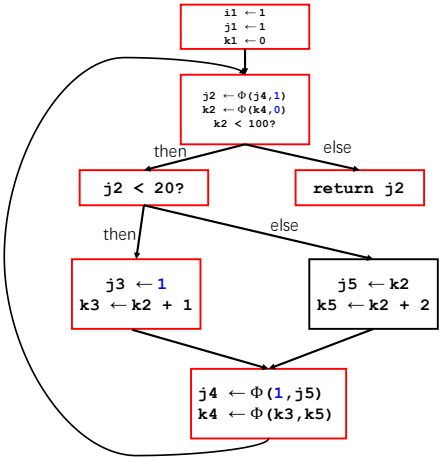
variable	value
i1	1
j1	1
j2	1
j3	T
j4	T
j5	T
k1	0
k2	0
k3	T
k4	T
k5	T

Figure 38: The read block is marked as executed. After walking the first two blocks, the value is shown above.



variable	value
i1	1
j1	1
j2	1
j3	1
j4	1
j5	T
k1	0
k2	0
k3	1
k4	1
k5	T

Figure 39: After walking 5 blocks.



variable	value
i1	1
j1	1
j2	1
j3	1
j4	1
j5	T
k1	0
k2	⊥
k3	1
k4	1
k5	T

Figure 40: Now k2 is ⊥, so the `return j2` is reachable.

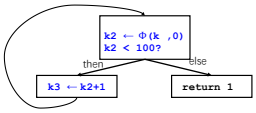


Figure 41: Code after applied SCC.

9.3 Copy Propogation

notes

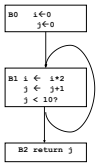
- delete x ← Φ (y,y,y) and replace all x with y
- delete x ← y and replace all x with y

9.4 Aggressive Dead Code Elimination

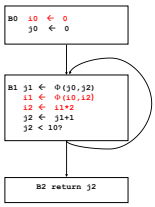
We can easily define the standard algorithm 10 below, but this algorithm may leave zombies. Look at the example in Figure 42

Algorithm 10 Dead Code Elimination

```
W ← list of all defs
while !W.isEmpty do
    Stmt S ← W.removeOne
    if |S.users|! = 0 then
        continue
    end if
    if S.hasSideEffects() then
        continue
    end if
    for def in S.operands.definers do
        def.users ← def.users - {S}
        if |def.users| == 0 then
            W ← W UNION {def}
        end if
    end for
    delete S
end while
```



(a) Original code. We can easily find that instructions relating to i are dead and can be eliminated.



(b) SSA format code. Since there is a circle use chain so we can not remove instructions relating to i because i_1 uses i_2, and i_2 uses i_1

.

Figure 42: An example to illustrate standard DCE can leave zombies.

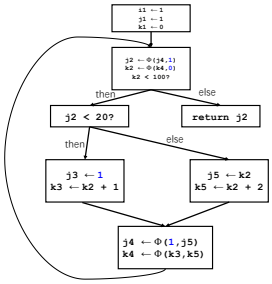
So instead of assuming everything is live until proven dead, we go another way: assuming everything is dead until proven live shown in algorithm 11.

Algorithm 11 Aggressive Dead Code Elimination

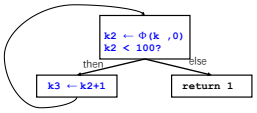
```
function INIT
    mark as live all stmts that have side-effects:
        I/O
        stores into memory
        returns
        calls a function that MIGHT have side-effects
    As we mark S live, insert S.operands.definers into W
    while |W| > 0 do
        S ← W.removeOne()
        if (S is live) then
            continue
        end if
        mark S live, insert S.operands.definers into W
    end while
end function
```

9.4.1 Problems within algorithm 11

After Aggressive Dead Code Elimination applied to function shown in 43a, there is only one `return` statement left. However, control flow is undecidable in general, so possibly the loop in 43b will iterate indefinitely and the `return` instruction will never be executed. The problem here is we simply mark the branch statement dead.



(a) Original code in SSA format.



(b) After CCP

(c) After ADCE

Figure 43: An example to illustrate the algorithm 11 has a problem.

Also when we apply this algorithm to 42, we can find that `j2 < 10` is marked dead which is wrong. Of course, we can simply mark all branches live in the initialize stage, but this is not the ideal solution.

Now we need to carefully consider which conditional branches need to be marked live.

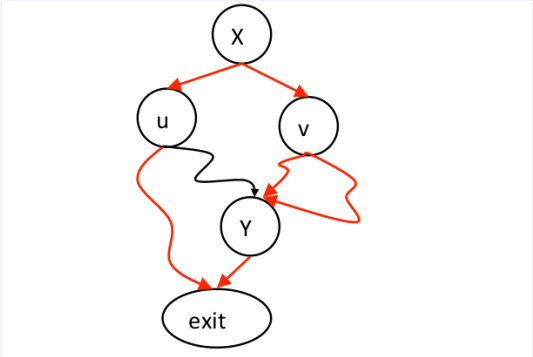
9.4.2 Control Dependence

control dependence

Y is control-dependent on X if

- X branches to u and v
- $\exists$ a path u $\rightarrow$ exit which does not go through Y
- $\forall$ paths v $\rightarrow$ exit go through Y

This means X casn determine whether or not Y is executed.



9.4.3 Aggressive Dead Code Elimination(Fixed Version)

So we make a little modification 12. When we mark S is live, we should also mark live those conditional branches upon which S is control dependent.

Algorithm 12 Aggressive Dead Code Elimination(Fixed Version)

```
function INIT
    mark as live all stmts that have side-effects:
        I/O
        stores into memory
        returns
        calls a function that MIGHT have side-effects
    As we mark S live, insert S.operands.definers into W
    S.CD^-1 into W
    while |W| > 0 do
        S ← W.removeOne()
        if (S is live) then
            continue
        end if
        mark S live, insert S.operands.definers into W
        S.CD^-1 into W
    end while
end function
```

9.4.4 Finding the Control Dependence Graph

- Construct CFG
- Add entry node and exit node
- Add (entry, exit) edge
- Create G′, the reverse CFG
- Compute D-tree in G′ (post-dominators of G)
- Compute $DF'_G(y)$ for all y ∈ G′ (post-DF of G)
- Add (x,y)∈ G to CDG if x ∈ $DF'_G(y)$

So let us calculate the control dependence for Figure 42a which is shwon in Figure 44. Since Block1 is control dependent on Block1, so the conditional branch in Block1 `j2 < 10 ?` should be marked live now.

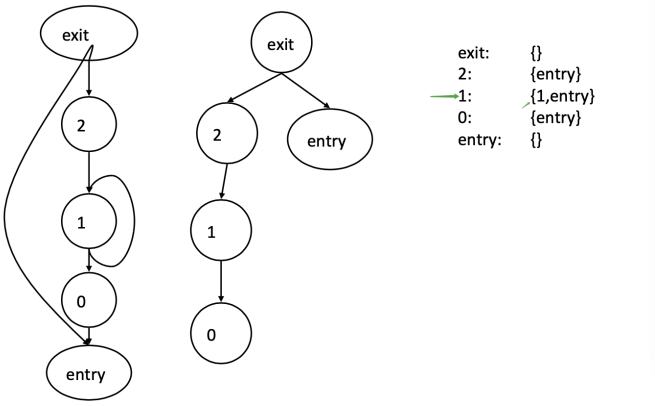


Figure 44: From left to right are G′, post-dominators of G and post-DF of G respectively.

10 The LLVM project

LLVM is an open-source framework providing a modern collection of modular and reusable compiler and toolchain technologies [20]. The project stemmed from the work of Chris Lattner, who first implemented core elements of LLVM to support the research of his master thesis in 2002 [26]. One of the key strengths of LLVM is that it faces active development from an expert community of contributors, and is widely used across the industry and academia alike [11]. The project received the ACM Software System Award in 2012 [4] as an acknowledgement of its contribution to compiler research and implementation.

LLVM officially acknowledges more than 10 main sub-projects [20], which range in diversity from a debugger [9] to a symbolic execution tool [8]. In addition to these, the official website presents a long list of miscellaneous projects that are based on components of the LLVM infrastructure [19].

All projects which are part of the LLVM ecosystem are built upon the core libraries, which are arguably the centrepiece of LLVM. They host the source- and target-independent optimizer (Section 3.3), the implementation of the LLVM intermediate representation (Section 3.2), and a suite of command-line tools useful for code manipulation (Section 3.4). The core libraries also implement various back-end passes that translate IR to machine code for different platforms (x86, PowerPC, Nvidia GPUs). For building a complete compiler for a source language, the LLVM core libraries readily provide the optimizing pass and code generation for common architectures, the frontend being the only missing component. Clang is by far the most prominent front-end implemented in LLVM [5], and it targets the family of C languages (C/C++ and Objective-C/C++). Coupled with the core libraries, Clang is a powerful compiler producing high-performance code, and positions itself as a direct competitor to both gcc and the Intel compiler.

11 Loop Invariant Computation and Code Motion

Loop-Invarian Code Motion (LICM) recognizes computations within a loop that produce the same result each time the loop is executed. These computations are called loop-invariant code and can be moved outside the loop body without changing the program semantics. The positive effects of LICM are:

- The shifted loop invariants exhibit a reduced execution frequency.
- The transformation may shorten variable live ranges leading to a decreased register pressure. This circumstance may in turn reduce the number of required spill code instructions.
- Moving code outside a loop reduces the loop's size which may be beneficial for the I-cache behavior since more loop code can reside in the cache.

Besides these positive effects on the code, LICM may also degrade performance. This is mainly due to two reasons. First, the newly created variables to store the loop-invariant results outside the loop, may increase the register pressure in the loops since their live ranges span across the entire loop nest. As a result, possibly additional spill code is generated. Second, LICM might lengthen other paths of the control flow graph. This situation can be observed if the invariants are moved from a less executed to a more frequently executed path, e.g., moving instructions above a loop's zero-trip test.

11.1 Finding natrual loops

Not every cicle is a loop in CFG. From a intuitive perspective, a loop must has a single entry and edges must from at least one circle.

Back Edge

A back edge is an arc t→h whose head h dominates its tail t

Natural Loop

The natural loop of a back edge t→h is the smallest set of nodes that includes t and h, and has no predecessors outside the set, except for the predecessors of the header h.

Reducible

A flow graph is reducible if every retreating edge in any DFST (Deep-First Spanning Tree) for that flow graph is a back edge.

Testing reducibility Take any DFST for the flow graph, remove the back edges, and check that the result is acyclic.

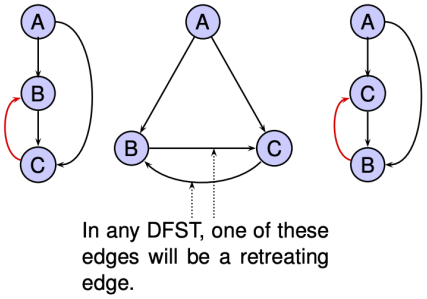


Figure 45: Example: Nonreducible Graph

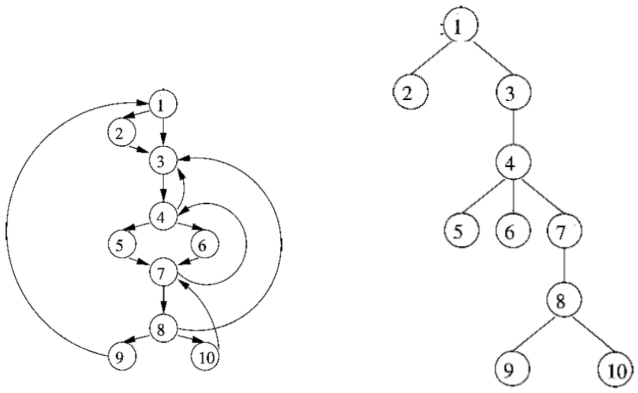
11.2 Algorithm to Find Natural Loops

11.2.1 Step 1. Finding Dominators

We can formulate this as Data Flow Analysis problem. Since Node d dominates node n in a graph (d dom n) if every path from the start node to n goes through d. So if d dom n iff iff d dom p for all pred p of n.

Direction	Forward
Values	Basic Blocks
Meet operator	$\cap$
Top(T)	Universal Set
Bottom	ϕ
Boundary condition for entry node	ϕ
Initialization for internal nodes	T
Finited escending chain?	✓
Transferfunction	OUT $[\mathbf{b}] = \{\mathbf{b}\} \cup \left(\cap_{\{p=pred(b)\}} \text{OUT } [\mathbf{p}]\right)$
Monotone&Distributive?	✓

With rPostorder, most flow graphs (reducible flow graphs) converge in 1 pass.



(a) CFG.

(b) Dominance Tree of the corresponding CFG.

Figure 46: An example of Dominance tree.

11.2.2 Step 2. Finding Back Edges

Depth-first spanning tree Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree. We categorize edges in CFG as follows:

- Forward edges (node to proper descendant).
- Retreating edges (node to ancestor).
- Cross edges (between two nodes, neither of which is an ancestor of the other.)

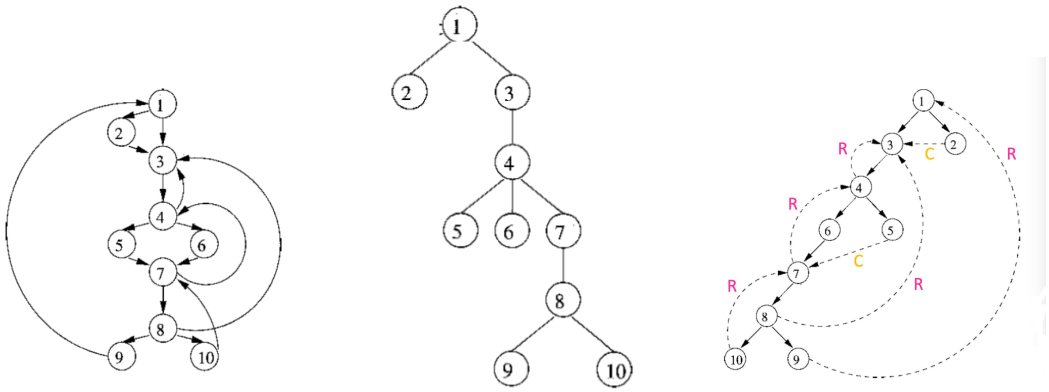
This is something difficult to understand. Let's make it simpler. We can number each node when we visit it. So each edge should be satisfied the following property:

Forward/Advancing edges $n_1 \to n_2$	$\text{num}(n_1) < \text{num}(n_2)$ and n_1 is ancestor of n_2
Cross edges $n_1 \to n_2$	$\text{num}(n_1) > \text{num}(n_2)$ and neither n_1 is ancestor of n_2 nor n_2 is ancestor of n_1
Retreating edges $n_1 \to n_2$	$\text{num}(n_1) > \text{num}(n_2)$ and n_2 is ancestor of n_1

Of these edges, only retreating edges go from high to low in DF order.

Algorithm

- Perform a depth first search
- For each retreating edge $t \to h$, check if h is in t′s dominator list



(a) CFG.

(b) Dominance Tree of the corresponding CFG.

(c) R means back edges and C means cross edge.

Figure 47: An example of Back Eges.

11.2.3 Step 3. Constructing Natural Loops

Algorithm For each back edge $t \to h$:

- delete h from the flow graph
- find those nodes that can reach t (those nodes plus h form the natural loop of $t \to h$)

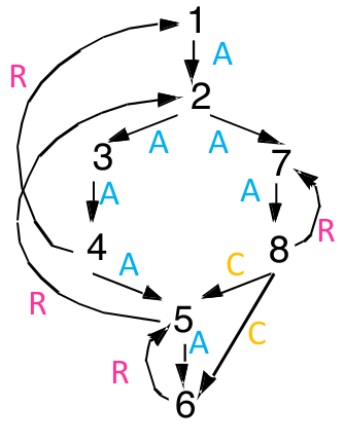


Figure 48: For this flow graph, for back edge $8 \to 7$, natrual loop is { 7,8 }, for back edge $5 \to 2$, natrual loop is { 2,3,4,5,6,7,8 },back edge $4 \to 1$, natrual loop is { 1,2,3,4,5,6,7,8}.

11.3 Inner Loops

If two different loops don't have ths same header, they are either disjoint or one is entirely contained the other (inner loop is the one that contains no other loop.). If two loops share the same header shown in reffig:p65, it is hard to tell which is the inner loop. But we can combine and treat as one loop.

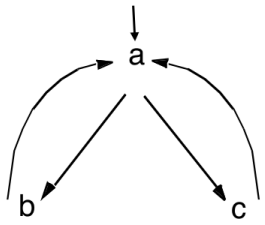


Figure 49: Two loops share the same header.

11.4 Loop-Invariant Computation and Code Motion

Loop-Invariant Computation

A loop-invariant computation is a computation whose value doesn't change as long as control stays within the loop. loop invariant whose operands are defined outside loop or invariant themselves.

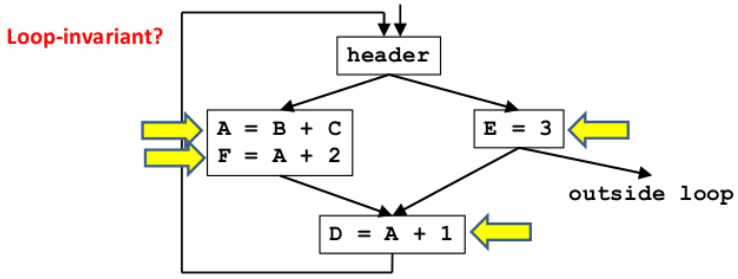


Figure 50: For this CFG, `A = B+C, F=A+2, E=3` are Loop-Invariant Computation, but `D = A + 1` is not.

Not all loop invariant instructions can be moved to preheader.

11.5 LICM Algorithm

- Find invariant expressions
- Conditions for code motion

- Code transformation

11.6 Find invariant expressions

- Compute reaching definitions
- Repeat: mark `A = B + C` as invariant if
  - All reaching definitions of B are outside the loop or there is exactly one reaching definition for B and it is from a loop-invariant statement inside the loop.
  - Check the same for C.
- Code transformation

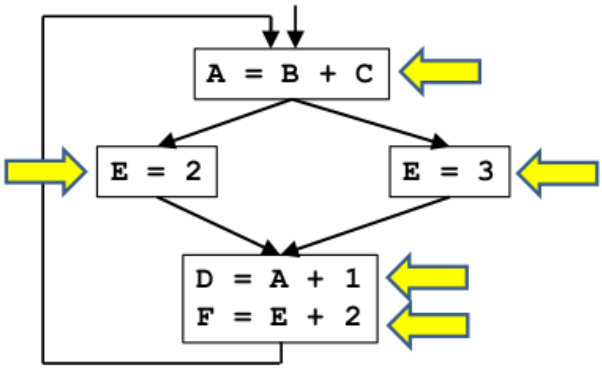


Figure 51: For this CFG, `A = B+C`, `E = 2`, `E = 3`, `D = A + 1` are Loop-Invariant Computation, but `F = E + 2` is not because two definitions of E reach `F = E + 2`.

11.7 Conditions for Code Motion

Algorithm 13 Code Motion Algorithm

Given: a set of nodes in a loop
Compute reaching definitions
Compute loop invariant computation
Compute dominators
Find the exits of the loop (i.e. nodes with successor outside loop)
Candidate statement for code motion:
    loop invariant
    in blocks that dominate all the exits of the loop shown in 52
    assign to variable not assigned to elsewhere in the loop
    in blocks that dominate all blocks in the loop that use the variable assigned shown in 53
Perform a depth-first search of the blocks
    Move candidate to preheader if all the invariant operations it depends upon have been moved

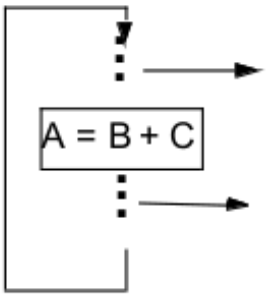


Figure 52: It is not safe to move `A = B + C` outside the loop because we can jump of the loop before executing `A = B + C`

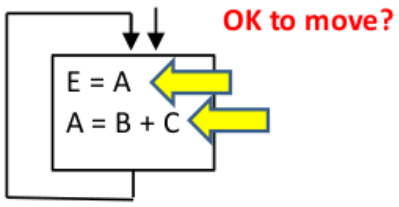


Figure 53: It is not safe to move `A = B + C` outside the loop because if so, the first time we enter the loop `E = A` will not the same as before.

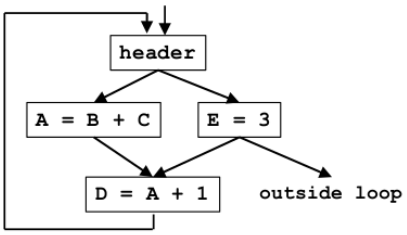


Figure 54: Only `E = 3` can be moved outside the loop.

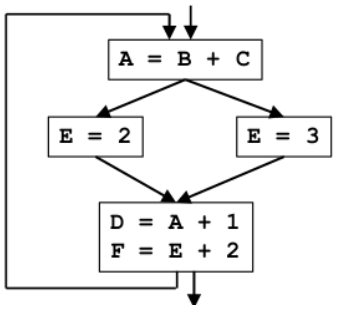


Figure 55: Only `A = B + C , D = A + 1` can be moved outside the loop.

11.8 More Aggressive Optimizations

11.8.1 Gamble on: most loops get executed

We can relax constraint of dominating all exits on some cases.

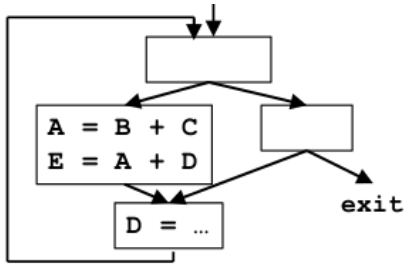


Figure 56: `A = B + C` cannot be removed outside the loop because it doesn't dominat the exit. But if A is not live after the loop, we can actually move it to the preheader. The only thing we need to consider is that this statement can cause an exception.

11.8.2 Landing pads

`while` loop is not very convenient for optimization since it checks before entering the loop. We can use landing pads to solve this.

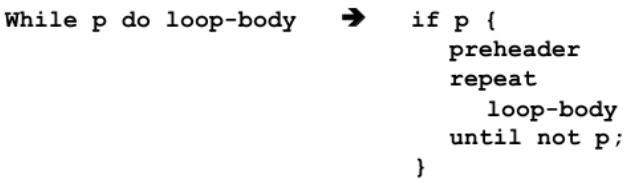


Figure 57: Transforms for while loops.

12 Induction Variables and Strength Reduction

Strength reduction is an optimization technique which substitutes expensive operations with computationally cheaper ones. For example, a very weak strength reduction algorithm can substitute the instruction `b = a * 4` with `b = a << 2`.

12.1 Motivation

Opportunities for strength reduction arise routinely from details that the compiler inserts to implement source-level abstractions. To see this, consider the simple code fragment shown in Figure 84. Figure 58a shows source code and the same loop in a low-level intermediate code. Notice the instruction sequence that begins at the label L2. The compiler inserted this code (with its multiply) as the expansion of A[i]. Figure 58b shows the code that results from applying Strength Reduction, Figure 58c is followed by dead-code elimination. The compiler created a new variable, $t2'$, to hold the value of the expression $i * 4 + A$. Its value is computed directly, by incrementing it with the constant 4, rather than recomputing it on each iteration as a function of i. Strength reduction automates this transformation.

```
for(i=0; i<100; i++)
       A[i] = 0;
```

```
    i = 0
L2: IF i>=100 GOTO L1
    t1 = 4 * i
    t2 = &A + t1
    *t2 = 0
    i = i+1
    GOTO L2
L1:
```

(a) Origianl code.

```
t1' = 0
t2' = &A
L2: IF t1'>=400 GOTO L1
    t1 = t1'
    t2 = t2'
    *t2 = 0
    t1' = t1'+4
    t2' = t2'+4
    GOTO L2

L1:
```

(b) After induction variable substitute.

```
       t2' = &A
       t3' = &A + 400
L2:    IF t2'>=t3' GOTO L1 *t2'= 0
       t2' = t2'+ 4
       GOTO L2
L1:
```


(c) Final code.

Figure 58: An example of strength reduction.

12.2 Definitions

Basic Induction Variable

A basic induction variable (e.g., i as shown in Figure 58a) is a variable X whose only definitions within the loop are assignments of the form: X = X+c or X = X-c, where c is either a constant or a loop-invariant variable.

Induction Variable

An induction variable is either a basic induction variable B, or or a variable defined once within the loop (e.g., t1,t2 as shown in Figure 58a) , whose value is a linear function of some basic induction variable at the time of the definition: $A = c_1 * B + c_2$

The FAMILY of a basic induction variable B is the set of induction variables A such that each time A is assigned in the loop, the value of A is a linear function of B. (e.g., t1, t2 is in family of i as shown in Figure 58a)

12.3 Optimizations

12.3.1 Strength Reduction

Algorithm 14 Strength Reduction Optimizations

A is an induction variable in family of basic induction variable B (i.e., $A = c_1 * B + c_2$)
Create new variable A′
Initialize in preheader A′ $= c_1 * B + c_2$
Track value of B: add after $B = B + x$: $A' = A' + x * c_1$
Replace assignment to A: replace lone $A = \ldots$ with $A = A'$

12.3.2 Optimizing non-basic induction variables

- copy propagation
- dead code elimination

12.3.3 Optimizing basic induction variables

Eliminate basic induction variables used only for calculating other induction variables and loop tests.

Algorithm 15 Optimizing basic induction variables

Select an induction variable A in the family of B, preferably with simple constants ($A = c_1 * B + c_2$).
Replace a comparison such as `if B > X goto L1` with `if (A′ >` $c_1 * X + c_2$`) goto L1` (assuming c 1 is positive)
if B is live at any exit from the loop, recompute it from A′, After the exit, $B = (A' - c2)/c1$

12.4 Further Details

```
k = 0;
for (i = 0; i < n; i++){
       k = k+3 ;
       … = m;
       if(x<y)
         k = k+4
       if(a < b)
         m = 2 * k;
       k = k - 2;
       … = m;
}
```

(a) A more complex example. k and i are both basic induction variables. m is in the family of k.

```
k = 0;
m' = 0;
for (i = 0; i < n; i++){
       k = k+3 ;
       m' = m'+6;
       … = m;
       if(x<y)
         k = k+4
         m' = m'+8;
       if(a < b)
         m = m' ;
       k = k - 2;
       m' = m'-4;
       … = m;
}
```

(b) After induction variable substitute.

Figure 59: A more complex example of strength reduction.

12.5 Finding Induction Variable Families

Let B be a basic induction variable, A is in the family of B if it satisfies one the following conditions

- **Condition C1** A has a single assignment in the loop L of the form A = B*c, c*B, B+c, etc
- **Condition C2** A is in family of B if $D = c_1 * B + c_2$ for basic induction variable B and:
  - Rule 1: A has a single assignment in the loop L of the form A = D*c, D+c, etc
  - Rule 2: No definition of D outside L reaches the assignment to A
  - Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

```
L2: IF i>=100 GOTO L1
    t2 = t1 + 10
    t1 = 4 * i
    t3 = t1 * 8
    i = i + 1
    goto L2
L1:
```

Figure 60: i is abasic induction variable, t1 t2 are in family ofi, but t2 is not because it violates the condition C2 rule 2.

```
L3: IF i>=100 GOTO L1
    t1 = 4 * i
    IF t1 < 50 GOTO L2
    i = i + 2
L2: t2 = t1 + 10
    i = i + 1
    goto L3
L1:
```

Figure 61: i is abasic induction variable, t1 is in the family of i. t2 is not because it violates the Condition2 rule3(some path reaches t2 includes `i = i+1` but some not.).

13 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is a global optimization introduced by Morel and Renvoise[9]. It combines and extends two other techniques: common subexpression elimination and loop-invariant code motion.

An expression is partially redundant at point p if it is redundant along some, but not all, paths that reach p. PRE converts partially-redundant expressions into redundant expressions. The basic idea is simple. First, it uses data-flow analysis to discover where expressions are partially redundant. Next, it solves a data-flow problem that shows where inserting copies of a computation would convert a partial redundancy into a full redundancy. Finally, it inserts the appropriate code and deletes the redundant copy of the expression.

A key feature of PRE is that it never lengthens an execution path. To see this more clearly, consider the example shown in Figure 62. In the fragment on the left, the second computation of `x + y` is partially redundant; it is only available along one path from the if. Inserting an evaluation of `x + y` on the other path makes the computation redundant and allows it to be eliminated, as shown in the right-hand fragment. Note that the left path stays the same length while the right path has been shortened.

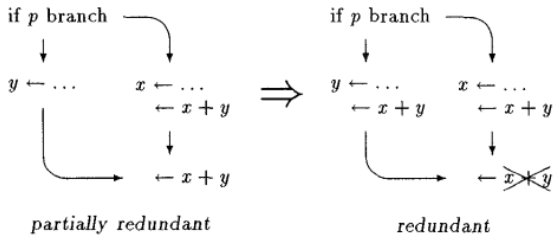


Figure 62

Loop-invariant expressions are also partially redundant, as shown in Figure 63. On the left, `x + y` is partially redundant since it is available from one predecessor (along the back edge of the loop), but not the other. Inserting an evaluation of `x + y` before the loop allows it to be eliminated from the loop body.

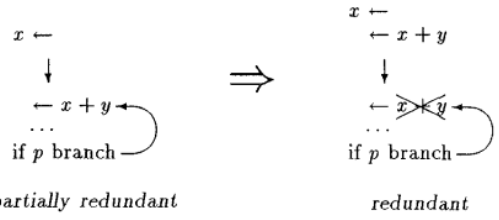


Figure 63

13.1 Finding Partially Available Expressions

For every expression, we can do a dataflow analysis.

Direction	Forward
Meet operator	$\cup$
Lattice	$\{0, 1\}$
Top(T)	0
Bottom	1
Boundary condition for entry node	0
Initialization for internal nodes	T
Finited escending chain?	✓
Transferfunction	$PAVOUT[i] = (PAVIN[i] - KILL[i]) \cup AVLOC[i]$
Monotone&Distributive?	✓
AVLOC	Expression is locally available (AVLOC) if downwards exposed.
KILL	Expression is killed (KILL) if any assignments to operands.

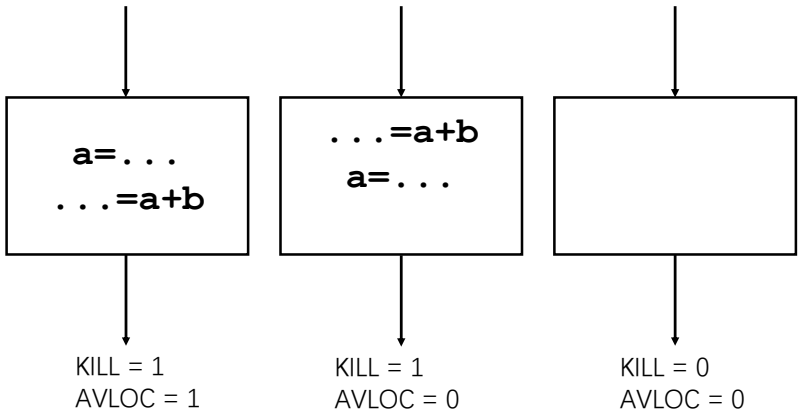


Figure 64: For `a+b`, the result of Partially Available Expressions's transfer function within a basic block.

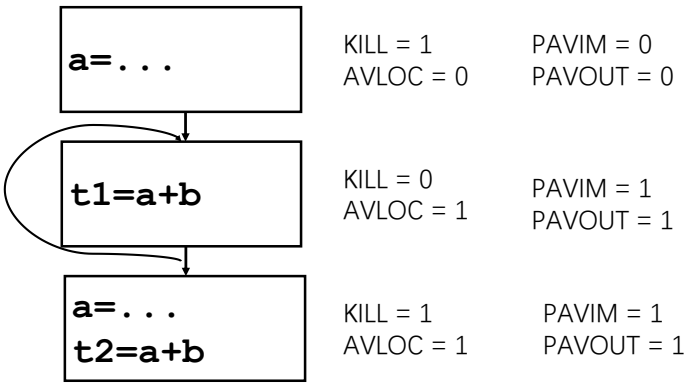


Figure 65: For `a+b`, the result Partially Available Expressions of dataflow analysis.

13.2 Finding Anticipated Expression

For PRE, care must be taken that the hoisting would do no harm: Never introduce a new expression along any path. Otherwise the hoisting will lengthen at least one trace of the program, defying optimality; even worse, if the hoisted instruction throws an exception, the program's semantics change.

Local Anticipability(ANTLOC)

An expression may be locally anticipated in a block i if there is at least one computation of the expression in the block i, and if the commands appearing in the block before the first computation of the expression do not modify its operands.

Direction	backward
Meet operator	$\cap$
Lattice	$\{0, 1\}$
Top(T)	1
Boundary condition for exit node	0
Initialization for internal nodes	T
Finited escending chain?	✓
Transferfunction	$ANTIN[i] = ANTLOC[i] \cup (ANTOUT[i] - KILL[i])$
Monotone&Distributive?	✓
ANTLOC	Expression is locally anticipated(ANTLOC)) is upward exposed.

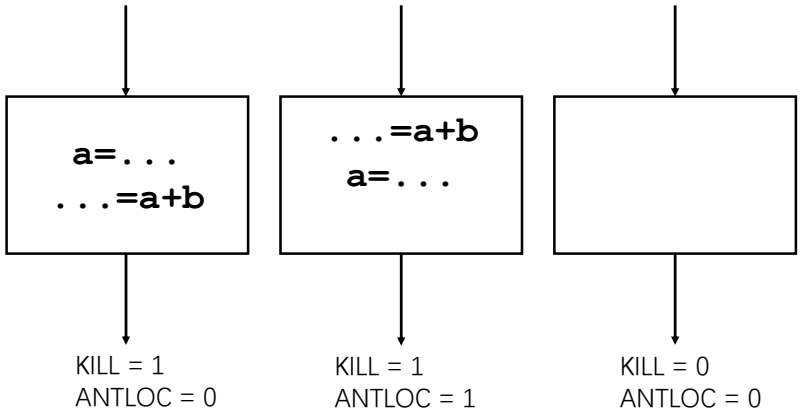


Figure 66: For `a+b`, the result of Anticipated Expression's transfer function within a basic block.

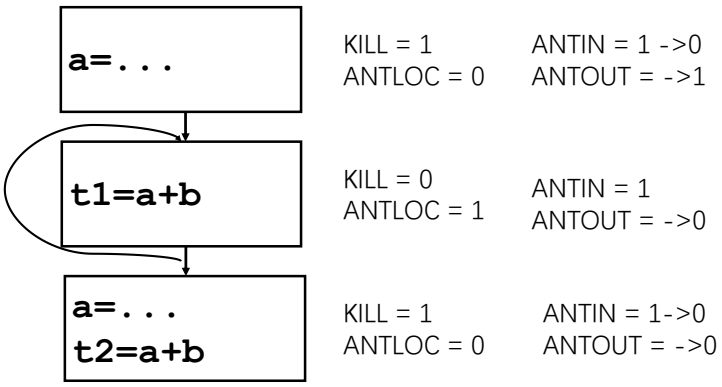


Figure 67: For `a+b`, the result Anticipated Expression of dataflow analysis.

13.3 Where Do we Want to Insert Computations?

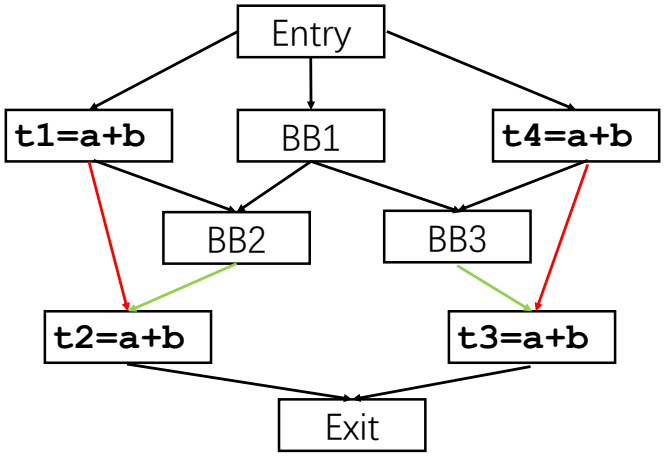


Figure 68: For `a+b`, `t2 = a + b` and `t3 = a + 4` are both partially redundant. But where can we insert the new computation `a + b` in order to optimally eliminate redundancy? The best choice is BB1 because this wil make `a + b` fully redundant. But if we insert to BB2 and BB3, there are some path that calculate `a + b` more than once (e.g. Entry→`t1= a+b` → BB2→`t2= a+b`)

We define "**Placement Possible** "(PP) dataflow analysis. First of all, we are only going to place computations at the end of the blocks. We want to insert the computation at the earliest place where our **Placement Possible** is true. If the **Placement Possible** is true output of a block(PPOUT), it means it is fine to insert at the end of this block or earlier. If it is true athe the beginning of the block, it means we could insert it at the beginning of the block or earlier. Becuase we want to insert it at the end of the block, so if **Placement Possible** is true at the beginning of the block(PPIN), you won't insert it in the block, it means you need to take care of something before. So when PPIN is true, it really means is for every predecessor block, it is either possible to keep moving it back and make it fully redundant by placing in that block or earlier or it is not necessary because it is already generatedin one of those blocks.

We insert if PPOUT is true and either PPIN is false so we cannot move it back any further or if we locally kill it then clearly we have to insert it here because trying to insert it earlier would not work. And we only want to insert it if it is not already available.

We want to delete an expression where PPIN is true (somehow we make it fully redundant) and it is anticipated locally.

For safety reasons, if we want to place at output of a block, we want to place at entry of all successors. (PPOUT)

$$\text{PPOUT[i]} = \begin{cases} 0 & i = \text{ entry} \\ \bigcap_{s \in \text{succ}(i)} \text{PPIN[s]} & \text{otherwise} \end{cases}$$

When PPIN is true, if means

- we have a local computation to place, or a placement at the end of this block which we can move up
- we want to move computation to output of all predecessors where expression is not already available (don't insert at input) (for every predecessor,)
- we gain something by moving it up (PAVIN heuristic) (not too far)

$$\text{PPIN[i]} = \begin{cases} 0 & i = \text{ exit} \\ ([\text{ ANTLOC[i]} \cup (\text{PPOUT } [i] - \text{KILL}[i])] & \\ \cap \bigcap_{p \in \text{ preds } (i)} (\text{ PPOUT } [p] \cup \text{ AVOUT } [p]) & \text{otherwise} \\ \cap \text{ PAVIN } [i]) & \end{cases}$$

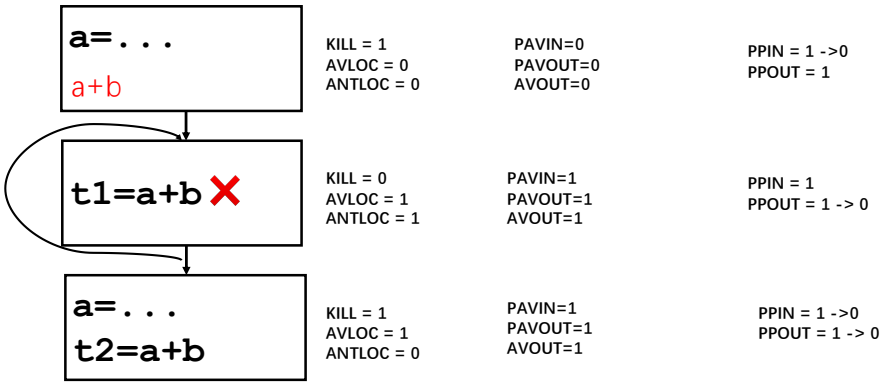


Figure 69: Example for PRE for `a+b`

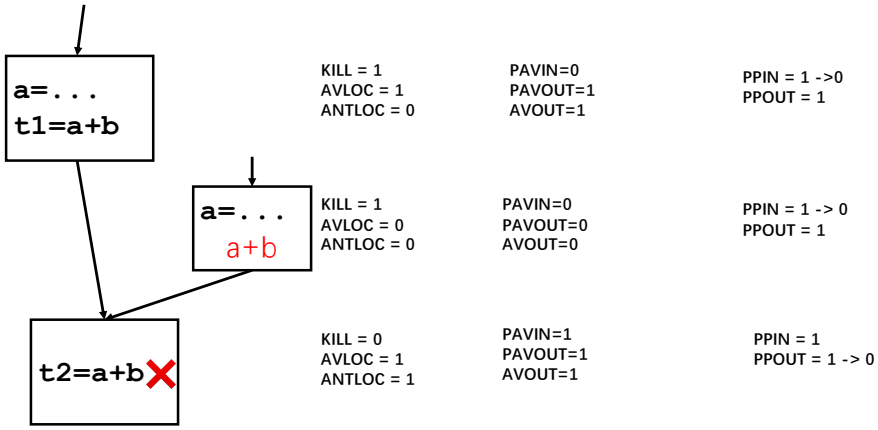


Figure 70: Example for PRE for `a+b`

13.3.1 Safety

It is safe to insert only if anticipated.

$$\text{PPIN[i]} \subseteq (\text{PPOUT[i]} - \text{KILL[i]}) \cup \text{ANTLOC[i]}$$

$$\text{PPOUT[i]} = \begin{cases} 0 & i = \text{entry} \\ \bigcap_{s \in \text{succ}(i)} \text{PPIN[s]} & \text{otherwise} \end{cases}$$

$$\text{INSERT} \subseteq \text{PPOUT} \subseteq \text{ANTOUT}$$

So it is safe.

13.4 Perform

On every path from an INSERT, there is a DELETE.

13.5 Limiations

13.6 A new way to think about partial redundancy

14 Lazy Code Motion

In 1979, Morel and Renvoise came up with an exciting new technique, the suppression of partial redundancies [9]. Their technique uniformly subsumed loop invariant code motion, common subexpression elimination, and the elimination of redundant computations. Probably, the most appealing facet of their technique was its structural purity: the transformation was solely based on data-flow analysis and did not require any specific knowledge on the control flow of the programs under investigation. On the other hand, their genuine proposal was conceptually complex. It involved an equation system of four highly interacting global properties while still suffering from three deficiencies. First, too few partial redundancies are removed. Intuitively, this was due to Morel's and Renvoise's design decision to insert code at the nodes of the underlying flow graph rather than at its edges, which unnecessarily restricted the possible computation points. Second, code was moved too far, which led to unnecessary register pressure. And third, the node placement was based on bidirectional data-flow equations which — from a conceptual point of view — are difficult to comprehend and — from a computational point of view — are more costly to compute.

As we started our work on lazy code motion, we were convinced that research efforts based on modifying the Morel/- Renvoise-style equations had led into a dead end. Partial redundancy elimination (PRE) is a beautiful technique with a simple underlying basic idea: expressions are hoisted to earlier program points increasing thereby their potential to make the original ones fully redundant, which can then be eliminated. We had the strong belief that it must be possible to construct a PRE-technique which is solely composed out of simple and well-understood components.

Decomposing the problem. A key idea to attack the problem was its **decomposition** based on a clean separation of concerns. We noticed that there are two optimization goals with a natural hierachy: the primary is to reduce the number of computations to a minimum (computational optimality); the secondary to avoid unnecessary code movement to **minimize the lifetimes of temporaries and hence the register pressure (lifetime optimality)**. There is no a topological order for the bidirectional dataflow analysis, so this does complicate things.

Solving the problem. Fortunately, we already had an offthe-shelf solution for the first optimization goal. Investigating the relationship between model checking and data-flow analysis has led to a modal logic specification of a computationally optimal PRE following an as-early-as-possible code placement strategy [10]. We called the resulting transformation "Busy Code Motion (BCM)," as it hoists code as far as possible. Technically it required only two simple unidirectional data-flow analyses. This simplicity revealed the solution to our secondary goal, the avoidance of unnecessary code motion: the code only had to "sink back" from the BCM insertion points as far as computational optimality was preserved, which can be realized simply by adding another unidirectional data-flow analysis. The resulting transformation, which **solves the problem of unnecessary register pressure**, hoists code just far enough to ensure computationally optimal results, the reason for it being called "Lazy Code Motion (LCM)."

This successful way of playing with simple analysis components was later extended to also control/minimize code size [11]. Here, the natural trade-off between the optimization goals led to different solutions depending on the chosen priority between size and speed.

14.1 Big Picture

First calculates the "earliest" set of blocks for insertion, this maximizes redundancy elimination but may also result in long register lifetimes. Then it calculates the "latest" set of blocks for

insertion, this achieves the same amount of redundancy elimination as "earliest" but hopefully reduces register lifetimes.[12, 13]

14.2 PRE vs. LCM

The goal of PRE is that by moving around the places where an expression is evaluated and keeping the result in a temporary variable when necessary, we often can reduce the number of evaluations of this expression along many of the execution paths, while not increasing that number along any path. However, it is not possible to eliminate all redundant computations along every path, unless we are allowed to change the control flow graph by creating new blocks and duplicating blocks.

New blocks creation

It can be used to break "critical edge", which is an edge leading from a node with more than one successor to a node with more than one predecessor.

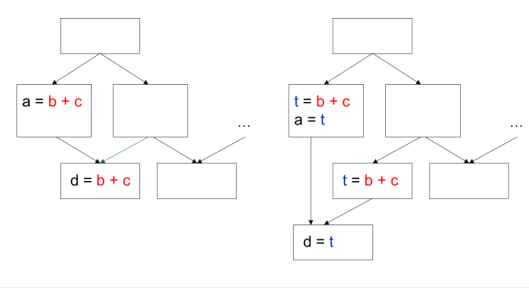


Figure 71: Example of new block creation

Block duplication

It can be used to isolate the path where redundancy is found.

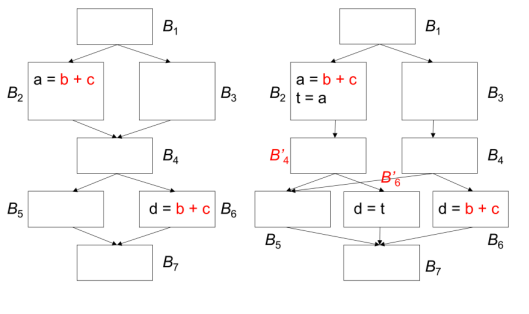


Figure 72: Example of block duplication

14.3 Preprocessing: Preparing the Flow Graph

First, we need modify the flow graph. Inorder to ensure redundancy elimination power, we add a basic block for every edge that leads to a basic block with multiple predecessors. Also, in LCM, we restrict placement of instructions to the beginning of a basic block Consider each statement as its own basic block to keep algorithm simple.

Full Redundancy: A Cut Set in a Graph

Full redundancy at p: expression a+b redundant on all paths

- a cut set: nodes that separate entry from p (could have multiple cut sets).
- each node in a cut set contains a calculation of a+b.
- a, b not redefined.

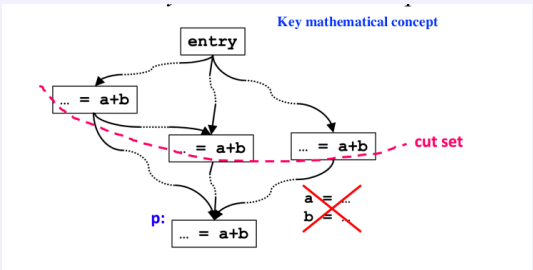


Figure 73: Full Redundancy

Partial Redundancy: Completing a Cut Set

Partial redundancy at p: redundant on some but not all paths

- Add operations to create a cut set containing a+b
- Note: Moving operations up can eliminate redundancy

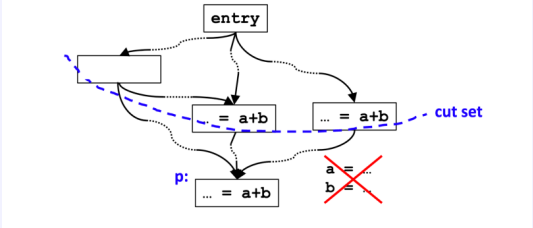


Figure 74: Partial Redundancy

14.4 Pass 1: Anticipated Expression

Direction	Backword
Domain	Set of expressions
Meet operator	$\cap$
Boundary	$IN[EXIT] = \phi$
Initialization for internal nodes	$IN[B] = \{allexpressions\}$
Finited escending chain?	✓
Transferfunction	$f_b(x) = EUse_b \cup (x - EKill_b)$
Monotone&Distributive?	✓
$EUse_b$	set of expressions computed in B (EUse, UEEXP).
$EKill_b$	set of expressions any of whose operands are defined in B

14.5 Where to insert/move instructions?

14.5.1 Choice 1 : frontier of anticipation

What is the result if we insert t = a + b at the frontier of anticipation ? i.e., those BBs for which a + b is anticipated to the entry of BB, but not anticipated to the entry of its parents.?

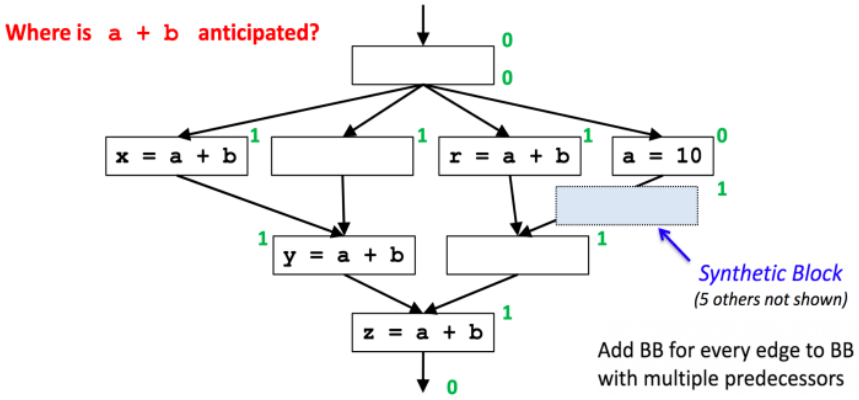


Figure 75: Frontier may be good for this example.

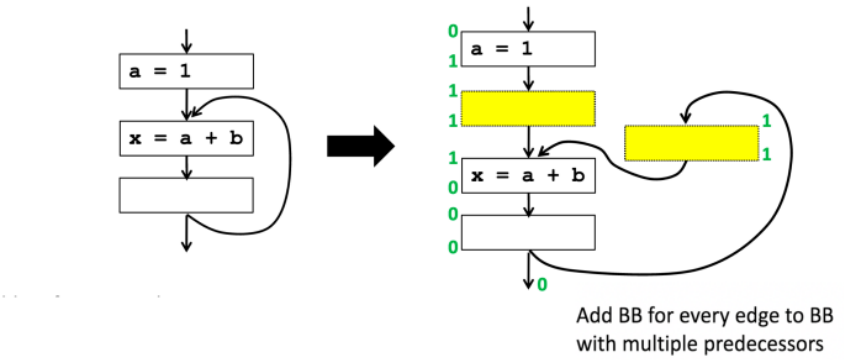


Figure 76: Frontier are clolored in yellow. Frontier of anticipation may be not a good choice. For this example, if we insert t = a + b at the frontier of anticipation, this doesn't eliminate redundancy within loop!

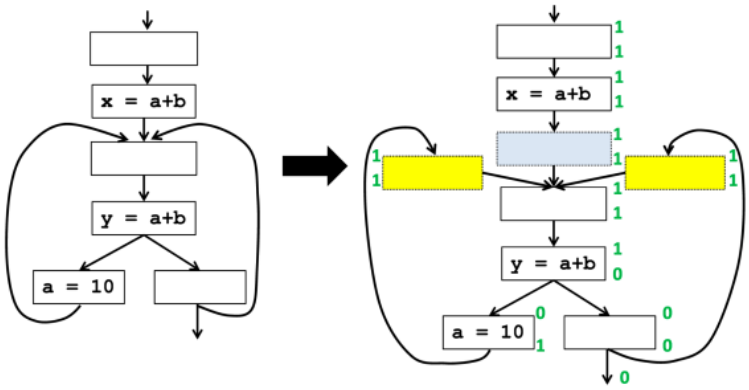


Figure 77: This example can also illustrate frontier is not our choice. In fact, we ideally like to insert "a+b" in this case to the left BB.

How to find such anticipated frontier and exclude "those not needed blocks" discussed in previous loop examples? Our final solution: Place expression at "anticipated" but not "will be available" blocks

14.6 Pass 2: Place As Early As Possible

Name	Available Expressions
Direction	Forward
Transferfunction	$f_b(x) = (Anticipated[b].in \cup x) - EKill_b$
Domain	Set of expressions
Meet operator	$\cap$
Boundary	$OUT[Entry] = \phi$
Initialization for internal nodes	$OUT[B] = \{allexpressions\}$

$$\text{earliest[b]} = \text{anticipated[b]} - \text{available[b]}$$

14.7 Pass 3: Lazy Code Motion

The values of expressions found to be redundant are usually held in registers until they are used. Computing a value as late as possible minimizes its lifetime: the duration between the time the value is defined and the time it is last used. Minimizing the lifetime of a value in turn minimizes the usage of a register.

postponable

An expression e is postponable at a program point p if

- all paths leading to p have seen earliest placement of e
- but not a subsequent use

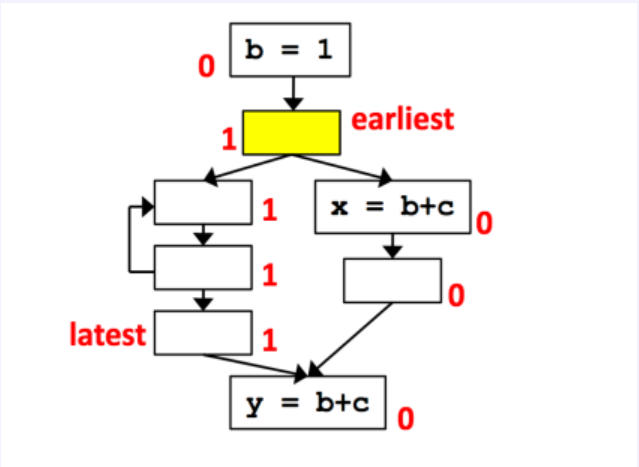


Figure 78: An example to illustrate Postponable Expressions.

Name	Postponable Expressions
Direction	Forward
Transferfunction	$f_b(x) = (earliest[b] \cap x) - EUse_b$
Domain	Set of expressions
Meet operator	$\cap$
Boundary	$OUT[Entry] = \phi$
Initialization for internal nodes	$OUT[B] = \{allexpressions\}$

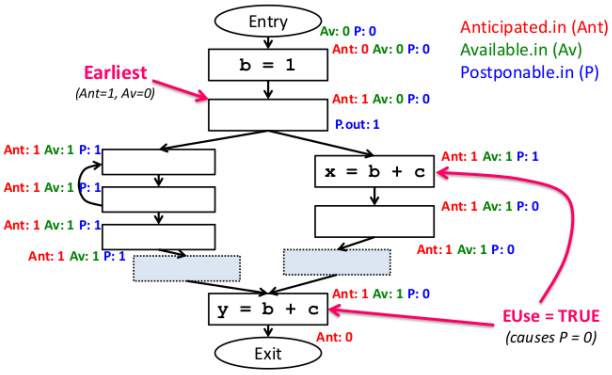


Figure 79: An example to illustrate Postponable.

latest[b] = (earliest[b] $\cup$ postponable.in[b]) $\cap$

($EUse_b \cup \neg(\bigcap_{s \in succ[b]}$(earliest[s] $\cup$ postponable.in[s])))

- OK to place expression: earliest or postponable
- Need to place at b if either
  - used in b, or
  - not OK to place in one of its successors

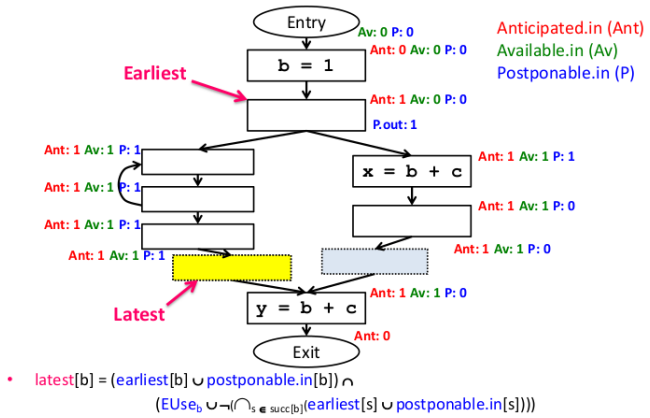


Figure 80: An example to illustrate "Latest".

14.8 Pass 4: Cleaning Up

Name	Used Expressions
Direction	Backward
Transferfunction	$f_b(x) = (EUse[b] \cap x) - latest[b]$
Domain	Set of expressions
Meet operator	$\cup$
Boundary	$in[exit] = \phi$
Initialization for internal nodes	$in[b] = \phi$

For all basic blocks b, if (x+y) $\in$ (latest[b] $\cap$ used.out[b]), at beginning of b: add new t = x+y, then replace every original x+y by t.

15 A Variation of Knoop, Ruthing, and Steffen's Lazy Code Motion

15.1 Where to Insert?

We want to insert the new computation where it is not partially available there.

Anticipable(Very Busy) Expression

An expression e is anticipable at a program point p if e will be computed along every path from p to p_{end}, and no variable in e is redefined until its computation. It is safe to move an expression to a basic block where that expression is anticipable. By "safe" we mean "performance safe", i.e., no extra computation will be performed. Notice that if an expression e is computed at a basic block where it is both available and anticipable, then that computation is clearly redundant.

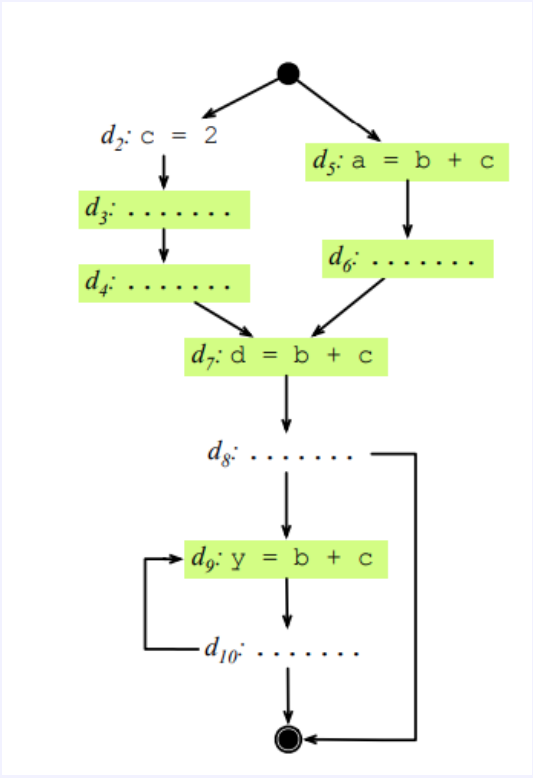


Figure 81: For `b+c`, the green blocks are anticipable points.

The key to partial redundancy elimination is deciding where to add computations of an expression to change partial redundancies into full redundancies (which may then be optimized away). There are now two steps that we must perform:

- First, we find the earliest places in which we can move the computation of an expression without adding unnecessary computations to the CFG. This step is like pushing the computation of the expressions up.
- Second, we try to move these computations down, closer to the places where they are necessary, without adding redundancies to the CFG. This phase is like pulling these computations down the CFG. So that we can, for instance, reduce register pressure.

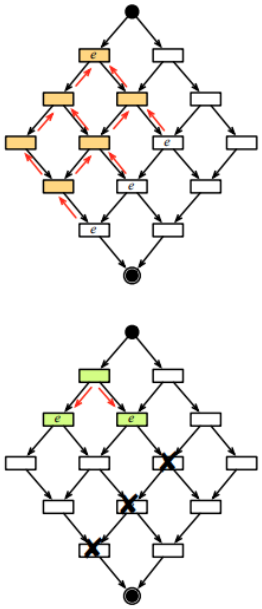


Figure 82: Pushing up, Pulling down.

15.1.1 Earliest Placemen

We must now find the earliest possible places where we can compute the target expressions. Earliest in the sense that p1 comes before p2 if p1 precedes p2 in any topological ordering of the CFG.

$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

For the Fisrt part, We can move an expression e to an edge ij only if e is anticipabled at the entrance of j. If the expression is available at the beginning of the edge, then we should not move it there. But the Second part, If an expression is anticipable at i, then we should not move it to ij, because we can move it to before i. On the other hand, if i kills the expression, then it cannot be computed before i.

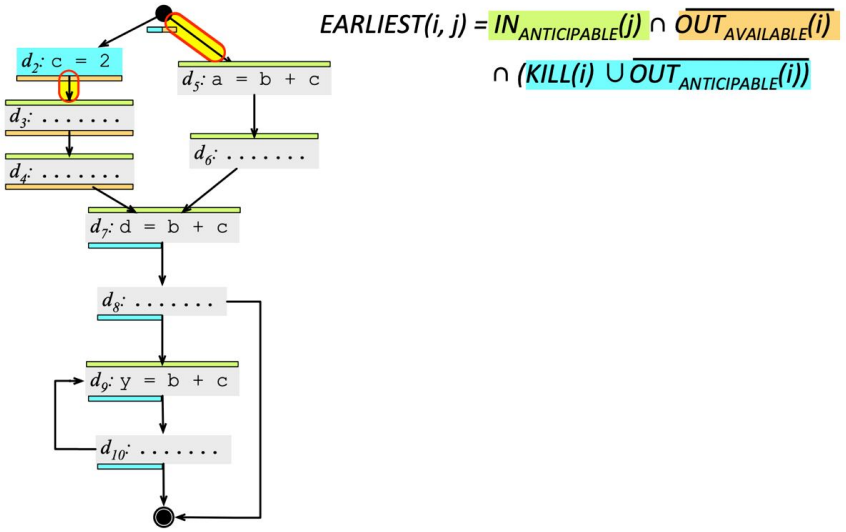


Figure 83: An example for calculating EARLIEST.

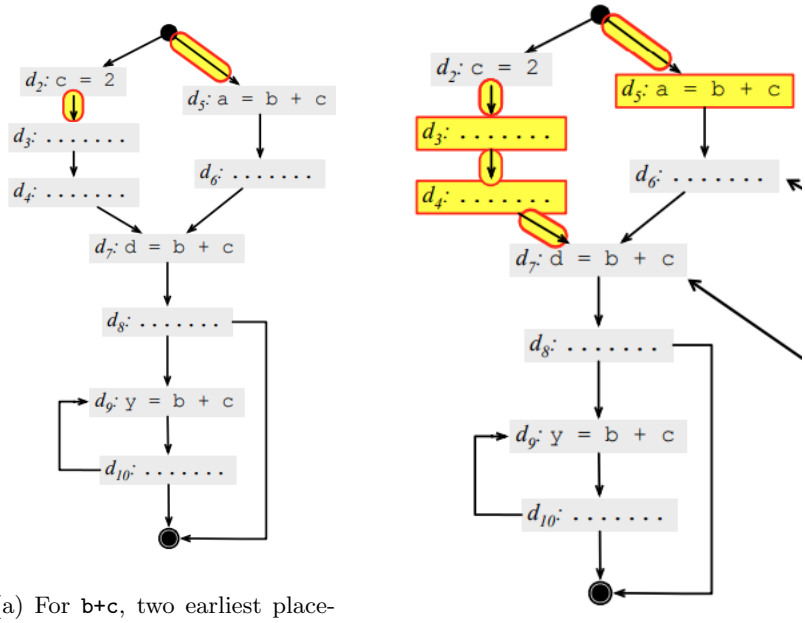
15.1.2 Latest Placement

$$\text{IN}_{\text{LATER}}(j) = \cap_{i \in \text{pred}(j)} \text{LATER}(i,j)$$
$$\text{LATER}(i,j) = \text{EARLIEST}(i,j) \cup \left(\text{IN}_{\text{LATER}}(i) \cap \overline{\text{EXPR}(i)}\right).$$

LATER(i,j) is true if we can move the computation of the expression down the edge ij. An expression e is in EXPR(i) if e is computed at i. This predicate is also computed for edges, although we have IN_{LATER} being computed for nodes.

For LATER(i,j): If EARLIEST(i, j) is true, then LATER(i,j) is also true, as we can move the computation of e to edge ij without causing redundant computations. If $IN_{\text{LATER}}(i,j)$ is true, and the expression is not used at i, then LATER(i,j) is true. If the expression is used at i, then there is no point in computing it at ij, because it will be recomputed at i anyway.

For $IN_{\text{LATER}}(i,j)$, it is a condition that we propagate down. If all the predecessors of a node j accept the expression as nonredundant, then we can compute the expression down on j.



(a) For `b+c`, two earliest placement points is colored in red.

(b) For `b+c`, Latest placement edhes and blocks.

Figure 84: A more complex example of strength reduction.

15.1.3 Where to Insert Computations?

We insert the new computations at the latest possible place.That is

$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

There are different insertion points, depending on the structure of the CFG, if x ∈ INSERT(i, j):

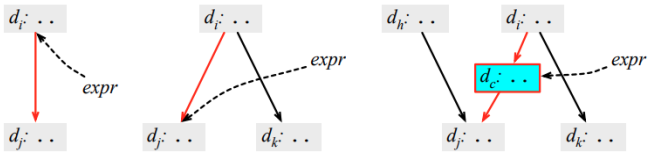


Figure 85: Different inser9on points

15.2 Modify CFG

Rename all compuation of the expression.

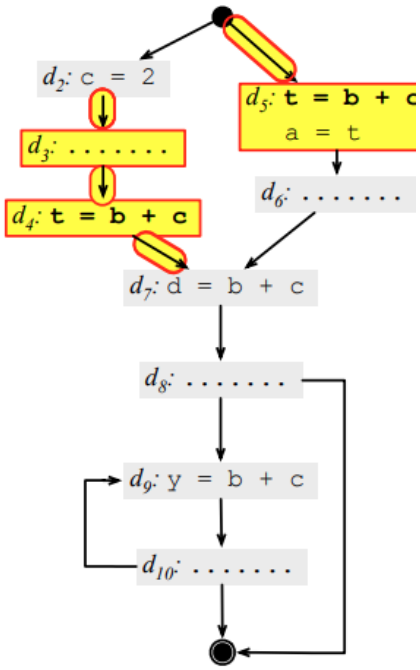


Figure 86: For `b+c`, the result of applyiny modifying CFG.

15.3 Which Computations to Remove?

We remove computations that are already covered by the latest points, and that we cannot use later on.

$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

For First part, of course, the expression must be used in the block, otherwise we would have nothing to delete. For second part, The expression may not be a computation that is necessary later on.

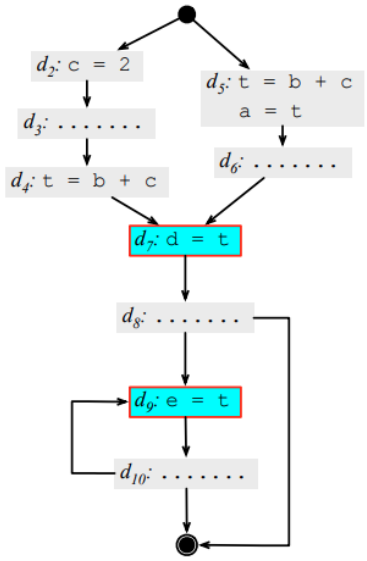
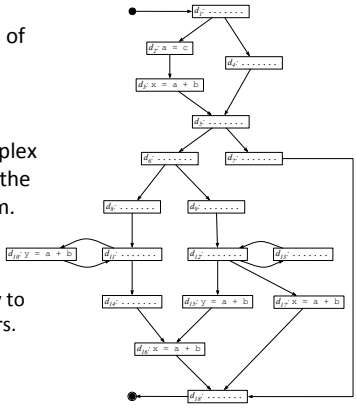


Figure 87: For `b+c`, the result of applyiny deleting redundancy `b+c`

15.4 A fully explained example

An Example to Conquer them All

- The original formulation of Lazy Code Motion was published in a paper by Knoop *et al*[◇].
- The authors used a complex example to illustrate all the phases of their algorithm.
- Many papers are build around examples.
  - That is a good strategy to convey ideas to readers.



◇: Lazy Code Motion, PLDI (1992)

Available Expressions

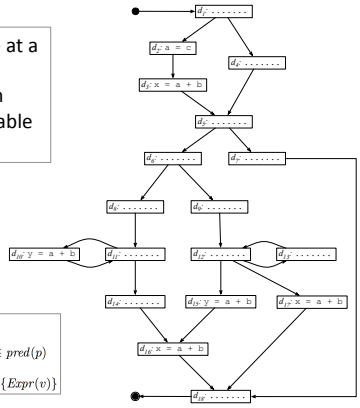
An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

What is the OUT set of available expressions in the example?

$p : v = E$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$



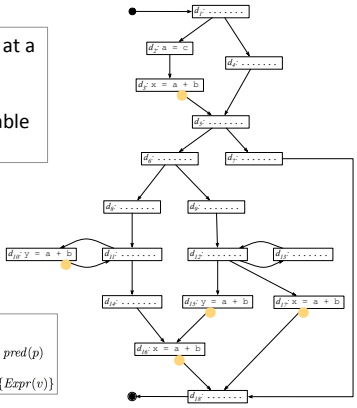
Available Expressions

An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

$p : v = E$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$



Anticipable Expressions

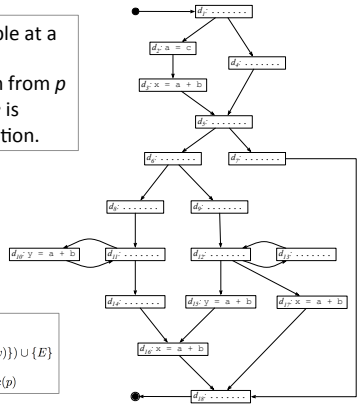
An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.

What is the IN set of anticipable expressions in the example?

$p : v = E$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$



Anticipable Expressions

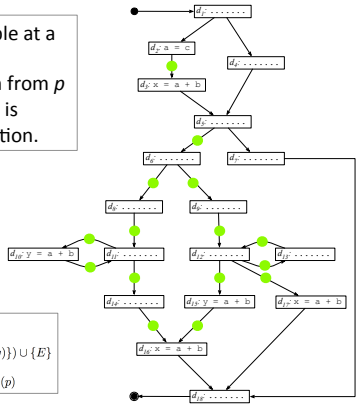
An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.

What is the IN set of anticipable expressions in the example?

$p : v = E$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

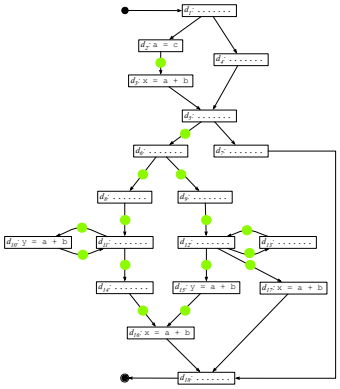
$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$



EARLIEST(i, j) = $IN_{ANTICIPABLE}$(j) ∩ ~~$OUT_{AVAILABLE}$(i)~~ ∩ (KILL(i) ∪ ~~$OUT_{ANTICIPABLE}$(i)~~)

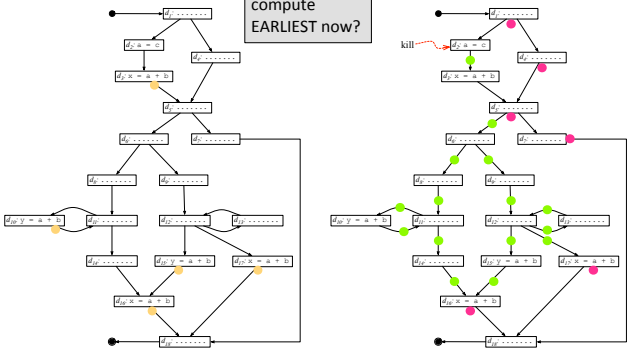
What is KILL(i) ∪ $\overline{OUT_{ANTICIPABLE}(i)}$ in our running example?

This figure shows the IN sets of anticipability analysis.



EARLIEST(i, j) = $IN_{ANTICIPABLE}$(j) ∩ ~~$OUT_{AVAILABLE}$(i)~~ ∩ (KILL(i) ∪ ~~$OUT_{ANTICIPABLE}$(i)~~)

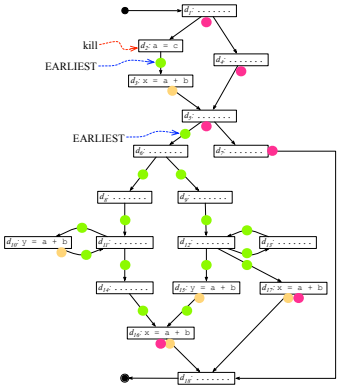
Can you compute EARLIEST now?



EARLIEST(i, j) = $IN_{ANTICIPABLE}$(j) ∩ ~~$OUT_{AVAILABLE}$(i)~~ ∩ (KILL(i) ∪ ~~$OUT_{ANTICIPABLE}$(i)~~)

We have two EARLIEST edges in this CFG.

- Anticipable at IN(j)
- Not anticipable at OUT(i)
- Available at OUT(i)

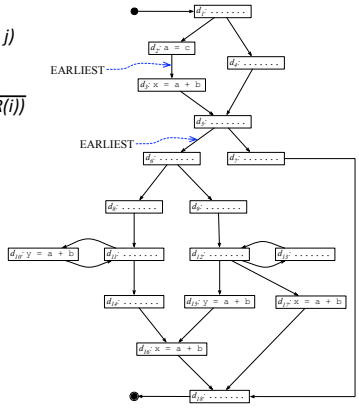


Latest

$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$

$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$

The goal now is to compute the latest IN sets, and the latest edges. Can you do it?



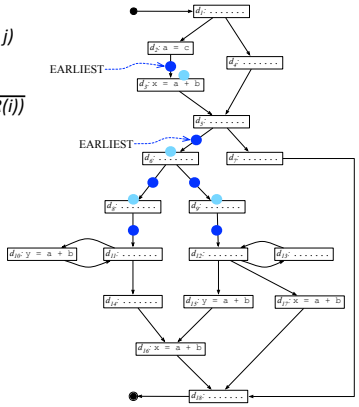
Latest

$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$

$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$

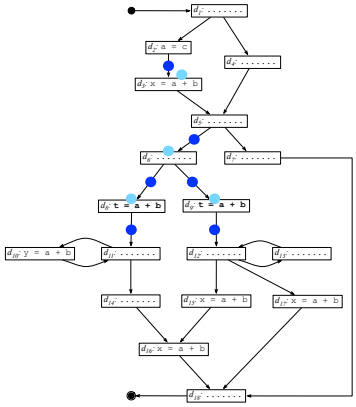
The LATEST edges are marked with the dark blue circles.

The LATEST IN sets are marked with the light blue circles.



$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$

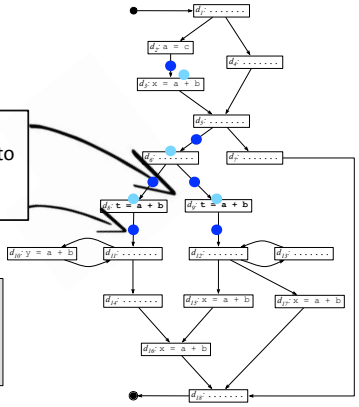
We must now find the sites where we can insert new computations of a + b. Can you compute INSERT(i, j)?



$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$

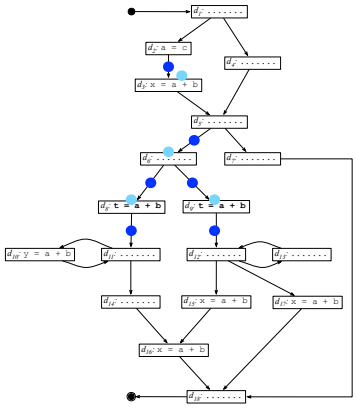
In this example, we only have two sites to insert new computations.

Do you remember why we insert the computation of a + b at d_8, instead of d_{11}?



$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

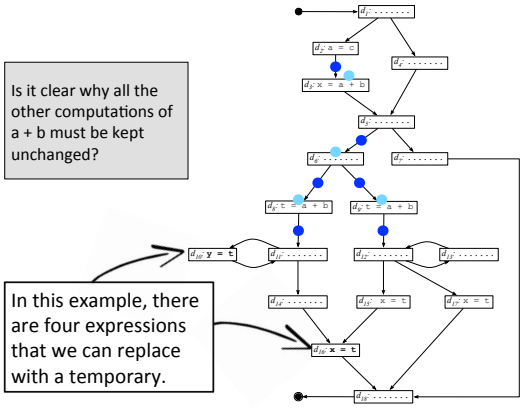
We must now delete redundant computations that exist at program points p. Can you determine DELETE(p)?



$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

Is it clear why all the other computations of a + b must be kept unchanged?

In this example, there are four expressions that we can replace with a temporary.



16 Region-Based Analysis

The iterative data-flow analysis algorithm we have discussed so far is just one approach to solving data-flow problems. Here we discuss another approach called region-based analysis[14]. Recall that in the iterative-analysis approach, we create transfer functions for basic blocks, then find the fixedpoint solution by repeated passes over the blocks. Instead of creating transfer functions just for individual blocks, a region-based analysis finds transfer functions that summarize the execution of progressively larger regions of the program. Ultimately, transfer functions for entire procedures are constructed and then applied, to get the desired data-flow values directly.

While a data-flow framework using an iterative algorithm is specified by a semilattice of data-flow values and a family of transfer functions closed un-der composition, region-based analysis requires more elements. A region-based framework includes both a semilattice of data-flow values and a semilattice of transfer functions that must possess a meet operator, a composition operator, and a closure operator.

A region-based analysis is particularly useful for data-flow problems where paths that have cycles may change the data-flow values. The closure operator allows the effect of a loop to be summarized more effectively than does iterative analysis. The technique is also useful for interprocedural analysis, where transfer functions associated with a procedure call may be treated like the transfer functions associated with basic blocks.

16.1 Motivating Example

Consider the example in 88, we want to know how many bits needed to store the return value for a fpga device. We can pessimistically solve for the worst case. But we can also try to be more precise to calculate for each call site. It would be nice if instead of having to go back and iteratively solve a problem, for each different value of x.

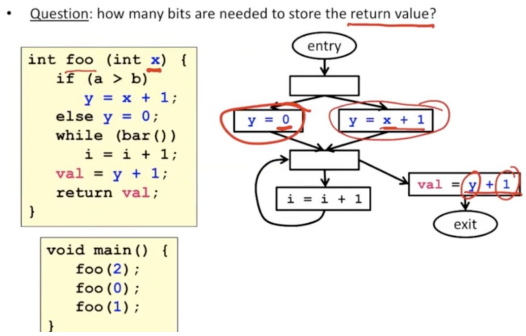


Figure 88: Motivating Example for region-based analysis

The idea behind region-based analysis is that we want to create a transfer function for the entire procedure.

16.2 Algorithm

Region

A region in a flow graph is a set of nodes with a header that dominates all other nodes in a region.

In Iterative Analysis, Transfer function F_B summarize effect from beginning to end of basic block B.

In Region-Based Analysis, Transfer function $F_{R,B}$ summarize effect from beginning of region R to end of basic block B. Recursively construct a larger region R from smaller regions construct $F_{R,B}$ from transfer functions for smaller regions until the program is one region. Let P be the region for the entire program, and v be initial value at entry node, out[B] = $F_{R,B}$ (v), in[B] = $\cap_{B'} out[B']$ where B' is a predecessor of B

We will use Reaching definitions as our transfer function to illustrate Region-Based Analysis.

16.2.1 Operations on Transfer Functions

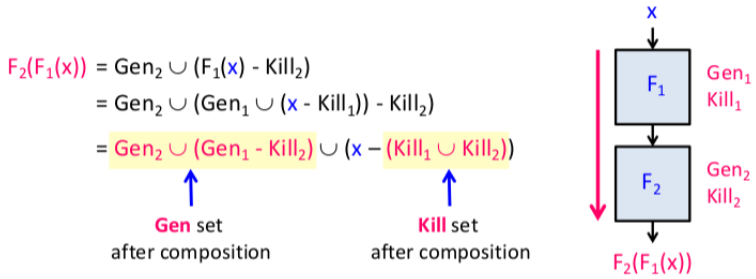


Figure 89: Operations on Transfer Functions: Composition

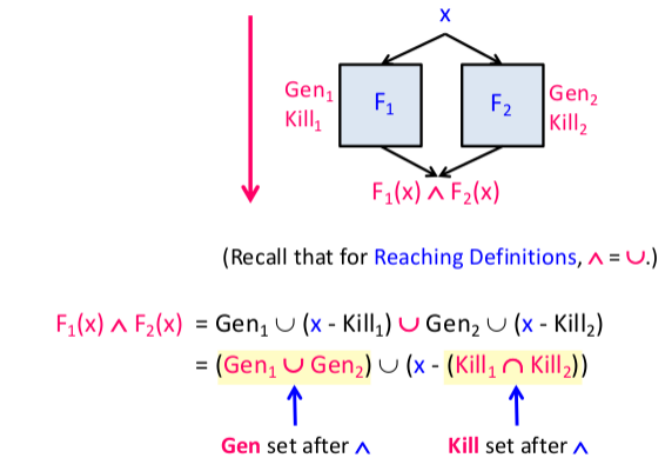


Figure 90: Operations on Transfer Functions: Meet

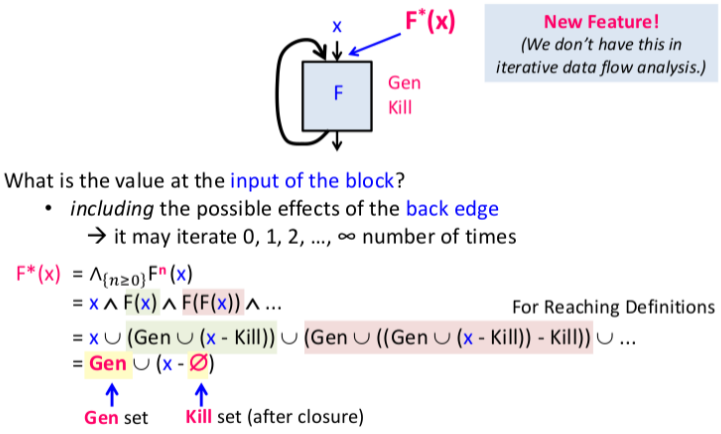


Figure 91: Operations on Transfer Functions: Closure

16.2.2 Structure of Nested Regions (An Example)

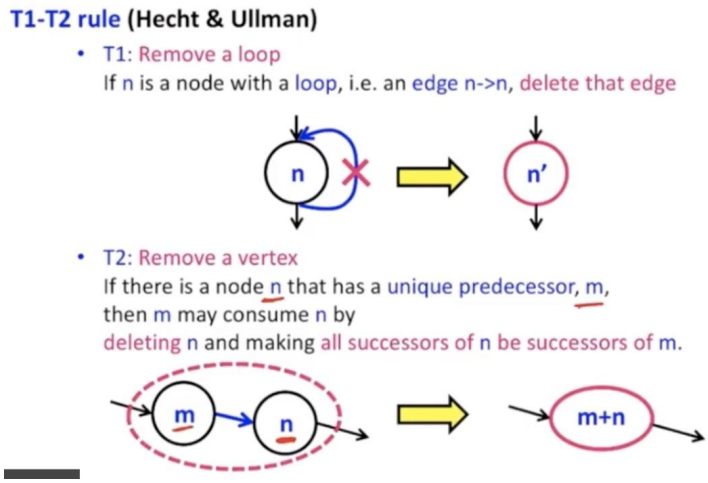
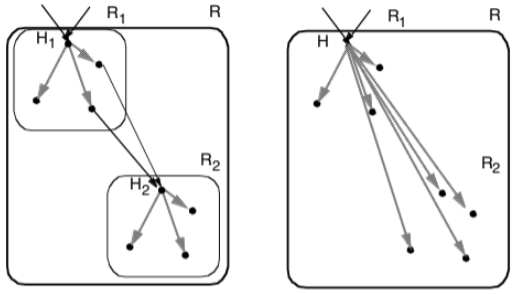


Figure 92

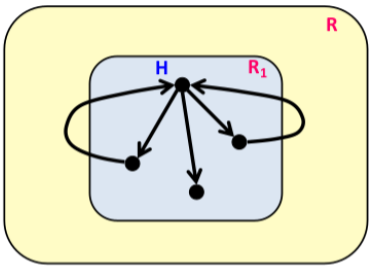
16.2.3 Transfer Functions for T2 Rule



- **Transfer function**
  $F_{R,B}$: summarizes the effect from beginning of R to end of B
  $F_{R,in(H2)}$: summarizes the effect from beginning of R to beginning of H2
  - Unchanged for blocks B in region R_1 ($F_{R,B} = F_{R1,B}$)
  - $F_{R,in(H2)} = \wedge_P F_{R,P}$, where p is a predecessor of H_2
  - For blocks B in region R_2: $F_{R,B} = F_{R2,B} \cdot F_{R,in(H2)}$

Figure 93

16.2.4 Transfer Functions for T1 Rule



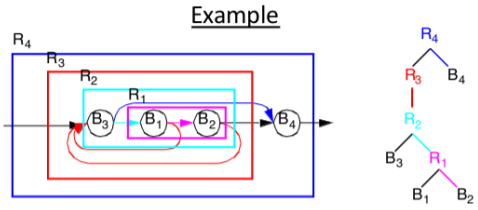
R: new region
(subsumes back edges from $R_1 \rightarrow R_1$)

Observations:

- the header of R_1 (i.e. **H**) is also the header of **R**
- we already know how to get from **H** to **B** for every block B in R_1: i.e. $F_{R1,B}$
  - this will be the *last step* in getting from the new **R** to **B** (**composition**)
- what's new: we need to get from **R** to the input of **H**, *including back edges!*
  - this involves both **meet** ($\wedge$) and **closure** (*) operations

Figure 94

16.2.5 Example: Reaching Definitions



R	Rule	R'	$F_{R,in(R')}$	$F_{R,B1}$	$F_{R,B2}$	$F_{R,B3}$	$F_{R,B4}$
R_1	T_2	B_2	F_{B1}	F_{B1}	$F_{B2} \circ F_{R1,in(B2)}$	--	--
R_2	T_2	R_1	F_{B3}	$F_{R1,B1} \circ F_{R2,in(R1)}$	$F_{R1,B2} \circ F_{R2,in(R1)}$	F_{B3}	--
R_3	T_1	R_2	$(F_{R2,B1} \wedge F_{R2,B2})^*$	$F_{R2,B1} \circ F_{R3,in(R2)}$	$F_{R2,B2} \circ F_{R3,in(R2)}$	$F_{R2,B3} \circ F_{R3,in(R2)}$	--
R_4	T_2	B_4	$F_{R3,B3} \wedge F_{R3,B2}$	$F_{R3,B1}$	$F_{R3,B2}$	$F_{R3,B3}$	$F_{B4} \circ F_{R4,in(B4)}$

- R: region name; R': region w/subsumed header; R'': region w/header that remains
- T_2: $F_{R,B} = F_{R'',B}$ for $B \in R''$; $F_{R,B} = F_{R',B} \circ F_{R,in(R')}$ for $B \in R'$; $F_{R,in(R')} = \wedge_P F_{R,P}$, $p \in pred(HR')$
- T_1: $F_{R,B} = F_{R',B} \circ F_{R,in(R')}$; $F_{R,in(R')} = (\wedge_P F_{R',P})^*$, $p \in pred(HR')$

Figure 95

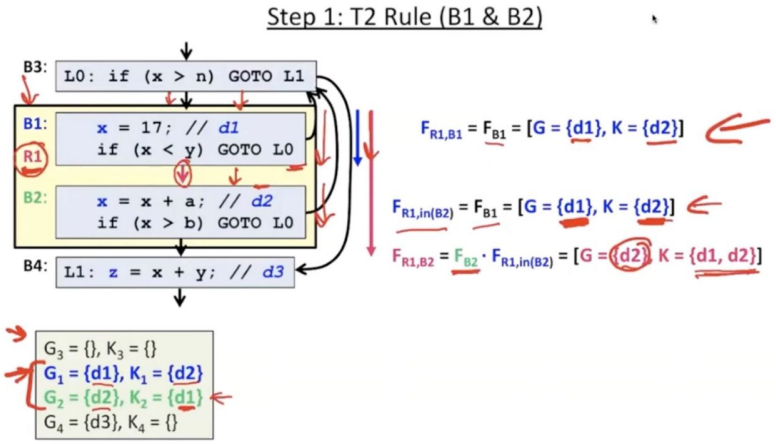


Figure 96

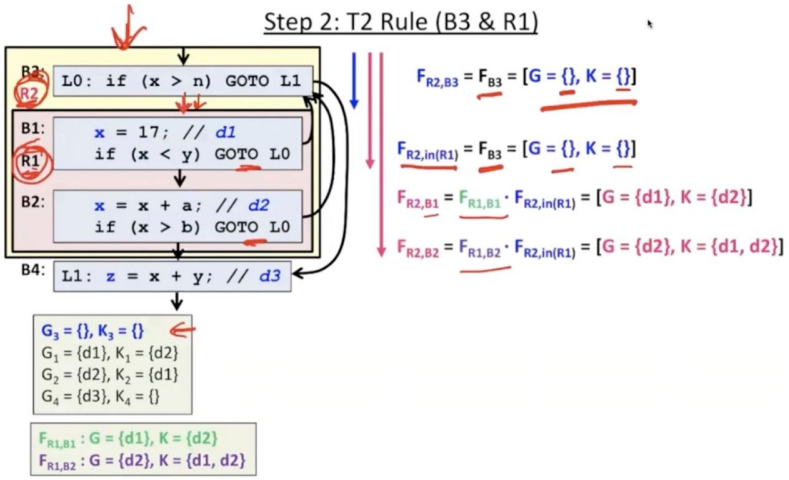


Figure 97

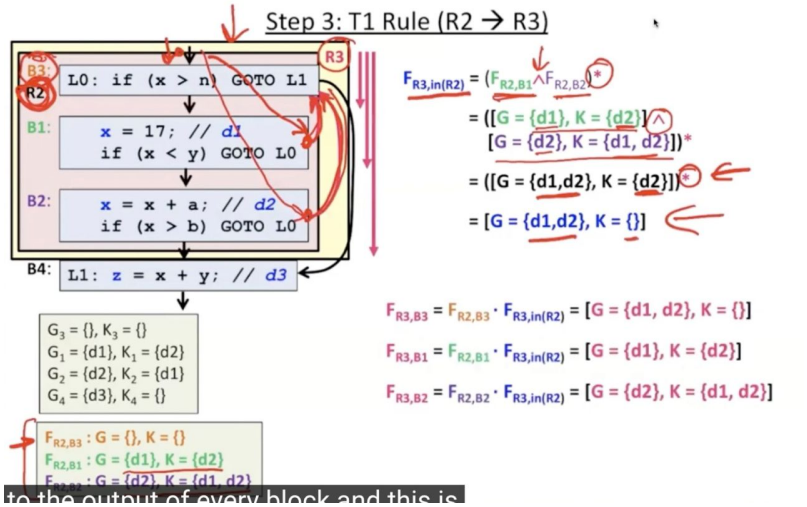


Figure 98

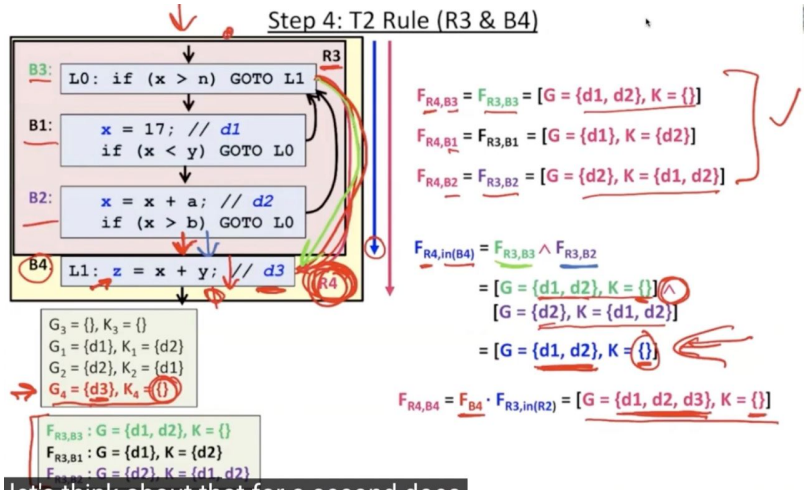


Figure 99

17 Pointer Analysis

Pointer analysis is a compile-time technique that helps identify relationships between pointer variables and the memory locations that they point to during program execution. Pointers are powerful programming constructs and allow complex memory manipulation during program execution through techniques such as pointer arithmetic and dynamic memory allocation. Pointer relationships can thus be complex and difficult to analyze at compile time. On the other hand, however, they provide the benefit of simplifying various other compile time analyses such as constant propagation, alias analysis in C programs that allow extensive use of pointers.[15]

Potential applications of pointer analysis for multithreaded programs include: the development of sophisticated software engineering tools such as race detectors, memory leak detectors, wild pointer detectors and program slicers; memory system optimizations such as prefetching and moving computation to remote data; automatic batching of long latency file system operations; support parallelism in different levels: instruction-level parallelism and thread-level parallelism ; and to provide information required to apply traditional compiler optimizations such as constant propagation, common subexpression elimination, register allocation, code motion and induction variable elimination to multithreaded programs.[16]

Aliases

Two variables are aliases if they reference the same memory location.

The Pointer Alias Analysis Problem

Decide for every pair of pointers at every program point: do they point to the same memory location?

17.1 Background

A pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location. A points-to analysis , or similarly, an analysis based on a compact representation , attempts to determine what storage locations a pointer can point to. This information can then be used to determine the aliases in the program. Alias information is central to determining what memory locations are modifed or referenced.

There are several dimensions that aect the cost/precision trade-offs of interprocedural pointer analyses. How a pointer analysis addresses each of these dimensions helps to categorized the analysis. An empirical comparison with a difference in more than one dimension can limit the usefulness of the comparison. Some of the dimensions are

Flow-sensitivity: Is control-ow information of a procedure used during the analysis? By not considering control ow information, and therefore computing a conservative summary, ow-insensitive analyses compute one solution for either the whole program or for each method, whereas a ow-sensitive analysis computes a solution for each program point. Flow-insensitive analyses thus can be more efficient, but less precise than a ow-sensitive analysis. Flow-insensitive analyses are either equality-based, which treat assignments as bidirectional and typically use a union-find data structure, or subset-based, which treat an assignment as a unidirectional ow of values.

Context-sensitivity: Is calling context considered when analyzing a function or can values ow from one call through the function and return to another caller?

Heap modeling: Are ob jects named by allocation site, or is a more sophisticated shape analysis performed?

Aggregate modeling: Are elements of aggregates distinguished or collapsed into one object?

Whole program: Does an analysis require the whole program or can a sound solution be obtained by analyzing only components of a program?

Alias representation: Is an explicit alias representation [51, 64] or a points-to/compact representation used?

17.2 Flow-Sensitivity

Flow-sensitive pointer analysis respects a program's control flow and computes a separate solution for each program point, in contrast to a flow-insensitive analysis, which ignores statement ordering and computes a single solution that is conservatively correct for all program points.

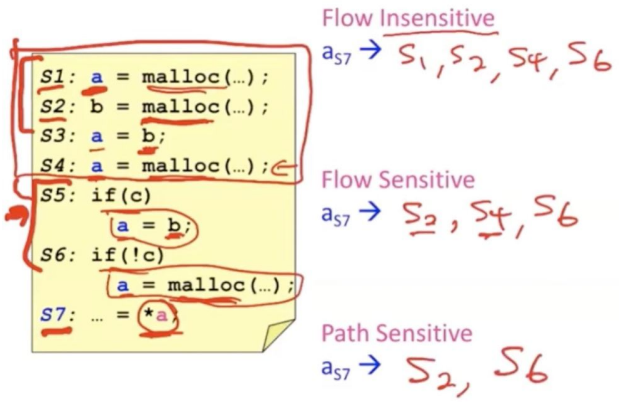


Figure 100: Flow-sensitive vs. flow-insensitive analysis

17.3 Context-sensitive

Context sensitivity has a most significant impact on analysis precision due to separate treatment for each method in the program. In practice, each time analysis considers new method call it creates a new structure that represents unique method scope in the memory. It treats all read/write operations inside this method in scope of this context and makes this information available later for its processing.

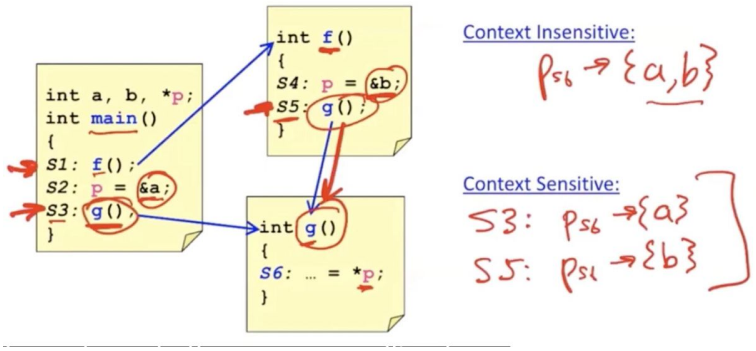


Figure 101: Context-sensitive vs. Context-insensitive analysis

17.4 Modeling Aggregates

A key implementation detail is whether aggregate com- ponents are distinguished or summarized into one object. C/C++'s weak typing makes this diÆcult to address cor- rectly. Thus, most published work does not distinguish ag- gregates. However, this diÆculty does not exist in a strongly- type language like Java, and therefore, components should be distinguished in such languages. Most recent work has chosen to distinguish components. Unfortunately, few researchers have studied the impact of this decision.

17.5 Andersen's Points-To Analysis

[17]Two common kinds of pointer analysis are alias analysis and points-to analysis. Alias analysis computes sets S holding pairs of variables pp, qq, where p and q may (or must) point to the same location. Points-to analysis, as described above, computes a relation points-topp, xq, where p may (or must) point to the location of the variable x. We will focus primarily on points-to analysis, beginning with a simple but useful approach originally proposed by Andersen (PhD thesis: "Program Analysis and Specialization for the C Programming Language"). Our initial setting will be C programs. We are interested in analyzing instructions that are relevant to pointers in the program. Ignoring for the moment memory allocation and arrays, we can decompose all pointer operations into four types: taking the address of a variable, copying a pointer from one variable to another, assigning through a pointer, and dereferencing a pointer:

$$
\begin{array}{rcl}
I & ::= & \ldots \\
 & | & p := \&x \\
 & | & p := q \\
 & | & *p := q \\
 & | & p := *q
\end{array}
$$

Andersen's points-to analysis is a context-insensitive interprocedural analysis. It is also a flowinsensitive analysis, that is an analysis that does not consider program statement order. Contextand

flow-insensitivity are used to improve the performance of the analysis, as precise pointer analysis can be notoriously expensive in practice.

We will formulate Andersen's analysis by generating set constraints which can later be processed by a set constraint solver using a number of technologies. Constraint generation for each statement works as given in the following set of rules. Because the analysis is flow-insensitive, we do not care what order the instructions in the program come in; we simply generate a set of constraints and solve them.

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p} \textit{ address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q} \textit{ copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q} \textit{ assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q} \textit{ dereference}$$

The constraints generated are all set constraints. The first rule states that a constant location lx, representation the address of x, is in the set of location pointed to by p. The second rule states that the set of locations pointed to by p must be a superset of those pointed to by q. The last two rules state the same, but take into account that one or the other pointer is dereferenced.

A number of specialized set constraint solvers exist and constraints in the form above can be translated into the input for these. The dereference operation (the * in *p $\subseteq$ q) is not standard in set constraints, but it can be encoded—see Fahndrich's Ph.D. thesis for an example of how ¨ to encode Andersen's points-to analysis for the BANE constraint solving engine. We will treat constraint-solving abstractly using the following constraint propagation rules:

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \textit{ copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \textit{ assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \textit{ dereference}$$

Figure 102

We can also apply Andersen's analysis to programs with dynamic memory allocation, such as:

$$
\begin{array}{ll}
1: & q := malloc_1() \\
2: & p := malloc_2() \\
3: & p := q \\
4: & r := \&p \\
5: & s := malloc_3() \\
6: & *r := s \\
7: & t := \&s \\
8: & u := *t
\end{array}
$$

Figure 103

In this example, the analysis is run the same way, but we treat the memory cell allocated at each malloc or new statement as an abstract location labeled by the location n of the allocation point. We can use the rules:

$$\frac{}{[\![p := malloc_n()]\!] \hookrightarrow l_n \in p} \; malloc$$

Figure 104

We must be careful because a malloc statement can be executed more than once, and each time it executes, a new memory cell is allocated. Unless we have some other means of proving that the malloc executes only once, we must assume that if some variable p only points to one abstract malloc'd location l_n, that is still may-alias information (i.e. p points to only one of the many actual cells allocated at the given program location) and not must-alias information.

Analyzing the efficiency of Andersen's algorithm, we can see that all constraints can be generated in a linear $O(n)$ pass over the program. The solution size is $O(n^2)$ because each of the $O(n)$ variables defined in the program could potentially point to $O(n)$ other variables.

We can derive the execution time from a theorem by David McAllester published in SAS'99. There are $O(n)$ flow constraints generated of the form p $\subseteq$ q, *p $\subseteq$ q, or p $\subseteq$ *q. How many times could a constraint propagation rule fire for each flow constraint? For a p $\subseteq$ q constraint, the rule may fire at most $O(n)$ times, because there are at most $O(n)$ premises of the proper form $l_x \in p$. However, a constraint of the form p $\subseteq$ *q could cause $O(n^2)$ rule firings, because there are $O(n)$ premises each of the form $l_x \in p$ and$l_r \in q$. With $O(n)$ constraints of the form p $\subseteq$ *q and $O(n^2)$firings for each, we have $O(n^3)$ constraint firings overall. A similar analysis applies for *p $\subseteq$ q constraints. McAllester's theorem states that the analysis with $O(n^3)$ rule firings can be implemented in $O(n^3)$ time. Thus we have derived that Andersen's algorithm is cubic in the size of the program, in the worst case.

17.5.1 Field-Sensitive Analysis

What happens when we have a pointer to a struct in C, or an object in an object-oriented language? In this case, we would like the pointer analysis to tell us what each field in the struct or object points to. A simple solution is to be field-insensitive, treating all fields in a struct as equivalent. Thus if p points to a struct with two fields f and g, and we assign:

$$1: \quad p.f := \&x$$
$$2: \quad p.g := \&y$$

Figure 105

A field-insensitive analysis would tell us (imprecisely) that p.f could point to y. In order to be more precise, we can track the contents each field of each abstract location separately. In the discussion below, we assume a setting in which we cannot take the address of a field; this assumption is true for Java but not for C. We can define a new kind of constraints for fields:

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f} \textit{field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q} \textit{field-assign}$$

Figure 106

Now assume that objects (e.g. in Java) are represented by abstract locations l. We can process field constraints with the following rules:

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_q.f}{l_f \in p} \textit{field-read}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_p.f} \textit{field-assign}$$

Figure 107

If we run this analysis on the code above, we find that it can distinguish that p.f points to x and p.g points to y.

17.6 Steensgaard's Points-To Analysis

For large programs, a cubic algorithm is too inefficient. Steensgaard proposed an pointer analysis algorithm that operates in near-linear time, supporting essentially unlimited scalability in practice. The first challenge in designing a near-linear time points-to analysis is to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing all of these pointers explicitly will inherently take $O(n^2)$space. The solution Steensgaard found is based on using constant space for each variable in the program. His analysis associates each variable p with an abstract location named after the variable. Then, it tracks a single points-to relation between that abstract location p and another one q, to which it may point. Now, it is possible that in some

real program p may point to both q and some other variable r. In this situation, Steensgaard's algorithm unifies the abstract locations for q and r, creating a single abstract location representing both of them. Now we can track the fact that p may point to either variable using a single points-to relationship.

For example, consider the program below:

$$
\begin{aligned}
1: &\quad p := \&x \\
2: &\quad r := \&p \\
3: &\quad q := \&y \\
4: &\quad s := \&q \\
5: &\quad r := s
\end{aligned}
$$

Figure 108

Andersen's points-to analysis would produce the following graph:

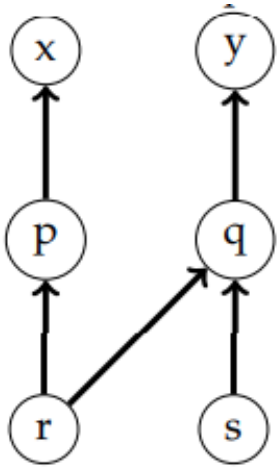


Figure 109

But in Steensgaard's setting, when we discover that r could point both to q and to p, we must merge q and p into a single node:

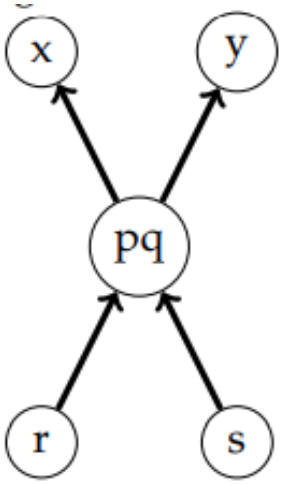


Figure 110

Notice that we have lost precision: by merging the nodes for p and q our graph now implies that s could point to p, which is not the case in the actual program. But we are not done. Now pq has two outgoing arrows, so we must merge nodes x and y. The final graph produced by Steensgaard's algorithm is therefore:

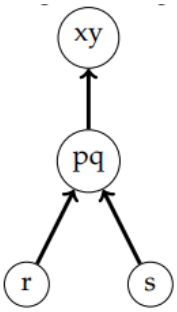


Figure 111

To define Steensgaard's analysis more precisely, we will study a simplified version of that ignores function pointers. It can be specified as follows:

$$\frac{}{[\![p := q]\!] \hookrightarrow join(*p, *q)}\ copy$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow join(*p, x)}\ address\text{-}of$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow join(*p, **q)}\ dereference$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow join(**p, *q)}\ assign$$

Figure 112

With each abstract location p, we associate the abstract location that p points to, denoted *p. Abstract locations are implemented as a union-find data structure so that we can merge two abstract locations efficiently. In the rules above, we implicitly invoke find on an abstract location before calling join on it, or before looking up the location it points to.

The join operation essentially implements a union operation on the abstract locations. However, since we are tracking what each abstract location points to, we must update this information also. The algorithm to do so is as follows:

```
join(e1, e2)
    if (e1 == e2)
        return
    e1next = *e1
    e2next = *e2
    unify(e1, e2)
    join(e1next, e2next)
```

Figure 113

Once again, we implicitly invoke find on an abstract location before comparing it for equality, looking up the abstract location it points to, or calling join recursively.

As an optimization, Steensgaard does not perform the join if the right hand side is not a pointer. For example, if we have an assignment p := q and q has not been assigned any pointer value so far in the analysis, we ignore the assignment. If later we find that q may hold a pointer, we must revisit the assignment to get a sound result.

Steensgaard illustrated his algorithm using the following program:

$$
\begin{array}{ll}
1: & a := \&x \\
2: & b := \&y \\
3: & \textbf{if } p \textbf{ then} \\
4: & \quad y := \&z \\
5: & \textbf{else} \\
6: & \quad y := \&x \\
7: & c := \&y
\end{array}
$$

Figure 114

His analysis produces the following graph for this program:

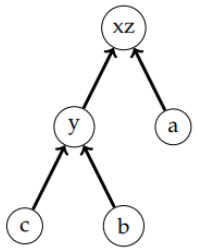


Figure 115

Rayside illustrates a situation in which Andersen must do more work than Steensgaard:

$$
\begin{array}{ll}
1: & q := \&x \\
2: & q := \&y \\
3: & p := q \\
4: & q := \&z
\end{array}
$$

Figure 116

After processing the first three statements, Steensgaard's algorithm will have unified variables x and y, with p and q both pointing to the unified node. In contrast, Andersen's algorithm will have both p and q pointing to both x and y. When the fourth statement is processed, Steensgaard's algorithm does only a constant amount of work, merging z in with the already-merged xy node. On the other hand, Andersen's algorithm must not just create a points-to relation from q to z, but must also propagate that relationship to p. It is this additional propagation step that results in the significant performance difference between these algorithms.

Analyzing Steensgaard's pointer analysis for efficiency, we observe that each of n statements in the program is processed once. The processing is linear, except for find operations on the unionfind data structure (which may take amortized time $O(\alpha(n))$ each) and the join operations. We note that in the join algorithm, the short-circuit test will fail at most $O(n)$ times—at most once for each variable in the program. Each time the short-circuit fails, two abstract locations are unified, at cost $O(\alpha(n))$. The unification assures the short-circuit will not fail again for one of these two variables. Because we have at most $O(n)$ operations and the amortized cost of each operation is at most $O(\alpha(n))$, the overall running time of the algorithm is near linear: $O(n * \alpha(n))$. Space consumption is linear, as no space is used beyond that used to represent abstract locations for all the variables in the program text.

Based on this asymptotic efficiency, Steensgaard's algorithm was run on a 1 million line program (Microsoft Word) in 1996; this was an order of magnitude greater scalability than other pointer analyses known at the time. Steensgaard's pointer analysis is field-insensitive; making it field-sensitive would mean that it is no longer linear.

17.7 Adding Context Sensitivity to Andersen's Algorithm

We can define a version of Andersen's points-to algorithm that is context-sensitive. In the following approach, we analyze each function separately for each calling point. The analysis keeps track of the current context, the calling point n of the current procedure. In the constraints, we track separate values for each variable x_n according to the calling context n of the procedure defining it, and we track separate values for each memory location l_n^k according to the calling context n active when that location was allocated at new instruction k. The rules are as follows:

$$\frac{n \vdash p := \mathbf{new}_k\ A}{l_n^k \in p_n}\ \textit{new}$$

$$\frac{n \vdash p := q \quad l_n \in q_n}{l_n \in p_n}\ \textit{copy}$$

$$\frac{n \vdash x.f := y \quad l_x \in x_n \quad l_y \in y_n}{l_y \in l_x.f}\ \textit{field-read}$$

$$\frac{n \vdash x := y.f \quad l_y \in y_n \quad l_z \in l_y.f}{l_z \in x_n}\ \textit{field-assign}$$

$$\frac{n \vdash f_k(y) \quad l_y \in y_n \quad [\![f(z) = e]\!] \in \textit{Program}}{l_y \in z_k \quad k \vdash e}\ \textit{call}$$

Figure 117

To illustrate this analysis, imagine we have the following code:

```
interface A { void g(); }
class B implements A { void g() { ... } }
class C implements A { void g() { ... } }
class D {
    A f(A a1) { return a1; }
}

// in main()
D d1 = new D();
if (...) {
    A x = d1.f(new B());
        x.g() // which g is called?
else
    A y = d1.f(new C());
        y.g() // which g is called?
```

Figure 118

The analysis produces the following aliasing graph:

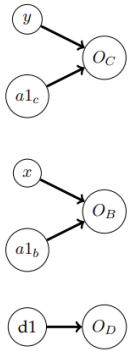


Figure 119

In this example, tracking two separate versions of the variable a1 is sufficient to distinguish the objects of type B and C as they are passed through method f, meaning that the analysis can accurately track which version of g is called in each program location.

Call-string context sensitivity has its limits, however. Consider the following example, adapted from notes by Ryder:

```
interface X { void g(); }
class Y implements X { void g() { ... } }
class Z implements X { void g() { ... } }
class A {
        X x;
        void setX(X v) { helper(v)^A; }
        void helper(X vh) { x = vh; }
        X getX() { return x; }
}

// in main()
A a1 = new A();    // allocates Oa1
A a2 = new A();    // allocates Oa2
a1.setX(new Y())^Y; // allocates OY
a2.setX(new Z())^Z; // allocates OZ
X x1 = a1.getX();
X x2 = a2.getX();
x1.g();            // which g() is called?
x2.g();            // which g() is called?
```

Figure 120

If we analyze this example with a 1-CFA style call-string sensitive pointer analysis, we get the following analysis results:

Context	Variable	Location	Notes
•	a1	Oa1	
•	a2	Oa2	
Y	this	Oa1	
Y	v	OY	
h	this	Oa1	
h	vh	OY	
Oa1	x	OY	
Z	this	Oa2	
Z	v	OZ	
h	this	Oa1,Oa2	updated
h	vh	OY, OZ	updated
Oa1	x	OY, OZ	updated
Oa2	x	OY, OZ	
•	x1	OY, OZ	
•	x1	OY, OZ	

Figure 121

Essentially, because of the helper method, one function call's worth of context sensitivity is insufficient to distinguish the calls to setX and helper for the objects Oa1 and Oa2. We could fix this by increasing context sensitivity, e.g. by going to a 2-CFA analysis that tracks call strings of length two. This has a very high cost in practice, however; 2-CFA does not scale well to large object-oriented programs.

A better solution comes from the insight that in the above example, call-strings are really tracking the wrong kind of context. What we need to do is distinguish between Oa1 and Oa2. In other words, the call chain does not matter so much; we want to be sensitive to the receiver object.

An alternative approach based on this idea is called object-sensitive analysis. It uses for the context not the call site, but rather the receiver object. In this case, we index everything not by a calling point n but instead by a receiver object l. The rules are as follows:

$$\frac{l \vdash p := \mathbf{new}_k\ A}{l_l^k \in p_l}\ \textit{new}$$

$$\frac{l \vdash p := q \quad l_l \in q_l}{l_l \in p_l}\ \textit{copy}$$

$$\frac{l \vdash x.f := y \quad l_x \in x_l \quad l_y \in y_l}{l_y \in l_x.f}\ \textit{field-read}$$

$$\frac{l \vdash x := y.f \quad l_y \in y_l \quad l_z \in l_y.f}{l_z \in x_l}\ \textit{field-assign}$$

$$\frac{l \vdash x.f(y) \quad l_x \in x_l \quad l_y \in y_l \quad [\![f(z) = e]\!] \in \textit{Program}}{l_x \in \mathbf{this}_{l_x} \quad l_y \in z_{l_x} \quad l_x \vdash e}\ \textit{call}$$

Figure 122

Now if we reanalyze the example above, we get:

Context	Variable	Location
•	a1	Oa1
•	a2	Oa2
Oa1	v	OY
Oa1	vh	OY
Oa1	x	OY
Oa2	v	OZ
Oa2	vh	OZ
Oa2	x	OZ
•	x1	OY
•	x1	OZ

Figure 123

In practice, object-sensitive analysis appears to be the best approach to context sensitivity in the pointer or call-graph construction analysis of object-oriented programs. Intuitively, it seems that organizing a program around objects makes the objects themselves the most interesting thing to analyze.

The state of the art implementation technique for points-to analysis of object-oriented programs was presented by Bravenboer and Smaragdakis in OOPSLA 2009. Their approach generates declarative Datalog code to represent the input program, and a datalog evaluation engine solves what are essentially declarative constraints to get the analysis result.

In an more recent POPL 2011 paper analyzing object-sensitivity, Smaragdakis, Bravenboer, and Lhotak demonstrate that it is more effective than call-string sensitivity. They also propose a ´ technique known as type-sensitive analysis which tracks only the type of the receiver (and, for depths ≥ 2, the type of the object that created the reciever, etc.), and show that type-sensitive analysis is nearly as precise as object-sensitive analysis and much more scalable.

18 Register Allocation

In this section[18] we discuss register allocation, which is one of the last steps in a compiler before code emission. Its task is to map the potentially unbounded numbers of variables or "temps" in pseudo-assembly to the actually available registers on the target machine. If not enough registers are available, some values must be saved to and restored from the stack, which is much less efficient than operating directly on registers. Register allocation is therefore of crucial importance in a compiler and has been the subject of much research. Register allocation is also covered thorougly in the textbook, but the algorithms described there are complicated and difficult to implement. We present here a simpler algorithm for register allocation based on chordal graph coloring due to Hack [19] and Pereira and Palsberg [PP05]. Pereira and Palsberg have demonstrated that this algorithm performs well on typical programs even when the interference graph is not chordal. The fact that we target the x86-64 family of processors also helps, because it has 16 general registers so register allocation is less important than for the x86 with only 8 registers (ignoring floating-point and other special purpose registers). Most of material below is based on Pereira and Palsberg [20], where further background, references, details, empirical evaluation, and examples can be found.

18.1 Graph Coloring

Register allocation via graph coloring, like linear-scan register allocation, considers allocating registers to variables across a whole procedure. However, it uses live ranges instead of live intervals. This addresses the problem mentioned above with linear scan allocation, namely that if there are "holes" in a variable's live interval, then it can be wasteful to tie up a register for the entire live interval (as illustrated below).

```
x = ...;
  .
  .
  .
use x;
  .    \
  .     => no use of x.  x will be overwritten anyway so we don't need
  .    /                 to keep its value in the register here.
x = ...;
  .
  .
  .
use x;
```

Figure 124: Motivation for graph coloring

A live range is a pair of the form: (¡variable¿, ¡set of CFG nodes¿). A live range for variable x is roughly all of the nodes of the control flow graph starting from a definition of x, up to all the uses of x reached by that definition. If two live ranges don't overlap then they can use the same register. For example:

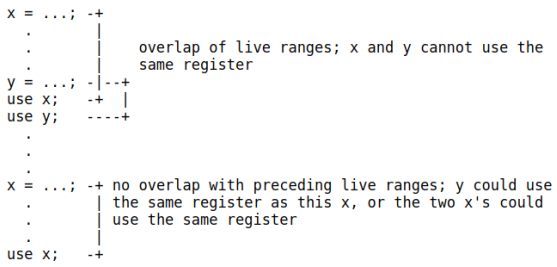


Figure 125: Motivation for graph coloring

The algorithm for global register allocation via graph coloring consists of 4 steps:

- Step 1: Compute live ranges
- Step 2: Build the interference graph
- Step 3: Color the graph
- Step 4: Convert colors to registers

18.1.1 Step 1: compute live ranges

- Build the CFG.
- Do reaching defs and live variable analysis. Note: the variables of interest are those that are candidates for registers. Variables that are not candidates might include:
  - variables that could be aliased:
    - variables that could be pointed to
    - globals (also, could be changed in calls to functions that won't know which register the global is in)
  - array elements (too hard to tell which element is being referred to)
  - structs/unions (too big to fit in a register, too hard to deal with individual fields)
  - floating-point values (too big to fit in a single register; also, on machines like the Sparc, floating-point registers are not saved across calls)

  What's left: locals that are scalar, and not floating point. This might include parameters, though they are sometimes more difficult to handle than "plain" locals.
- Build initial live ranges:
  - For each CFG node D that defines variable x, the initial live range for D consists of: $(< x >, < \{D\}$ `union` $\{N|$`x in N.live-before and D in N.reaching-defs-before`$\} >$) Note: the live range is a pair: the variable defined at D, and the set of nodes in the range.

– Convert initial live ranges to final live ranges (collapse overlapping initial live ranges for the same variable):

```
for each var x
   for each live range R for x
         if there is another live range R' for x such that R intersect R' != {}
         then R "absorbs" R' (i.e. R = R U R', R' goes away)
```

Figure 126

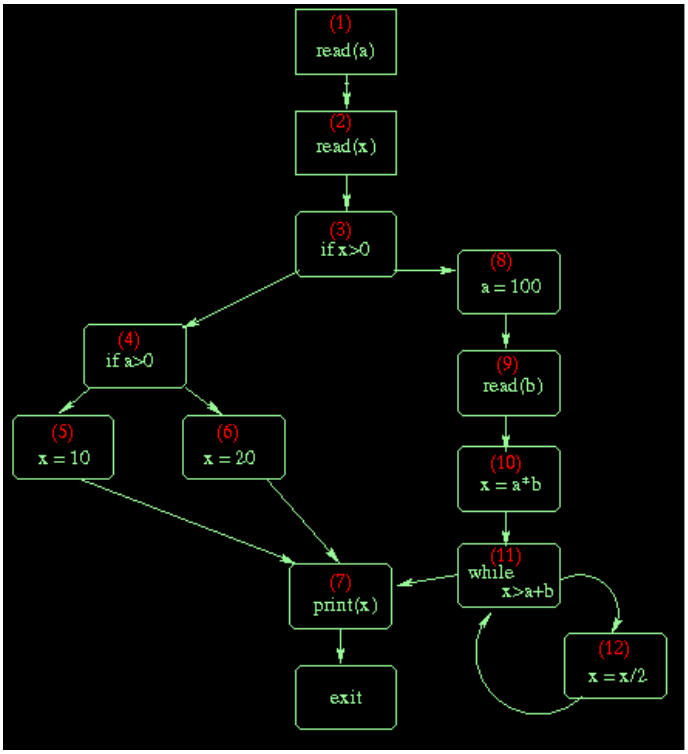


Figure 127

```
initial live ranges
-------------------
Def of a at node (1), {1, 2, 3, 4}
Def of x at node (2), {2, 3}
Def of x at node (5), {5, 7}
Def of x at node (6), {6, 7}
Def of a at node (8), {8, 9, 10, 11, 12}
Def of b at node (9), {9, 10, 11, 12}
Def of x at node (10), {7, 10, 11, 12}
Def of x at node (12), {7, 11, 12}

Final live ranges
-----------------
( <a>, {1, 2, 3, 4} )
( <x>, {2, 3} )
( <x>, {5, 6, 7, 10, 11, 12} )
( <a>, {8, 9, 10, 11, 12} )
( <b>, {9, 10, 11, 12} )
```

Figure 128

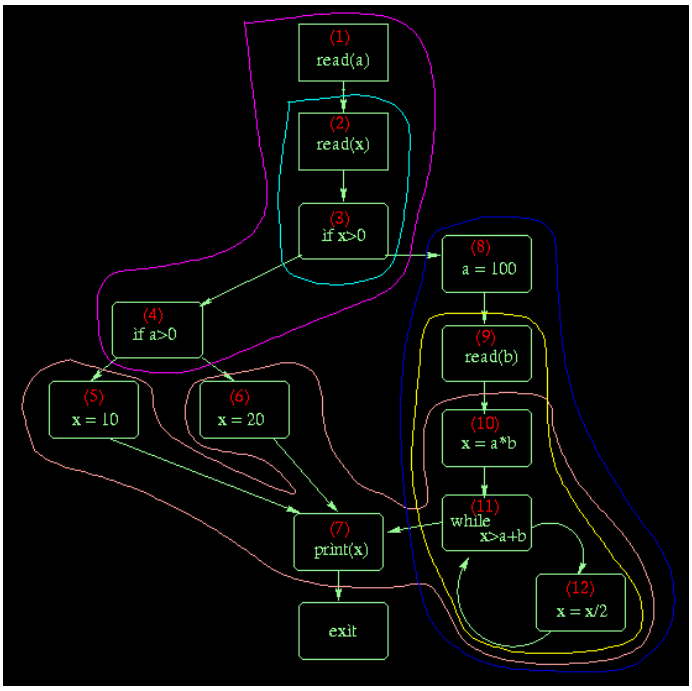


Figure 129

18.1.2 Step 2 - Build the Interference Graph

- 1 node for each live range

- 1 undirected edge n-m iff n intersect m != {}

Here is the graph for the live ranges shown above; the colors used above to encircle the live ranges are used to color the nodes of the interference graph.

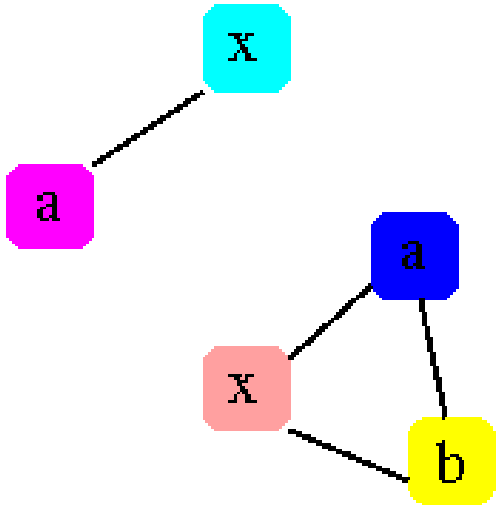


Figure 130

18.2 Register Allocation via Graph Coloring

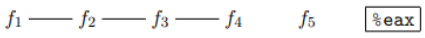


Figure 131

Once we have constructed the interference graph, we can pose the register allocation problem as follows: construct an assignment of K colors (representing K registers) to the nodes of the graph (representing variables) such that no two connected nodes are of the same color. If no such coloring exists, then we have to save some variables on the stack which is called spilling. Unfortunately, the problem whether an arbitrary graph is K-colorable is NP-complete for K $\geq$ 3. Chaitin[21] has proved that register allocation is also NP-complete by showing that for any graph G there exists some program which has G as its interference graph. In other words, one cannot hope for a theoretically optimal and efficient register allocation algorithm that works on all machine programs. Fortunately, in practice the situation is not so dire. One particularly important intermediate form is static single assignment (SSA). Hack[19] observed that for programs in SSA form, the interference graph always has a specific form called chordal. Coloring for chordal graphs can be accomplished in time $O(|V|+|E|)$ and is quite efficient in practice. Better yet, Pereira and Palsberg[20] noted that as

much as 95% of the programs occuring in practice have chordal interference graph. Moreover, using the algorithms designed for chordal graphs behaves well in practice even if the graph is not quite chordal. Finally, the algorithms needed for coloring chordal graphs are quite easy to implement compared, for example, to the complex algorithm in the textbook. You are, of course, free to choose any algorithm for register allocation you like, but we would suggest one based on chordal graphs explained in the remainder of this lecture.

18.3 Chordal Graphs

An undirected graph is chordal if every cycle with 4 or more nodes has a chord, that is, an edge not part of the cycle connecting two nodes on the cycle. Consider the following three examples:

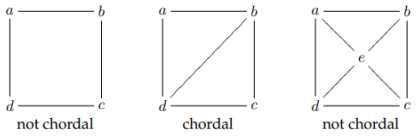


Figure 132

chordal chordal not chordal Only the second one is chordal. In the other two, the cycle abcd does not have a chord.

On chordal graphs, optimal coloring can be done in two phases, where optimal means using the minimum number of colors. In the first phase we determine a particular ordering of the nodes called simplicial elimination ordering, in the second phase we apply greedy coloring based on this order.

18.4 Simplicial Elimination Ordering

plicial Elimination Ordering A node v in a graph is simplicial if its neighborhood forms a clique, that is, all neighbors of v are connected to each other. An ordering v1, . . . , vn of the nodes in a graph is called a simplicial elimination ordering if every node vi is simplicial in the subgraph $v_1, \ldots, v_i$. Interestingly, a graph has a simplicial elimination ordering if and only if it is chordal. We can find a simplicial elimination ordering using maximum cardinality search, which can be implemented to run in $O(|V|+|E|)$ time. The algorithm associates a weight $wt(v)$ with each vertex which is initialized to 0 updated by the algorithm. We write $N(v)$ for the neighborhood of v, that is, the set of all adjacent nodes.

If the graph is not chordal, the algorithm will still return some ordering although it will not be simplicial. Such an ordering can still be used in the coloring phase, but does not guarantee that only the minimal numbers of colors will be used.

Algorithm: Maximum cardinality search
Input: $G = (V, E)$ with $|V| = n$
Output: A simplicial elimination ordering $v_1, \ldots, v_n$
For all $v \in V$ set $\mathrm{wt}(v) \leftarrow 0$
Let $W \leftarrow V$
For $i \leftarrow 1$ to n do
    Let v be a node of maximal weight in W
    Set $v_i \leftarrow v$
    For all $u \in W \cap N(v)$ set $\mathrm{wt}(u) \leftarrow \mathrm{wt}(u) + 1$
    Set $W \leftarrow W - \{v\}$

Figure 133

In our example 131,if we pick f_1 first, the weight of f_2 will become 1 and has to be picked second, followed by f_3 and f_4. Only f_5 is left and will come last, ignoring here %eax which is already colored. It is easy to see that this is indeed a simplicial elimination ordering. $f_2, f_4, f_3, \ldots$ is not, because the neighborhood of f3 in the subgraph f_2, f_4, f_3 does not form a clique.

18.5 Greedy Coloring

Given an ordering, we can apply greedy coloring by simply assigning colors to the vertices in order, always using the lowest available color. Initially, no colors are assigned to nodes in V . We write $\Delta(G)$ to the maximum outdegree of a node in G.

Algorithm: Greedy coloring
Input: $G = (V, E)$ and sequence $v_1, \ldots, v_n$.
Output: Assignment $\mathrm{col}(v) = c$, $0 \leq c \leq \Delta(G)$, $v \in V$.
For $i \leftarrow 1$ to n do
    Let c be the lowest color not used in $N(v_i)$
    Set $\mathrm{col}(v_i) \leftarrow c$

Figure 134

The algorithm will always assign at most $\Delta(G) + 1$ colors. If the ordering is a simplicial elimination ordering, the result is furthermore guaranteed to use the fewest possible colors.

In our example 131, we would just alternate color assigments:

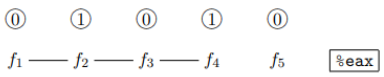


Figure 135

Of course, %eax is represented by one of the colors. Assuming this color is 0 and %edx is the name of register 1, we obtain the following program:

```
%eax ← 1
%edx ← 1
%eax ← %edx + %eax
%edx ← %eax + %edx
%eax ← %edx + %eax
%eax ← %eax
```

Figure 136

It should be apparent that some optimizations are possible. Some are immediate, such as the redundant move of a register to itself.

18.6 Register Spilling

So consider that we have applied the above coloring algorithm and it turns out that there are more colors needed than registers available. In that case we need to save some temporary values. In our runtime architecture, the stack is the obvious place. One convenient way to achieve this is to simply assign stack slots instead of registers to some of the colors. The choice of which colors to spill can have a drastic impact on the running time. Pereira and Palsberg suggest two heuristics: (i) spill the least-used color, and (ii) spill the highest color assigned by the greedy algorithm. For programs with loops and nested loops, it may also be significant where in the programs the variables or certain colors are used: keeping variables used frequently in inner loops may be crucial for certain programs. Once we have assigned stack slots to colors, it is easy to rewrite the code using temps that are spilled if we reserve a register in advance for moves to and from the stack when necessary. For example, if %r11 on the x86-64 is reserved to implement save and restore when necessary, then $t \leftarrow t + s$ where t is assigned to stack offset 8 and s to %eax can be rewritten to

```
%r11     ← 8(%rsp)
%r11     ← %r11 + %eax
8(%rsp)  ← %r11
```

Figure 137

Sometimes, this is unnecessary because some operations can be carried out directly with memory references. So the assembly code for the above could be shorter

```
ADDL %eax, 8(%rsp)
```

Figure 138

although it is not clear whether and how much more efficient this might be than a 3-instruction sequence

```
MOVL 8(%rsp), %r11
ADDL %eax, %r11
MOVL %r11, 8(%rsp)
```

Figure 139

We recommend generating the simplest uniform instruction sequences for spill code.

18.7 Register Coalescing

After register allocation, a common further optimization is used to eliminate register-to-register moves called register coalescing. Algorithms for register coalescing are usually tightly integrated with register allocation. In contrast, Pereira and Palsberg describe a relatively straightforward method that is performed entirely after graph coloring called greedy coalescing.

The algorithm considers each move between variables $t \leftarrow s$ occurring in the program in turn. If t and s they are the same color, the move can be eliminated without further action. If there is an edge between them, that is, they interfere, they cannot be coalesced. Otherwise, if there is a color c which is not used in the neighborhoods of t and s, $N(t)UN(s)$, and which is smaller than the number of available registers, then the variables t and s are coalesced into a single new variable u with color c. We create edges from u to any vertex in $N(t)UN(s)$ and remove t and s from the graph. Because of the tested condition, the resulting graph is still K-colored, where K is the number of available registers. Of course, we also need to eventually rewrite the program appropriately to maintain a correspondence with the graph.

18.8 Precolored Nodes

Some instructions on the x86-64, such as IDIV, require their arguments to be passed in specific registers and return their results also in specific registers. There are also call and ret instructions that use specific registers and must respect caller-save and callee-save register conventions. We will return to the issue of calling conventions later in the course. When generating code for a straight-line program as in the first lab, some care must be taken to save and restore callee-save registers in case they are needed. First, for code generation, the live range of the fixed registers should be limited to avoid possible correctness issues and simplify register allocation. Second, for register allocation, we can construct an elimination ordering as if all precolored nodes were listed first. This amounts to the initial weights of the ordinary vertices being set to the number of neighbors that are precolored before the maximum cardinality search algorithm starts. The resulting list may or may not be a simplicial elimination ordering, but we can nevertheless proceed with greedy coloring as before.

19 List Scheduling

Instruction scheduling[22] plays a critical role in determining the performance of compiled code on today's computers. Today's microprocessors rely on the compiler to hide memory latencies and to keep functional units busy—both are tasks for the instruction scheduler. On the microprocessors of tomorrow, the quality of instruction scheduling may be more important, since these machines will feature longer memory latencies and more functional units. List scheduling is the most widely used technique for instruction scheduling.

Before scheduling, the compiler applies a series of optimizations to the iloc code. This includes pointer analysis, dead code elimination, global value numbering, lazy code motion, constant propagation, strength reduction, register coalescing, dead code elimination, and empty block removal. For the purposes of this paper, no register allocation was performed; this eliminates interactions between allocation and scheduling and isolates the impact of scheduling.

After optimization, the compiler passes the code to the scheduler. Each block is scheduled individually. The first step constructs a data-precedence graph (DPG) for the block. The DPG $G = (N, E, E')$ has a node $n \in N$ for each operation. Edges $e = (n_i, n_j) \in E$ represent dependences between operations; their direction matches the flow of values. Edges in E' represent anti-dependences in the code that prevent reordering. An anti-edge $e = (n_i, n_j) \in E'$ indicates that moving n_j before n_i would change the flow of values because of a name that n_i uses and n_j redefines. The details of the individual schedulers vary.

To evaluate the schedules, we use several variations on a simple processor model. Each architecture consists of k identical pipelined functional units. Each functional unit can execute any iloc operation. For our experiments, we vary k between one and three. Each iloc operation has a latency—the number of cycles required before its results are available. Register values are read in the cycle when the instruction begins execution, and results are defined in the last cycle of its latency. Thus, an operation u can begin execution when all operations $v|(v, u) \in E$ have completed, and all operations $w|(u, w) \in E'$ have already been issued.

19.1 Data-precedence graph

A data dependency[23] in computer science is a situation in which a program statement (instruction) refers to the data of a preceding statement. In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

19.1.1 Flow dependency (True dependency)

A Flow dependency, also known as a data dependency or true dependency or read-after-write (RAW), occurs when an instruction depends on the result of a previous instruction: also known as name dependency

1. A = 3
2. B = A
3. C = B

Instruction 3 is truly dependent on instruction 2, as the final value of C depends on the instruction updating B. Instruction 2 is truly dependent on instruction 1, as the final value of B depends on the instruction updating A. Since instruction 3 is truly dependent upon instruction 2 and instruction 2 is

truly dependent on instruction 1, instruction 3 is also truly dependent on instruction 1. Instruction level parallelism is therefore not an option in this example.

19.1.2 Anti-dependency

An anti-dependency, also known as write-after-read (WAR), occurs when an instruction requires a value that is later updated. In the following example, instruction 2 anti-depends on instruction 3 — the ordering of these instructions cannot be changed, nor can they be executed in parallel (possibly changing the instruction ordering), as this would affect the final value of A.

1. B = 3
2. A = B + 1
3. B = 7

Example :

```
MUL R3,R1,R2
ADD R2,R5,R6
```

It is clear that there is anti-dependence between these 2 instructions. At first we read R2 then in second instruction we are Writing a new value for it.

An anti-dependency is an example of a name dependency. That is, renaming of variables could remove the dependency, as in the next example:

1. B = 3
N. B2 = B
2. A = B2 + 1
3. B = 7

A new variable, B2, has been declared as a copy of B in a new instruction, instruction N. The anti-dependency between 2 and 3 has been removed, meaning that these instructions may now be executed in parallel. However, the modification has introduced a new dependency: instruction 2 is now truly dependent on instruction N, which is truly dependent upon instruction 1. As flow dependencies, these new dependencies are impossible to safely remove.

19.1.3 Output dependency

An output dependency, also known as write-after-write (WAW), occurs when the ordering of instructions will affect the final output value of a variable. In the example below, there is an output dependency between instructions 3 and 1 — changing the ordering of instructions in this example will change the final value of A, thus these instructions cannot be executed in parallel.

1. B = 3
2. A = B + 1
3. B = 7

As with anti-dependencies, output dependencies are name dependencies. That is, they may be removed through renaming of variables, as in the below modification of the above example:

1. B2 = 3
2. A = B2 + 1
3. B = 7

A commonly used naming convention for data dependencies is the following: Read-after-Write or RAW (flow dependency), Write-After-Read or WAR (anti-dependency), or Write-after-Write or WAW (output dependency).

19.2 The List Scheduling Algorithm

Here we describe our implementation of list scheduling. First, the dpg is built as described in the previous section. Next, priorities are assigned to each node in the graph. There are several different heuristics that can be used to assign priorities. A common and effective strategy is to use the latency weighted depth of the node [24, 25]. The depth of a node n is the length (number of nodes) of the longest path in the dpg from n to some leaf (including n and the leaf.) The latency weighted depth is computed the same way, but the nodes along the path are weighted using the latency of the operation the node represents. The following formula summarizes the priority computation for a node n:

$$\text{priority}(n) = \max\left(\forall_{l \in \text{ leaves } (DPG)} \forall_{p \in \text{paths}(n,\ldots,l)} \sum_{p_i=n}^{l} \text{latency } (p_i)\right)$$

Dynamic programming can be used to compute the priorities efficiently, and we take into consideration the anti-edges described above:

$$\text{priority}(n) = \begin{cases} \text{latency}(n) & \text{if } n \text{ is a leaf.} \\ \max\big(\text{latency}(n) + \max_{(m,n)\in E}(\text{ priority } (m)), & \\ \quad \max_{(m,n)\in E'}(\text{ priority } (m))\big) & \text{otherwise.} \end{cases}$$

The final phase is the actual list scheduling algorithm that constructs the schedule for the block. Starting at cycle 0, the list scheduler places operations into the schedule cycle by cycle. Any operation that is "ready" at cycle X (i.e. all its operands have been computed), is a candidate to be scheduled at cycle X. The priorities computed in the previous step are used to determine which ready operation to schedule, by selecting the highest priority operation first. Any tie in the priority of two operations is broken arbitrarily. The algorithm is detailed in Figure140. Through the rest of the paper we refer to this algorithm as ls.

```
Algorithm:

cycle = 0
ready-list = root nodes in DPG
inflight-list = empty list
while ( ready-list or inflight-list not empty, and an issue slot is available )
      for op = (all nodes in ready-list in descending priority order)
            if (a functional unit exists for op to start at cycle)
                  remove op from ready-list and add to inflight-list
                  add op to schedule at time cycle
                  if (op has an outgoing anti-edge)
                        Add all targets of op's anti-edges that are ready to ready-list
                  endif
            endif
      endfor
      cycle = cycle + 1
      for op = (all nodes in inflight-list)
            if (op finishes at time cycle)
                  remove op from inflight-list
                  check nodes waiting for op in DPG and add to ready-list
                        if all operands available
            endif
      endfor
endwhile
```

Figure 140: List Scheduling algorithm

19.3 List Scheduling Alternatives

Here we present two alternatives to the ls algorithm discussed in the last section. For a survey of scheduling techniques see . A machine learning approach to scheduling has been developed by Moss and others.

19.3.1 Random Tie Breaking

A traditional list scheduler returns a single solution by breaking any ties in the priority of two or more operations arbitrarily. By running the list scheduler several times and breaking ties randomly, we could potentially generate more and better solutions. Figure141 is an example from the tomcatv benchmark. Assume all load immediates (LDI) take one cycle, all add operations (ADD) take two cycles, and the copy (i2i) takes one cycle. Assume we are scheduling on a machine with two identical functional units. The numbers next to the operations are the priority values that list scheduling uses. In this figure we see two different list schedules that could be generated from the dpg. The second one requires one less cycle. The critical decision comes in the second cycle, where the tie between the LDId and LDIc must be broken. Scheduling LDId early enough results in a shorter schedule.

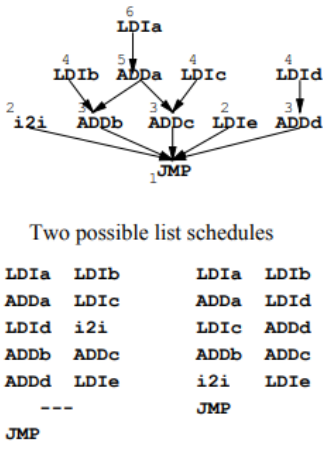


Figure 141: Example block from tomcatv

19.3.2 Backward list scheduling

In addition, there are some blocks for which a backward list scheduler can generate a better solution. A backward list scheduler works by reversing the direction of all edges in dpg, and scheduling the finish times of each operation. (Note that the start time of operations must be used to ensure enough available functional units for a given cycle.) This technique tends to cluster operations toward the end of the schedule instead of the beginning like a forward list scheduler. For an example of a block that benefits from backward list scheduling see Figure 142, which shows a block from the go benchmark. Assume there are two integer units that can execute the LDI operations (one cycle), the LSL operation (one cycle), the ADD operations (two cycles), ADDI operation (one cycle), and the CMP operation (one cycle). A separate memory unit executes the ST operations (four cycles). All functional units are completely pipelined. A forward list scheduler will schedule the four LDI operations and the the LSL before scheduling any of the ADD operations. This delays the start of the higher latency store operations (ST). A better schedule can be found by a backward list scheduler as shown in the example.

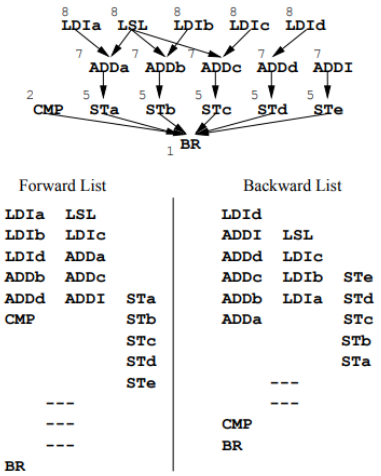


Figure 142: Example block from go, showing the benefits of backward list scheduling

19.4 Iterative Repair Scheduling

Here we introduce the application of a repair based scheduling technique called "iterative repair" to the problem of instruction scheduling in a compiler. This algorithm comes from the AI community and is described by Lin and Kernighan[26], and Zweben, et. al.[27, 28]. The technique has shown promise for several scheduling problems including space shuttle mission scheduling.

The generalized algorithm is presented in figure143. The idea is straightforward. First, create an instruction schedule that begins each operation as early as possible with respect to the precedence constraints of the dpg, but ignores the resource constraints imposed by the limited number of processing elements. Now "repair" the schedule by moving operations that have a resource conflict to a point later in the schedule. This reduces the number of resource conflicts for the cycle being repaired. A resource conflict is simply a point in the schedule where more operations are scheduled than the available number of functional units. The earliest cycle with a conflict is found, and one of the conflicting operations is selected (line (1) in the algorithm). This operation and all operations that depend on it are removed from the schedule (called unscheduling). The selected operation and its dependent operations are then inserted back into the schedule (called rescheduling) at a later point (line (2) in the algorithm). We continue repairing the schedule until there are no more resource conflicts. The algorithm is run a "user specified" number of times, and the shortest schedule over all the trials is selected as the final schedule.

We have tested several new variations of the iterative repair scheduling algorithm. The most effective one to date we refer to as ir-bias. In ir-bias the selection of which node to move (called the move-node) is not completely random. Rather, operations with lower priority values (the same priority values as used by the list scheduler) are more likely to be moved. The selection is probabilistic; the probability that a node is selected is inversely related to its priority.

The move-node is scheduled one cycle later than its original position. All successor nodes are rescheduled as early-as-possible with respect to this new start time. This could cause additional conflicts to be created later in the schedule, but a future repair will correct any new conflicts. After

each repair we compare the length of the new schedule to that of the old schedule. If the new length is greater, the repair is ignored, the state of the previous schedule is restored, and a new move-node is selected. A new schedule with a greater length than the previous schedule is kept ten per cent of the time to avoid local minima.

Input: Data Precedence Graph. Parameters of machine (instruction latencies, pipelining, number of functional units, etc.). The number of iterations to perform *iter*.

Output: A schedule containing all nodes in the graph that satisfies the precedence constraints in the DPG and the resource constraints of the machine.

Algorithm:

min = largest integer
shortest is a schedule initially empty
for *x* = (1 to *iter*)
    Create an initial *schedule* by scheduling all operations as early as possible subject
        to precedence constraints.
    while (there exist resource conflicts in *schedule*)
        conflict_time = the cycle of the first resource conflict in the schedule
(1)   select an *operation* that has a resource conflict at *conflict_time*
        unschedule *operation* and all its successors in DPG
(2)   reschedule *operation* and its successors later in *schedule*.
    endwhile
    if length of *schedule* is less than *min*
        then *min* = length of *schedule*
        shortest = *schedule*
    endif
endfor

Figure 143: Basic Iterative Repair Scheduling algorithm

20 Dynamic Code Optimization

Most materials are based on [29, 30] in this section.

Dynamic compilation systems explore an interesting tradeoff. On one hand, we would like to have code performance that is comparable to static compilation techniques. However, we would also like to avoid long startup delays, long latencies, and slow responsiveness, which implies that the dynamic compiler should be fast.

Many dynamic compilation systems attack this problem by using an interpreter and an optimizing compiler. They begin by interpreting the code, and when the execution count for the method reaches a certain threshold or by some other heuristic, they use the optimizing compiler to dynamically compile the code for the method[31, 32] . Some systems use a fast code generator (baseline compiler) rather than an interpreter [33, 34].

The problem with these systems is that the execution speed of the interpreted or baseline compiled code is significantly worse than that of fully optimized code — typically 30% to ten times slower for baseline compiled code [33, 34] and ten to a hundred times slower for interpreted code [31, 32]. Therefore, we would like to transfer into the optimized version as quickly as possible. However, the optimizing compiler can take a long time to compile. Waiting for the optimizing compiler to finish hurts program startup and response times. Some systems use a multi-level compilation approach, whereby they progress through a number of different compilation "levels", and thereby slowly "accelerate" into optimized execution [32, 35]. However, this simply exacerbates the problem of having a long delay until the program runs at full speed.

Unlike interpretation, compilation takes time that is proportional to the amount of code that is being compiled. Many analyses and optimizations are superlinear in the size of the code (basic blocks, instructions, registers, etc.) This can cause the compilation time to increase significantly as the amount of code being compiled gets large. Compilation of large amounts of code is the cause of undesirably long compilation times.

However, when compiling a method at a time, we do not really have much choice in the matter. Some methods are large to begin with, and others grow large after performing inlining. Even when being frugal and inlining only when it will make a noticeable difference in performance, methods can still grow large, and excessively restricting inlining can significantly hurt performance [31].

The root of the problem is that method boundaries do not correspond to the code that would most benefit from optimizing compilation. Even "hot" methods typically contain some code that is rarely or never executed, but often contain frequently-executed call sites to methods (which in turn, contain their own rarely-executed code.) Figure 144 contains a paraphrased example from the spec db benchmark. In this example, the `readdb` method is hot due to the while loop that it contains. However, the error handling code guarded by the if and the exception handler are rarely executed. Likewise, the call to `read()` is in the loop and therefore a good candidate for inlining. However, `read()` itself contains rarely-executed error handling code. The region that is important to compile — the while loop and the hot path in `read()` — have nothing to do with the method boundaries. Using a method granularity causes the compiler to waste time compiling and optimizing large pieces of code that do not matter.

```
void read_db(String fn) {
  int n = 0, act = 0;
  byte buffer[] = null;
  try {
    FileInputStream sif = new FileInputStream(fn);
    n = sif.getContentLength();
    buffer = new byte[n];
    int b;
    while ((b = sif.read(buffer, act, n-act))>0){
      act = act + b;
    }
    sif.close();
    if (act != n) {
      /* lots of error handling code, rare */
    }
  } catch (IOException ioe) {
    /* lots of error handling code, rare */
  }
}

int read(byte b[], int off, int len) {
  try {
    /* ... */
  } catch (IOException ioe) {
    /* lots of error handling code, rare */
  }
}
```

Figure 144: From spec db. Method boundaries do not correspond well to where the time is actually spent.

John Whaley[29] describes a technique to selectively compile and optimize partial methods. This gives us much better control over what we spend time compiling and optimizing. This technique uses dynamic profile data to make a prediction of what code will actually be executed, and selectively compiles and optimizes only that code. If the program actually attempts to branch to code that was not compiled (so-called "rare code"), the system falls back to interpretation or another dynamically compiled version.

20.1 Partial Method Compilation

view The general idea of the technique is to replace all entries into rare blocks with stubs that transfer control to the interpreter. The rare blocks are completely removed from the compiler's intermediate representation. Only very minimal changes to the compiler are necessary; optimizations can optionally use rare block information to attempt to better optimize the common paths. At the end of compilation, we store a map corresponding to each interpreter transfer point, which specifies how to reconstruct the interpreter state at that point.

We now describe each step of the process in detail.

1. Based on profile data, determine the set of rare blocks. The entry points of the rare basic blocks are mapped to abstract program locations, which then used to mark basic blocks as rare in the compiler's intermediate representation.

2. Perform live variable analysis. Before any transformations are performed, we perform live variable analysis to determine the set of live variables at rare block entry points.

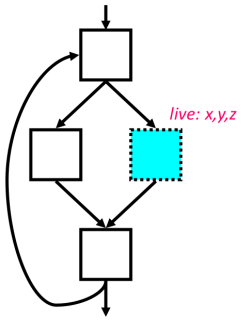


Figure 145: Determine the set of live variables at rare block entry points.

3. Redirect the control flow edges that targeted rare blocks, and remove the rare blocks. For each control flow edge from a non-rare block to a rare block, we generate a new basic block containing a single instruction that transfers control to the interpreter. This instruction uses all local variables and Java stack locations from the Java bytecode that are live at that point. We redirect the control flow edge to point to this new block, and add an edge from the new block to the exit node. See Figure 146 for an example. After this process, rare blocks can be removed as unreachable code.

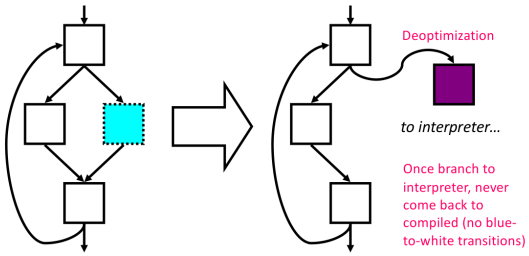


Figure 146: An example of redirecting the rare path. On the left, the dotted block is rare, so we redirect it to a block that calls the interpreter.

4. Perform compilation normally. All analysis, optimization, and code generation proceeds normally. Analyses treat the interpreter transfer point as an unanalyzable method call.

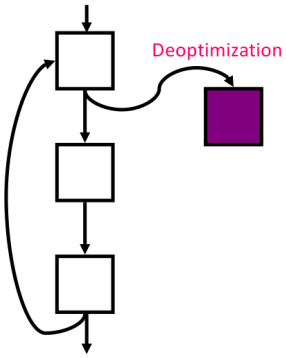


Figure 147: Analyses treat the interpreter transfer point as an unanalyzable method call.

5. Record a map for each interpreter transfer point. When generating the code to call the glue routine, we also generate a map that specifies the location, in registers or memory, of each of the local variables and Java stack locations used in the original Java bytecode. This map is used by the glue routine to reconstruct the interpreter state. The map is stored immediately after the call in the instruction stream. Each map is typically under 100 bytes long.

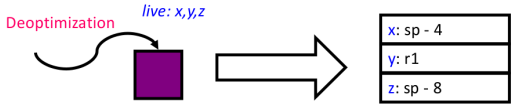


Figure 148: In code generation, generate a map that specifies the location, in registers or memory, of each of the live variables used to reconstruct the interpreter state

20.2 Partial dead code elimination

We modified our dead code elimination algorithm to treat rare blocks specially. This allows us to move computation that is only live on a rare path into the rare block, saving computation in the common case. Our dead code elimination uses an optimistic approach similar to the one described by Muchnick [36], originally due to Kennedy [37]. That analysis begins by marking all instructions that compute essential values, and then recursively marking all instructions that contribute to the computation of essential values. Any non-essential instructions are then eliminated.

Our analysis operates on SSA form. It first computes the essential instructions in all non-rare blocks, completely ignoring all rare blocks. An essential instruction computes a value that is used in a predicate, returned or output by the method, or has a potential side-effect. It then visits each rare block to discover instructions that are essential for that rare block, but not essential for non-rare blocks. If these instructions are recomputable at the point of the rare block, they can be safely copied there.

For each instruction in the rare block, it recursively visits all instructions that contribute to the computation of values for that instruction. If an instruction is marked as essential, it is skipped. If it is a Φ function, it depends on an earlier predicate, and is therefore (recursively) marked as essential. Otherwise, the instruction is added to a set of instructions associated with the rare block.

After computing sets for all rare blocks, it adds each of the non-essential instructions in a set to its corresponding rare block. Then, all instructions in non-rare blocks that are not marked as essential are eliminated.

Now we make an argument of correctness. Any instruction that was eliminated on the main path either computed a value that was not essential anywhere, in which case it is obviously correct to eliminate it, or it was only essential in some number of rare blocks, in which case it would have been copied into those rare blocks. Copying the instruction into a rare block is legal because, as the instruction is not a Φ function, the instruction dominates and is in the same loop as the rare block and therefore would have executed exactly once. Also, any instruction with a potential side effect or that read from or wrote to memory would have been marked as essential on the main path and therefore executed in its original location. Therefore, moving the instruction to a rare block does not violate exception or memory semantics.

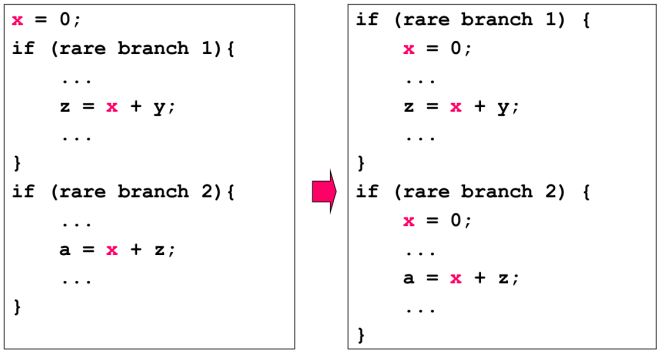


Figure 149: Partial Dead Code Example.

20.3 Escape Analysis[30]

Escape Analysis checks whether an allocated object escapes (i.e., can be used outside) the allocating method or thread. This happens, for example, if it is assigned to a global variable or heap object, or if it is passed as a parameter to some other method. Compilers use Escape Analysis to determine the dynamic scope and the lifetime of allocated objects. The result of this analysis allows the compiler to perform numerous optimizations on operations such as object allocations, synchronization primitives and field accesses.

Figure 150 shows a small piece of code that will serve as an example to show the benefits of Escape Analysis: The `getValue` method creates a new `Key` object and checks whether it is in the cache. If so, the method returns the cached value. Otherwise, it creates and returns a new value (the method `createValue` is not discussed here).

```
class Key {
  int idx;
  Object ref;
  Key(int idx, Object ref) {
    this.idx = idx;
    this.ref = ref;
  }
  synchronized boolean equals(Key
      other) {
    return idx == other.idx &&
           ref == other.ref;
  }
}
static CacheKey cacheKey;
static Object cacheValue;

Object getValue(int idx, Object ref) {
  Key key = new Key(idx, ref);
  if (key.equals(cacheKey)) {
    return cacheValue;
  } else {
    return createValue(...);
  }
}
```

Figure 150: Simple example.

When `getValue` is compiled, the compiler will most likely perform some inlining, which might cause the actually compiled code to look like 151. The Key constructor and the equals method have been inlined into the getValue method, and a `synchronized` block was created to achieve synchronization on the inlined equals method.

```
Object getValue(int idx, Object ref) {
  Key key = alloc Key;
  key.idx = idx;
  key.ref = ref;
  Key tmp1 = cacheKey;
  boolean tmp2;
  synchronized (key) {
    tmp2 = key.idx == tmp1.idx &&
           key.ref == tmp1.ref;
  }
  if (tmp2) {
    return cacheValue;
  } else {
    return createValue(...);
  }
}
```

Figure 151: Example from Figure 150 after inlining.

When Escape Analysis examines the resulting method, it will come to the conclusion that no reference to the allocated `Key` object escapes from the current compilation scope.

This implies that no references to the object exist after the method has returned, and that no other thread can ever see a reference to this object. The compiler can use these observations to perform a number of optimizations:

- The allocation of the object on the garbage collected heap can be replaced with allocation on the stack or in other non-garbage-collected allocation areas such as zones.
- Scalar Replacement can be used to eliminate the allocation altogether, by replacing the fields of the object with local variables.
- Since the object's lock will never be contended,Lock Elision can remove the synchronization on `key`.

If the compiler uses Scalar Replacement and Lock Elision, the result might look like in Figure 152. The allocation was replaced with the local variables `idx1` and `ref1`, and the `synchronized` statement was removed entirely.

```
1  Object getValue(int idx, Object ref) {
2    int idx1 = idx;
3    Object ref1 = ref;
4    Key tmp = cacheKey;
5    if (idx1 == tmp.idx && ref1 ==
         tmp.ref) {
6      return cacheValue;
7    } else {
8      return createValue(...);
9    }
10 }
```

Figure 152: Example from Figure 151 after inlining.

Traditionally, Escape Analysis uses algorithms such as Equi-Escape Sets [38] to determine which objects escape from the scope. These algorithms build sets of objects that have the same escape state, with each object initially being in a separate set. By analyzing all operations in the method the system can merge sets (e.g., when an object in one set is assigned to a field of an object in another set), or mark a set as escaping (e.g., when an object in this set is assigned to a global variable).

20.4 PARTIAL ESCAPE ANALYSIS

Escape Analysis allows a compiler to determine whether an object is accessible outside the allocating method or thread. This information is used to perform optimizations such as Scalar Replacement, Stack Allocation and Lock Elision, allowing modern dynamic compilers to remove some of the abstractions introduced by advanced programming models.

State-of-the-art Virtual Machines employ techniques such as advanced garbage collection, alias analysis and biased locking to make working with dynamically allocated objects as efficient as possible. But even if allocation is cheap, it still incurs some overhead. Even if alias analysis can

remove most object accesses, some of them cannot be removed. And although acquiring a biased lock is simple, it is still more complex than not acquiring a lock at all.

Escape Analysis can be used to determine whether an object needs to be allocated at all, and whether its lock can ever be contended. This can help the compiler to get rid of the object's allocation, using Scalar Replacement to replace the object's fields with local variables.

Escape Analysis checks whether an object escapes its allocating method, i.e., whether it is accessible outside this method. An object escapes, for example, if it is assigned to a static field, if it is passed as an argument to another method, or if it is returned by a method. In these cases the object needs to exist on the heap, because it will be accessed as an object in some other context.

In many cases, however, an object escapes just in a single unlikely branch. Nevertheless, this prevents optimizations. Therefore, we suggest a flow-sensitive Escape Analysis which we call Partial Escape Analysis.

The idea behind Partial Escape Analysis is to perform optimizations such as Scalar Replacement in branches where the object does not escape, and make sure that the object exists in the heap in branches where it does escape.

In many cases, making a global decision about the escapability of objects does not allow the compiler to perform the above optimizations. For example, the object allocated in Figure 157 escapes into the global variable `cacheKey`, so that Escape Analysis would consider it to be escaping.

```
1  Object getValue(int idx, Object ref) {
2    Key key = new Key(idx, ref);
3    if (key.equals(cacheKey)) {
4      return cacheValue;
5    } else {
6      cacheKey = key;
7      cacheValue = createValue(...);
8      return cacheValue;
9    }
10 }
```

Figure 153: Complex example.

However, if we only consider the path through the true branch of the if statement, the object does not escape. Analyzing the escapability of objects for individual branches is called Partial Escape Analysis. Partial Escape Analysis iterates over the code and maintains the current escape state and the current contents of allocated objects during this process. Initially, each allocated object is in the state `virtual`, which means that there was no reason yet to actually allocate it. As the algorithm progresses along the control flow, it updates this state when instructions operate on the allocated object.

The transition from Figure 154 to Figure 155 shows how Partial Escape Analysis lets the compiler optimize the code in this example:

```
1  Object getValue(int idx, Object ref) {
2    Key key = alloc Key;
3    key.idx = idx;
4    key.ref = ref;
5    Key tmp1 = cacheKey;
6    boolean tmp2;
7    synchronized (key) {
8      tmp2 = key.idx == tmp1.idx &&
9             key.ref == tmp1.ref;
10   }
11   if (tmp2) {
12     return cacheValue;
13   } else {
14     cacheKey = key;
15     cacheValue = createValue(...);
16     return cacheValue;
17   }
18 }
```

Figure 154: Example from 157 after inlining.

```
1  Object getValue(int idx, Object ref) {
2    Key tmp = cacheKey;
3    if (idx == tmp.idx && ref ==
         tmp.ref) {
4      return cacheValue;
5    } else {
6      Key key = alloc Key;
7      key.idx = idx;
8      key.ref = ref;
9      cacheKey = key;
10     cacheValue = createValue(...);
11     return cacheValue;
12   }
13 }
```

Figure 155: Example from 154 after Partial Escape Analysis.

- The allocation in line 2 is removed, and an entry for this object is created that specifies that it is `virtual` and that all fields have their default values.
- The assignments to the fields `idx` and `ref` in lines 3 and 4 are removed, and their effects are remembered by updating the object's field states.
- When entering the `synchronized` region in line 7, the object is still `virtual`. The monitor enter operation is removed, and the object's state is augmented with a `locked` flag that specifies that this object would have been `locked` if it actually existed at this point.
- The accesses to the `idx` and `ref` fields of the `virtual` object in lines 8 and 9 can be replaced using the object's current field states.
- When exiting the `synchronized` region in line 10, the object is still `virtual`. Thus, the monitor exit operation is removed, and the `locked` flag is removed from the object's state.

- At the `if` statement in line 11, a copy of the current state is created, because it has to be propagated to both successors of this control split.
- When continuing at line 12, the object is still `virtual`, and the return statement ends the processing of this branch.
- When continuing at line 14, the object is still `virtual`, but the assignment to the static field `cacheKey` lets the object escape. In order for it to escape, it needs to exist, and therefore the object needs to be created and initialized with the current state of its fields at this point. This process is called materialization in our system. The object is transitioned to the state escaped at this point, and the state of its fields cannot be used from here on since there could be assignments to the fields from outside the compilation scope.
- Lines 15 and 16 do not affect the state of the object anymore.

In effect, the allocation was moved into one branch of the if statement. While this did not lead to fewer allocation sites in the resulting code, it reduces the dynamic number of allocations at runtime. The actual reduction depends on the likelihood of the branch containing the allocation being reached, but there will always be at most as many dynamic allocations as in the original code.

21 Domian Specific Language

21.1 Introduction

A domain-specific language is a computer programming language of restricted expressiveness focused on a particular domain[39]. DSLs are in widespread use in a variety of domains and are becoming more popular. Examples of widely used DSLs are TeX and LaTeX for typesetting academic papers, SQL for database querying, Rails for web application development and VHDL for hardware design. OpenGL can also be viewed as a DSL. By exposing an interface for specifying polygons and the rules to shade them, OpenGL created a high-level programming model for real-time graphics decoupled from the hardware or software used to render it, allowing for aggressive performance gains as graphics hardware evolves. The use of DSLs can provide significant gains in the productivity and creativity of application developers, the portability of applications, and application performance. A programmer using one or more of these DSLs writes her programs using domain-specific notation and constructs. The programs appear sequential and all parallelism and use of the heterogeneous machine resources is implicit. DSLs raise the level of abstraction and can provide a sequential model which satisfies the productivity goal.

An additional benefit of using a domain-specific approach is the ability to use domain knowledge to apply static and dynamic optimizations to a program written using a DSL. Most of these domainspecific optimizations would not be possible if the program was written in a general-purpose language. General-purpose languages are limited when it comes to optimization for at least two reasons. First, they must produce correct code across a very wide range of applications. This makes it difficult to apply aggressive optimizations. Compiler developers must err on the side of correctness. Second, because of the general-purpose nature needed to support a wide range of applications (e.g. financial, gaming, image processing, etc.), compilers can usually infer little about the structure of the data or the nature of the algorithms the code is using. DSLs on the other hand, with their expressive power and knowledge of the domain's data structures and algorithms make such optimizations feasible. This makes DSLs a good choice to deliver on our performance goal.

Since interesting applications might leverage a variety of DSLs, it is critical to not only simplify the development of DSLs by creating a shared infrastructure, but also to allow these DSLs to interoperate.

The ability to easily embed DSLs simplifies the task of a DSL developer. However, assistance in parallelizing and targeting heterogeneous resources is also needed.

21.2 DSLS FOR HETEROGENEOUS PARALLELISM[40]

In this section we briefly illustrate the benefits of using DSLs for achieving both productivity and portable parallel performance in a heterogeneous environment. We will use OptiML [41], a DSL for machine learning, as a running example. We then address the common challenges faced when designing and building a new DSL targeted to heterogeneous parallelism.

21.2.1 DSL productivity

At the forefront of DSL design is the ability to exploit domain knowledge to provide constructs that express domain operations at a higher level of abstraction. As a consequence of working at this abstraction level much of the lower-level implementation details are provided by the DSL itself

rather than the application programmer. This often results in a significant reduction in total number of lines of code as well as improved code readability compared to a general-purpose language.

As an example, consider the snippet of OptiML code shown in Figire 157, which shows the core of a downsampling application. In contrast to the C++ implementation shown in Figure 157, the OptiML version concisely expresses what should be accomplished rather than how it should be accomplished.

```
1  val distances =
2    Stream[Double](data.numRows, data.numRows) {
3      (i,j) => dist(data(i), data(j))
4    }
5  for (row <- distances.rows) {
6    if(densities(row.index) == 0) {
7      val neighbors = row find {_ < apprxWidth}
8      densities(neighbors) = row count {_ < kernelWidth}
9    }
10 }
```

Figure 156: Downsampling in OptiML

```
11 #pragma omp parallel for shared(densities)
12 for (size_t i=0; i<obs; i++) {
13   if (densities[i] > 0)
14     continue;
15   // Keep track on observations we can approximate
16   std::vector<size_t> apprxs;
17   Data_t *point = &data[i*dim];
18   Count_t c = 0;
19   for (size_t j=0; j<obs; j++) {
20     Dist_t d = distance(point, &data[j*dim], dim);
21     if (d < apprx_width) {
22       apprxs.push_back(j);
23       c++;
24     } else if (d < kernel_width) c++;
25   }
26   for (size_t j=0; j<apprxs.size(); j++)
27     densities[apprxs[j]] = c;
28   densities[i] = c;
29 }
```

Figure 157: Downsampling in C++

21.2.2 Portable parallel performance

In addition to providing a means of writing concise, maintainable code, DSLs can also expose significantly more semantic information about the application than a generalpurpose language. In particular domain constructs can expose structured, coarse-grained parallelism within an application. The DSL developer must identify the mapping between domain constructs and known parallel patterns, and with the proper restrictions this allows the DSL to generate safe and efficient low-level parallel code from application source using a sequential programming model.

As an example consider the OptiML sum construct (shown in 158). Summations occur quite frequently in machine learning applications that focus on condensing large input datasets into concise, useful output. The construct allows the user to supply an anonymous function producing the elements to be summed that is subject to the restricted semantics enforced by the OptiML

compiler. The anonymous function is not allowed to access arbitrary indices of data structures or mutate global state. This restriction is not overly constraining for the majority of use cases and allows the function to be implemented efficiently as a map-reduce. In addition, the anonymous function is often non-trivial to evaluate, and therefore exposes coarse-grained parallelism which can be exploited to achieve strong scaling.

```
40  val sigma = sum(0,m) { i =>
41    val a = if (!x.labels(i)) x(i)-mu0 else x(i)-mu1
42    a.t ** a
43  }
```

Figure 158: The summation representing the bulk of computation in Gaussian Discriminant Analysis

Along with the ability to identify the parallelism inherent in an application, domain abstractions can also abstract away implementation details sufficiently to generate parallel code optimized for various hardware devices. The lack of implementation artifacts in the application source ultimately allows DSL programs to be portable across multiple current and future architectures.

21.2.3 Building DSLs

DSLs have the potential to be a solution for heterogeneous parallelism, but this solution rests on the challenging task of building new DSLs targeting parallelism. The first obvious challenge is designing and constructing a new language, namely implementing a full compiler (i.e., a lexer, parser, type checker, analyzer, optimizer, and code generator). In addition, the DSL must have the facilities to recognize parallelism in applications, and then to generate parallel code that is optimized for different hardware devices (e.g., both the CPU and GPU). This requires the DSL developer to be not only a domain expert, but also an expert in parallelism (to understand and implement parallel patterns) as well as architecture (to optimize for low-level hardware-specific details). Finally, the DSL developer must write a significant amount of plumbing whose implementation can have a significant impact on application performance and scalability. This includes choosing where and how to execute the parallel operations on a given hardware platform, managing data transfers across address spaces, and synchronizing reads and writes to shared data.

21.3 DSL COMPILERS VS. DSL LIBRARIES

As a simpler alternative to constructing a framework for building DSL compilers that target heterogeneous hardware, one could also create a framework for domain-specific libraries. In previous work we presented such a framework along with an earlier version of the OptiML DSL. This DSL could also target heterogeneous processing elements transparently from a single application source with no explicit parallelism and achieve performance competitive with MATLAB. These original versions of Delite and OptiML were implemented as pure libraries in Scala (with the OptiML library extending the Delite library).

21.4 Delite

To address the challenge of building DSLs for parallelism, Delite Compiler Framework and Runtime was presented as a means of dividing the required expertise across multiple systems developers. Delite uses DSL embedding and an extensible compilation framework to greatly reduce the effort in creating a DSL compiler, provides parallel patterns that the DSL developer can extend, performs heterogeneous code generation, and handles all the run-time details of executing the program correctly and efficiently on various hardware platforms. In short, Delite provides the expertise in parallelism and hardware. The DSL developer can then focus on being a domain expert, designing the language constructs and identifying the mapping between those domain constructs and the parallel patterns Delite provides. He or she must implement the data and control structures that inherit from Delite prototypes as well as add domain-specific optimization rules.

The Delite Compiler Framework aims to greatly decrease the burden of developing a compiler for an implicitly parallel DSL, by providing facilities for lifting embedded DSL programs to an intermediate representation (IR), exposing and expressing parallelism, performing generic, parallel, and domain-specific analyses and optimizations, and generating heterogeneous parallel code that will be executed and managed by the Delite Runtime.

21.4.1 Static optimizations and code generation

By introducing compilation Delite DSLs gain several key benefits that are crucial to achieving high performance for certain applications. First of all, we add the ability to perform static optimizations, which includes generic optimizations provided by the Delite framework as well as domain-specific ones provided by the DSL. With a library-based approach optimizations can only be performed dynamically.

In addition, adding code generation support can greatly improve the efficiency of the final executables by eliminating all the DSL abstractions and layers of indirection within the generated code, leaving only type-specialized, straight-line blocks of instructions and first-order control flow that target compilers can optimize heavily. Code generating from an IR also makes targeting hardware other than that supported by the DSL's hosting language much more tractable. A common solution for libraries is to rely on the host language's compiler to perform code generation for the CPU and manually provide native binaries targeting other hardware using the host language's foreign function interface. In our previous work we attempted to somewhat ease this burden on the DSL author for GPUs by writing a compiler plug-in that generated Cuda equivalents of Scala anonymous functions that had disjoint data accesses (i.e., maps). By building an IR, however, Delite is able to handle Cuda code generation seamlessly for both DSL and user-supplied functions, as well as perform static optimizations on the generated kernels that are only reasonable on GPU architectures. These code generators are also easily extensible to new target languages and architectures, making the execution target(s) of Delite DSLs truly independent of the DSL hosting language.

21.4.2 Runtime optimizations

It is also important to note that many of Delite's runtime features are contingent on full program static analyses, which are made possible by the compiler statically generating the execution graph of the application. Delite can make scheduling decisions and specialize the execution at walk-time, thereby incurring significantly less run-time overhead. Full program analysis is also essential for Delite's ability to manage GPU memory intelligently, as discussed in Section IV-C. A librarybased

system can also obtain an execution graph of the application by dynamically deferring the execution of each operation and building up the graph at run-time. We employed such a deferral strategy in our previous work, but were unable to defer past control flow, thereby creating "windows" of the application that could be executed at a time. These windows, however, were not sufficient to allow us to intelligently free GPU memory. We instead treated the GPU main memory as a software-managed cache of the CPU main memory, which was subject to undesirable evictions and could not always handle application datasets that severely pressured the GPU memory's capacity.

21.4.3 Compilation framework

The Delite Compiler Framework uses and extends a generalpurpose compiler framework designed for embedding DSLs in Scala called Lightweight Modular Staging (LMS) [42]. LMS employs a form of meta-programming to construct a symbolic representation of a DSL program as it is executed. For DSLs built on top of LMS, the application code is actually a program generator and each program expression, such as if (c) a else b, constructs an IR node when the program is run (in this case `IfThenElse(c,a,b)`). We use abstract types and type inference to safely hide the IR construction from the DSL user [43].

Through this mechanism the DSL compiler effectively reuses the front-end of the Scala compiler, and then takes over with the creation of the IR. Possible nodes in the IR are all constructs of the DSL or constructs the DSL developer chooses to inherit from Scala (e.g., If-Then-Else statements). The LMS framework provides all of the tools required for building the IR, performing analyses and optimizations, and generating code, which the DSL developer can then use and extend. Delite expands on this functionality by providing three primary views of the IR, namely the generic view, the parallel view, and the domain-specific view, as illustrated in Figure 159.

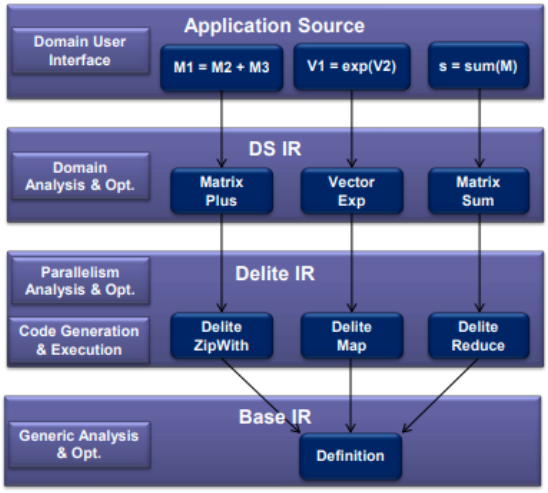


Figure 159: Views of the DSL IR. DSL applications produce an IR upon execution. This IR is defined by the LMS framework with enough information to perform generic analyses and optimizations. The Delite Compiler Framework extends the IR to add parallelism information, and this view allows parallel optimizations and parallel code generation. The DSL extends the parallel IR to form a domain-specific IR, which allows for domain-specific optimizations.

21.4.4 Gerneric IR

The lowest-level view of the IR is centered around symbols and definitions. Unlike many compilers, where individual statements are fixed to basic blocks, which are connected in a control flow graph (CFG), we use a "sea of nodes" representation [32]. Nodes are only connected by their (input and control) dependencies but otherwise allowed to float freely. Nodes in the IR are represented as instances of Scala classes; dependencies are represented as fields in each class. This representation enables certain optimizations to be performed during IR construction. For example, when a side-effect free IR node is constructed, the framework first checks if a definition for the node already exists. If a definition does exist it is reused to perform global common subexpression elimination (CSE). Pattern matching optimizations are also applied during node construction. The DSL compiler can override the construction of an IR node to look for a sequence of operations and rewrite the entire sequence to a different IR node. This mechanism is easy to apply and can be used to implement optimizations such as constant folding and algebraic rewrites. 160 shows an example of implementing a simple pattern matching optimization in OptiML.

```
30    override def matrix_plus[A:Manifest:Arith]
31      (x: Exp[Matrix[A]], y: Exp[Matrix[A]]) =
32        (x, y) match {
33          // (AB + AD) == A(B + D)
34          case (Def(MatrixTimes(a, b)),
35                Def(MatrixTimes(c, d))) if (a == c) =>
36                  matrix_times(a, matrix_plus(b,d))
37          // ...
38          case _ => super.matrix_plus(x, y)
39        }
```

Figure 160: Implementing pattern matching optimizations

Once the complete IR is built and all dependency information is available, transformations that require a global view of the program can take place and work towards a program schedule. Transformations that occur during scheduling include dead-code elimination, various code motion techniques (e.g., loop hoisting) and aggressive fusing of operations, in particular loops and traversals. During the course of these global transformations, the sea of nodes graph is traversed and the result is an optimized program in block structure. An important point is that since the IR is composed of domain operations, all of the optimizations described here are performed at a coarser granularity (e.g., Matrix-Multiply) than in a typical compiler.

It is important to note that in a general-purpose environment, it can be difficult to guarantee the safety of many important optimizations. However, because DSLs naturally use a restricted programming model and domain knowledge is encoded in the operations, a DSL compiler can do a much better job at optimizing than a general-purpose compiler that has to erron the side of completeness. These restrictions are especially important for tackling side effects in DSL programs in order to generate correct parallel code.

In the absence of side effects, the only dependencies among nodes in the IR are input dependencies, which are readily encoded by references from each node instance to its input nodes. While Delite and OptiML favor a functional, side-effect free programming style, prohibiting any kind of side effect would be overly restrictive and not in line with the driving goal of offering pragmatic solutions. However, introducing side effects adds control-, output-, and anti-dependencies that must be detected by the compiler to determine which optimizations can be safely performed. Dependency analysis is significantly complicated if mutable data can be aliased, i.e., a write to one variable may effect the contents of another variable. The key to fine-grained dependency information is to prove that two variables must never alias, which, in general, is hard to do. If separation cannot be ensured, a dependency must be reported. Tracking side effects in an overly conservative manner falsely eliminates both task-level parallelism and other optimization opportunities.

The approach adopted by Delite is to restrict side-effects to a more manageable level. Delite caters to a programming model where the majority of operations is side-effect free and objects start out as immutable. At any point in the program, however, a mutable copy of an immutable object can be obtained. Mutable objects can be modified in-place using side-effecting operations and turned back into immutable objects, again by creating a copy. A future version of Delite might even remove the actual data copies under the hood, based on the results of liveness analysis. The important aspect is that aliasing (and deep sharing) between mutable objects is prohibited.

DSL developers explicitly designate effectful operations and specify which of the inputs are mutated and/or whether the operation has a global effect (e.g., println). In addition, developers can specify for each kind of IR node which of its inputs are only read and which may be aliased

by the object the operation returns (the conservative default being that any input may be read or aliased). This information is used by the dependency analysis to serialize reads of anything that may alias one or more mutable objects with the writes to those objects. The target of a write, however, is always known unambiguously and no aliasing is allowed.

21.4.5 Parallel IR

The Delite Compiler extends the generic IR to express parallelism within and among IR nodes. Task parallelism is discovered by tracking dependencies among nodes. This information is used by the Delite Runtime to schedule and execute the program correctly and efficiently.

IR definition nodes are extended to be a particular kind of Delite op. There are multiple op archetypes, each of which expresses a particular parallelism pattern. A Sequential op, for example, has no internal parallelism, while a Reduce op specifies the reduction of some collection via an associative operator, and can therefore be executed in parallel (as a treereduce). Delite ops currently expose multiple common dataparallel patterns with differing degrees of restrictiveness. Some require entirely disjoint accesses (e.g., Map and Zip), while others allow the DSL to specify the desired synchronization across shared state for each iteration (e.g., Foreach).

Most Delite data-parallel ops extend a common loop-based ancestor, the MultiLoop op. A MultiLoop iterates over a range and applies one or more functions to each index in the range. MultiLoop also has an optional final reduction stage of thread-local results to allow Reduce-based patterns to be expressed. Like Map and Zip, MultiLoop functions must have disjoint access. However, a MultiLoop may consume any number of inputs and produce any number of outputs and is the key abstraction that enables Delite to fuse dataparallel operations together. Delite will fuse together adjacent or producer-consumer MultiLoops that iterate over the same range and do not have cyclic dependencies, creating a single pipelined MultiLoop. By fusing a MultiLoop that produces a set of elements together with a MultiLoop that consumes the same set, potentially large intermediate data structures can be entirely eliminated. Since fusing ops can create new opportunities for further optimization, fusion is iterated (and previously discussed optimizations reapplied) until a fixed point is reached. In addition to allowing multiple data-parallel ops in a single loop, fusion also effectively creates optimized MapReduce and ZipReduce ops (as well as any other combination, e.g., MapReduceReduce). Since Delite ops internally extend MultiLoop, DSL authors can benefit from fusion even while using only the simpler data parallel patterns.

Fusion can significantly improve the performance of applications by improving cache behavior and reducing the total number of memory accesses required. For example consider the OptiML code shown in 157. The application performs multiple subsequent operations on the input in order to update the result. Fusing these operations into a single traversal over the input collection that generates all of the outputs at once without temporary buffer allocations can produce a significant performance improvement for large inputs.

21.4.6 Domain-specific IR

The DSL developer extends the Delite Compiler to create domain-specific IR nodes that extend the appropriate Delite op. It is through this simple mechanism that a DSL developer expresses how to map domain constructs onto existing parallel patterns. This highest-level view of the IR is unique for each DSL and allows for domain-specific analyses and optimizations. For example, OptiML views certain IR nodes as linear algebra operations, which allows it to use pattern matching to apply linear algebra simplification rules. These rewrites can eliminate redundant work (e.g.,

whenever Transpose(Transpose(x)) is encountered, it is rewritten to be simply x) as well as yield significantly more efficient implementations that are functionally equivalent to the original. As an example, consider the snippet of OptiML code for Gaussian Discriminant Analysis (GDA) shown in 158. The OptiML compiler's pattern matcher recognizes that a summation of outer products can be implemented much more efficiently as a single matrix multiplication [41]. Specifically, it recognizes

$$\sum_{i=0}^{n} \vec{x}_i * \vec{y}_i \rightarrow \sum_{i=0}^{n} X(:,i) * Y(i,:) = X * Y$$

The transformed code allocates two matrices, populates them by performing the operations required to produce all of the inputs to the original outer product operation, and then performs the multiplication.

21.4.7 Heterogeneous code generation

The final stage of compilation is code generation. The DSL can extend one or more code generators, which are modular objects that translate IR nodes to an implementation in a lower level language. The LMS framework provides the basic mechanisms for traversing the IR and invoking the code generation method on each node. It also provides generator implementations for host language operations. On top of that, the Delite Compiler Framework supplies generator implementations for all Delite ops. Due to the ops' deterministic access patterns and restricted semantics, Delite is able to generate safe parallel code for CMPs and GPUs without performing complex dependency analyses. The DSL developer can also choose to override the code generation for an individual target (e.g., Cuda [44]) to provide a hand-optimized implementation or utilize an existing library (e.g. CUBLAS, CUFFT). We currently have implemented code generators for Scala, C++, and Cuda, which allow us to leverage their existing compilers to perform further low-level optimizations.

The Delite Compiler Framework adds a new code generator which generates a representation of the application as an execution graph of Delite ops with executable kernels. The design supports control flow nodes and nested graphs, exposing parallelism within a given loop or branch. For every Delite op, the Delite generator emits an entry in the graph containing the op's dependencies. It then invokes the other available generators (Scala, Cuda, etc.) for each op, generating multiple devicespecific implementations of each op kernel. For example, if a particular operation may be well-suited to GPU execution, the framework will emit both a CPU-executable variant of the op as well as a GPU-executable variant of the op. The runtime is then able to select which variant to actually execute. Since it is not always possible to emit a given kernel for all targets, each op in the graph is only required to have at least one kernel variant. By emitting this machine-agnostic execution graph of the application along with multiple kernel variants, we are able to defer hardware specific decisions to the runtime and therefore run the application efficiently on a variety of different machines. This mechanism also allows the DSL to transparently expand its set of supported architectures as new hardware becomes available. Once Delite supports code generation and runtime facilities for the new hardware, existing DSL application code can automatically leverage this support by simply recompiling.

21.5 HETEROGENEOUS RUNTIME

The Delite Runtime provides services required by DSLs to execute implicitly parallel programs, such as scheduling, data management, and synchronization, and optimizes execution for the particular machine.

21.5.1 Scheduling

The runtime takes as input the execution graph generated by the Delite Compiler, along with the kernels and any additional necessary code generated by the Delite Compiler, such as DSL data structures. The execution graph is a machine-agnostic description of the inherent parallelism within the application that enumerates all the ops in the program along with their static dependencies and supported target(s). The runtime schedules the application at walk-time [45], combining the static knowledge of the application behavior provided by the execution graph with a description of the current machine, i.e., the number of CPU cores, number of GPUs, etc. (see Figure 161). The scheduler traverses all of the nested graphs in the execution graph file and produces partial schedules for blocks of the application that are statically determinable. The partial schedules are dispatched dynamically during execution as the branch directions are resolved. The runtime scheduler currently utilizes a clustering algorithm that prefers scheduling each op on the same resource as one of its inputs. If an op has no dependencies it is scheduled on the next available resource. This algorithm attempts to minimize communication among ops and makes device decisions based on kernel and hardware availability. Data-parallel ops selected for CMP execution are split into a number of chunks (determined by resource availability) and then scheduled across multiple CPU resources.

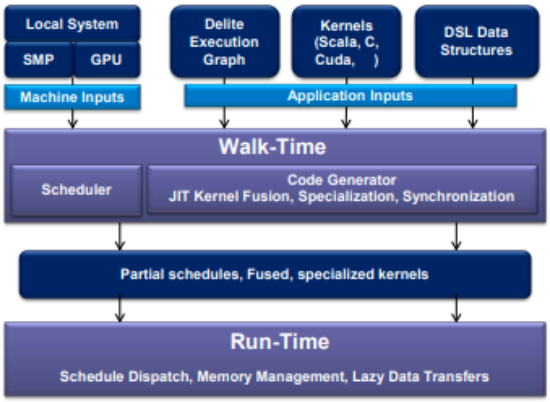


Figure 161: An overview of the Delite Runtime. The runtime uses the machine-agnostic execution graph representing the application as well as a machine description to schedule and execute the application on the available hardware. Walk-time code generation utilizes scheduling information to optimize kernels and synchronization, minimizing run-time overheads. Run-time systems execute the schedule, manage memory, and perform data transfers.

21.5.2 Schedule compilation

In order to avoid the overheads associated with dynamically interpreting the execution graph, the runtime generates an executable for each hardware resource that invokes the kernels assigned to that resource according to the partial schedules. Since the compiler is machine-agnostic, the runtime is responsible for generating an implementation of each dataparallel op that is specialized to the number of processors chosen by the schedule. For example, a Reduce op only has its reduction function generated by the compiler, and the runtime generates a tree-reduction implementation with the tree height specialized to the number of processors chosen to perform the reduction.

The generated code enforces the schedule by synchronizing kernel inputs and outputs across resources. The synchronization is implemented by transferring data through lock-based one-place buffers. This code generation allows for a distributed program at runtime (no master coordination thread is required) and also allows for multiple optimizations that minimize runtime overhead. For example, kernels scheduled on the same hardware resource with no communication between them are fused to execute back-to-back. All synchronization in the application is generated at this time and only when necessary (kernel outputs that do not escape a single hardware resource require no synchronization). So in the simplest case of targeting a traditional uniprocessor, the final executable code will not invoke any synchronization primitives (e.g., locks). The runtime also injects data transfers when the communicating resources reside in separate address spaces. When shared memory is available, it simply passes the necessary pointers.

21.5.3 Execution

The current implementation of the Delite Runtime is written in Scala and generates Scala code for each CPU thread and Cuda code to host each GPU. This environment allows it to support the execution of Scala kernels, C++ kernels, and Cuda kernels that are generated by the Delite compiler (using JNI as a bridge). The runtime spawns a JVM thread for each CPU resource assigned to a kernel, and also spawns a single CPU host thread per Cuda-compliant GPU.

The GPU host thread performs the work of launching kernels on the GPU device and transferring data between main memory and the device memory. For efficiency, it allows the address spaces to become out-of-sync by default, and only performs data transfers when the schedule requires them. Delite also provides memory management for the GPU. Before each Cuda kernel is launched, any memory on the device it will require is allocated and registered. The runtime uses the execution graph and schedule to perform liveness analysis for each input and output of GPU ops to determine the earliest time during execution at which it can be freed. By default, the runtime attempts to keep the host thread running ahead as much as possible by performing asynchronous memory transfers and kernel launches. When this causes memory pressure, however, the runtime uses the results of the liveness analysis to wait for enough data to become dead, free it, and then perform the new allocations. This analysis can be very useful due to the limited memory available in current GPU devices.

22 Memory Hierarchy Optimizations

Software-controlled data prefetching is a promising technique for improving the performance of the memory subsystem to match today's high-performance processors. While prefctching is useful in hiding the latency, issuing prefetches incurs an instruction overhead and can increase the load on the memory subsystem. As a resu 1 care must be taken to ensure that such overheads do not exceed the benefits.

22.1 Introduction

Various hardware and software approaches to improve the memory performance have been proposed recently [15]. A promising technique to mitigate the impact of long cache miss penalties is softwarecontrolled prefetching[5, 13, 16, 22 23]. Software-controlled prefetching[46] requires support from both hardware and software. The processor must provide a special "prefetch" instruction. The software uses this instruction to inform the hardware of its intent to use a particular data item, if the data is not cumently in the cache, the data is fetched in from memory. The cache must be lockup-free[17]; that is, the cache must allow multiple outstrmding misses. While the memory services the data miss, the program can continue to execute as long as it does not need the requested data. While prefetching does not reduce the latency of the memory access, it hides the memory latency by overlapping the access with computation and other accesses. Prefetches on a scalar machine are analogous to vector memory accesses on a vector machine. In both cases, memory accesses are overlapped with computation and other accesses. Furthermore, similar to vector registers, prefetching allows caches in scalar machines to be managed by software. A major difference is that while vedor machines can only operate on vectors in a pipelined manner, scalar machines can execute arbitrary sets of scalar operations well.

Another useful memory hierarchy optimization is to improve data locality by reordering the execution of iterations. One important example of such a transform is blocking[l, 9, 10, lZ 21, 23, 29]. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks, so that datu loaded into the faster levels of the memory hkmarchy are reused. Other useful transformations include unimodular loop transforms such as interchange, skewing and reversal[29]. Since these optimization improve the @de's data locality, they not only reduee the effeetive memory access time but also reduce the memory bandwidth requirement. Memory hierarchy optimization such as prefetching and blocking are crucial to turn high-performance microprocessors into effective scientific engines.

22.2 Blocking[47]

Blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks, so that data loaded into the faster levels of the memory hierarchy are reused. This paper presents cache performance data for blocked programs and evaluates several optimization to improve this performance. The data is obtained by a theoretical model of data conflicts in the cache, which has been validated by large amounts of simulation.

Due to high level integration and superscalar architectural designs, the floating-point arithmetic capability of microprocessors has increased significantly in the last few years. Unfortunately, the increase in processor speed has not been accompanied by a similar increase in memory speed. To fully realize the potential of the processors, the memory hierarchy must be efficiently utilized.

While data caches have been demonstrated to be effective for general-purpose applications in bridging the processor and memory speeds, their effectiveness for numerical code has not been established. A distinct chmactenstic of numerical applications is that they tend to operate on large data sets. A cache may only be able to hold a small fraction of a matrix; thus even if the data are reused, they may have been displaced from the cache by the time they are reused.

Consider the example of matrix multiplication for matrices of size NxN:

```
for i := 1 to N do
   for k := 1 to N do
      r = X[i,k]; /* register allocated */
      for j := 1 to N do
         Z[i,j] += r*Y[k,j];
```

Figure 162: matrix multiplication example

Figure 163(a) shows the data access pattern of this code. The same element $X[i,k]$ is used by all iterations of the innermost loop; it can be register allocated and is fetched from memory only once. Assuming that the matrix is organized in row major order, the innermost loop of this code accesses consecutive data in the Y and Z matrices, and thus utilizes the cache prefetch mechanism fully. The same row of Z accessed in an innermost loop is reused in the next iteration of the middle loop, and the same row of Y is reused in the otftermost loop. Whether the data remains in the cache at the time of reuse depends on the size of the cache. Unless the cache is large enough to hold at least one $N * N$ matrix, the data Y would have been displaced before reuse. If the cache cannot hold even one row of the data then Z data in the cache cannot be reused. In the worst case, $2N^3 + N^2$ words of data need to be read from memory in N^3 iterations. The high ratio of memory fetches to numerical operations can significantly slow down the machine.

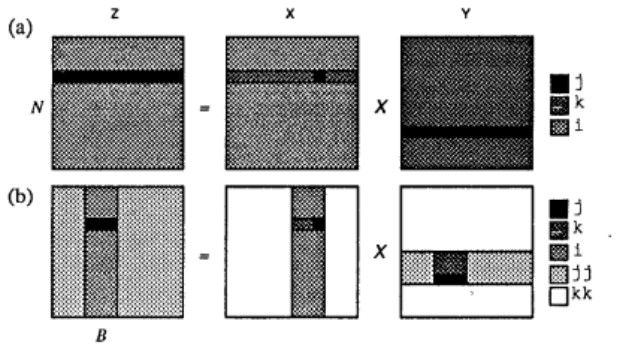


Figure 163: Data access pattern in (a) unblocked and (b) blocked matrix multiplication.

It is well known that the memory hierarchy can be better utilized if scientific algorithms are blocked. Blocking is also known as tiling. Instead of operating on individual matrix entries, the calculation is performed on submatrices.

Blocking can be applied to any and multiple levels of memory hierarchy, including virtual memory, caches, vector registers, and scalar registers. The matrix multiplication code blocked to reduce cache misses looks like this:

```
for kk := 1 to N by B do
  for jj := 1 to N by B do
    for i := 1 to N do
      for k := kk to min(kk+B-1, N) do
        r = X[i,k]; /* register allocated */
        for j := jj to min(jj+B-1, N) do
          Z[i,j] += r*Y[k,j];
```

Figure 164: Reduced matrix multiplication example.

Figure 163(b) shows the data access pattern of the blocked code. We observe that the original data access pattern is reproduced here, but at a smaller scale. The blocking factor, B , is chosen so that the $B * B$ submatrix of Y and a row of length B of Z can fit in the cache. In this way, both Y and Z are reused B times each time the data rue brought in. Thus, the total memory words accessed is $2N^3/B + N2$ if there is no interference in the cache.

Blocking is a general optimization technique for increasing the effectiveness of a memory hierarchy. By reusing data in the faster level of the hierarchy, it cuts down the average access latency. It also reduces the number of references made to slower levels of the hierarchy. Blocking is thus superior to optimization such us prefetching, which hides the latency but does not reduce the memory bandwidth requirement. This reduction is especially important for multiprocessors since memory bandwidth is often the bottleneck of the system.

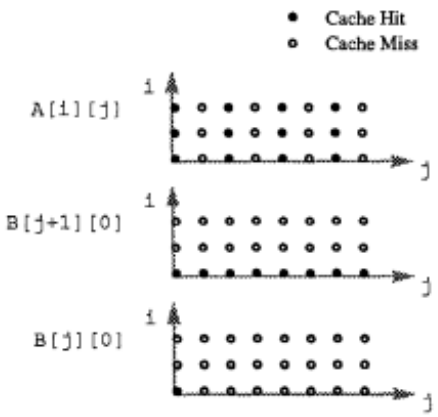
22.3 Prefetch[46]

In this section, we will use the code in Figure 165(a) as a running example to illustrate our prefetch algorithm. We assume, for this example, that the cache is 8K bytes, the prefetch latency is 100 cycles and the cache line size is 4 words (two double-word array elements to each cache line), In this case, the set of references that will cause cache misses can be determined by inspection (Figure 165(b)).

(a)

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

(b)



(c)

Reference	Locality	Prefetch Predicate
A[i][j]	$\begin{matrix} i \\ j \end{matrix}$ = none spatial	(j mod 2) = 0
B[j+1][0]	$\begin{matrix} i \\ j \end{matrix}$ = temporal none	i = 0

(d)

```
prefetch(&A[0][0]);
for (j = 0; j<6; j += 2) {
  prefetch(&B[j+1][0]);
  prefetch(&B[j+2][0]);
  prefetch(&A[0][j+1]);
}
for (j = 0; j<94; j += 2) {
  prefetch(&B[j+7][0]);
  prefetch(&B[j+8][0]);
  prefetch(&A[0][j+7]);
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (j = 94; j<100; j += 2) {
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (i = 1; i<3; i++) {
  prefetch(&A[i][0]);
  for (j = 0; j<6; j += 2)
    prefetch(&A[i][j+1]);
  for (j = 0; j<94; j += 2) {
    prefetch(&A[i][j+7]);
    A[i][j] = B[j][0]+B[j+1][0];
    A[i][j+1] = B[j+1][0]+B[j+2][0];
  }
  for (j = 94; j<100; j += 2) {
    A[i][j] = B[j][0]+B[j+1][0];
    A[i][j+1] = B[j+1][0]+B[j+2][0];
  }
}
```


Figure 165: Example of selective prefetching algorithm.

In Figure 165(d), we show code that issues all the useful prefetches early enough to overlap the memory accesses with computation on other data. (This is a source-level representation of the actual co generated by our compiler for this case). The first three loops correspond to the computation of the i=O iteration, and the remaining code executes the remaining iterations. This loop splitting step is necessary because the prefetch pattern is different for the different iterations. Furthermore, it takes three loops to implement the innermost loop. The first loop is the prolog, which prefetches data for the initial set of iterations; the second loop is the steady state where each iteration executes the code for the iteration and prefetches for future iterations; the third loop is the epilog that executes the last iterations. This so are pipelining transformation is necessary to issue the prefetches enough iterations shead of their use[18, 24].

This example illustrates the three major steps in the prefetch algorithm

- 1. For each reference, determine the accessesthat are likely to be cache misses and therefore need to be prefetched.
- 2. Isolate the predicted cache miss instances through loop splitting. This avoids the overhead of adding conditional statements to the loop bodies.
- 3. Software pipeliie prefetches for all cache misses.

In the following, we describe each step of the algorithm and show how the algorithm develops the prefetch code for the above example systematically.

22.3.1 Locality Analysis

The tirst step determines those references that are likely to cause a cache miss, This locality analysis is broken down into two substeps. The tirst is to discover the intrinsic data reuses within a loop nest; the second is to determine the set of reuses that cart be exploited by a cache of a particular size.

Reuse Analysis Reuse analysis attempts to discover those instances of array accesses that refer to the same cache line. There are three kinds of reuse: temporal, spatial and group. In the above example, we say that the reference A [i][j] has spatial reuse within the innermost loop since the same cache line is used by two consecutive iterations in the innermost loop. The reference B [j][0] has temporal reuse in the outer loop since iterations of the outer loop refer to the same locations. Lastly, we say that different accesses B [j][0] and B [j+ 1][0] have group reuse because many of the instances of the former refer to locations accessed by the latter.

Trying to determine accurately all the iterations that use the same data is too expensive. We can succinctly capture our intuitive characterization that reuse is carried by a specific loop with the following mathematical formulation. We represent an n-dimensional loop nest as a polytope in an n-dimensional iteration space, with the outermost loop represented by the first dimension in the space. We represent the' shape of the set of iterations that use the same data by a reuse vector space[291.

For example, the access of b[j][0] in our example is represented as $B\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}\right)$, so reuse occurs between iterations (i_1, j_1) and (i_2, j_2) whenever

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix}, \text{ or}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

That is, temporal reuse occurs whenever the difference between the two iterations lies in the nullspace of $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, that is, span $(1, O)$. We refer to this vector space as the temporal reuse vector space. This mathematical approach succinctly captures the intuitive concept that the direction of reuse of B [j][0] lies along the outer loop. This approach can handle more complicated access patterns such as C[i+j] by representing their reuse vector space as Span$(1, -1)$.

Similar analysis can be used to find spatial reuse. For reuse among different array references, Gannon et al. observe that data reuse is exploitable only if the references are uniformly generated, that is, references whose array index expressions differ in at most the constant term[11]. For examplq references B [j][0] and B[j+1][0] are uniformly generatd, references C [i] and C [j] are not. Pairs of uniformly generated references can be analyzed in a similar fashion[29]. For our example in Figure 165(a), our algorithm will determine that A [i] [j] has spatird reuse on the inner loop, and both B[j] [0] and B[j+l] [0] share group reuse wtd also have temporal reuse on the outer loop.

Localized Iteration Space Reuses translate to locality only if the subsequent use of data occurs before the data are displaced from the cache. Factors that determine if reuse translates to locality include the loop iteration count (since that determines how much data are brought in between reuses), the cache size, its set associativity and replacement policy.

We begin by considering the first two factors: the loop iteration count and the cache size. In the example above, reuse of B [j][0] lies along the outer dimension. If the iteration count of the innermost loop is large relative to the cache size (e.g., if the upper bound of the j loop in Figure 165(a) was 10,000 rather than 100), the data may be flushed from the cache before they are used in the next outer iteration. It is impossible to determine accurately whether data will remain in the cache due to factors such as symbolic loop iteration counts and the other cache characteristics. Instead of trying to represent exactly which reuses would result in a cache hi we capture only the dimensionality of the iteration space that has data locality ['291.We define the localized iteration space to be the set of loops that can exploit reuse. For example, if the localized iteration space consists of only the innermost loop, that means data fetched will be available to iterations within the same innermost loop, but not to iterations from the outer loops.

The localized iteration space is simply the set of innermost loops whose volume of data accessed in a single iteration does not exceed the cache size. We estimate the amount of data used for each level of loop nesting, using the reuse vector information. Our algorithm is a simplified version of those proposed previously[8, 11, 23]. We assume loop iteration counts that cannot be determined at compile time to be small-this tends to minimize the number of prefetches. A reuse can be exploited only if it lies within the localized iteration space. By representing the localized iteration space also as a vector space, locality exists only if the reuse vector space is a subspace of the localized vector space

Consider our example in Figure 165(a). In this case, the loop bound is known so our algorithm can easily determine that the volume of data used in each loop fits in the cache. Both loops are within the localized iteration space, and the localized vector space is represented as span $(1, 0), (0, 1)$.

Since the reuse vector space is necessarily a subspace of the localizad vector space, the reuses will correspond to cache hits, and it is not necessary to prefetch the reuses.

Similar mathematical treatment determines whether spatial reuse translates into spatial locality. For group reuses, our algorithm determines the sets among the group that can exploit locality using a similar technique. Furthermore, it determines for each set its leading reference, the reference that accesses new data first and is thus likely to incur cache misses. For example, of B [j] [0] and B [j+1] [0],B [j+1] [0] is the first reference that accesses new data. The algorithm need only issue prefetches for B [j+1] [0] and not B [j] [0].

In the dkcussion so far, we have ignored the effects of cache conflicts. For scientific programs, one important source of cache conflicts is due to accessing data in the same matrix with a constant stride. Such conflicts can be predicted, and can even be avoided by embedding the matrix in a larger matrix with dimensions that are less problemztic[19]. We have not implemented this optimization in our compiler. Since such interference can greatly disturb our simulation results, we manually changed the size of some of the matrices in the benchmarks (details are given in Section 3.) Conflicts due to interference between two different matrices are more difficult to analyze. We cumently approximate this effect simply by setting the "effective" cache size to be a fraction of the actual cache size.

The Prefetch Predicate The benefit of locality differs according to the type of reuse. If an access has temporal locality within a loop nest, only the first access will possibly incur a cache miss. If an access has spatial locality, only the ftrst access to the same cache line will incur a miss.

To simplify this exposition, we assume here that the iteration count starts at 0, and that the data arrays are aligned to start on a cache line boundary. Without any locality, the default is to prefetch all the time. However, the presence of temporal locality in a loop with index$i$means that prefetching is necessary only when $i = 0$. The presence of spatial locality in a loop with index i means that prefetching is necessary only when $(i \bmod l) = 0$, where l is the number of array elements in each cache line, Each of these predicates reduces the instances of iterations when data need to be prefetched. We define the prefetch predicate for a reference to be the predicate that determines if a particular iteration needs to be prefetched. The prefetch predieate of a loop nest with multiple levels of locality is simply the conjunction of all the predicates imposed by each form of locality within the loop nest.

Figure 165(c) summarizes the outcome of the first step of our prefetch algorithm when applied to our example. Because of the small loop iteration counnt, all the reuse in this case results in locality. The spatial and temporal locality each translate to different prefetch predicates. Finally, since B [j] [0] and B [j+1] [0] share group reuse, prefetches need to be generated only for the leading reference B[j+1] [0].

Loop Splitting Ideally, only iterations satisfying the prefetch predicate should issue prefetch instructions. A naive way to implement this is to enclose the prefetch instructions inside an IF statement with the prefetch predicate as the condition. However, such a statement in the innermost loop can be costly, and thus defeat the purpose of reducing the prefetch overhead. We can eliminare this overhead by decomposing the loops into different sections so that the predicates for all instances for the same section evaluate to the same value. This process is known as loop splitting. In general, the predicate $i = O$ requires the first iteration of the loop to be peeled. The predicate $(i \bmod l) = 0$ requires the loop to be unrolled by a factor of l. Peeling and unrolling can be applied recursively to handle predicates in nested loops.

Going back to our example in Figure 165(a), the $i = 0$ predicate causes the compiler to peel the$i$loop. The $(j \bmod 2) = 0$ predicate then causes the$j$loop to be unrolled by a factor of 2-both in the peel and the main iterations of the$i$loop.

However, peeling and unrolling multiple levels of loops can potentially expand the code by a signitieant amount. This may reduce the effectiveness of the instruction cachq also, existing optimizing compilers are often ineffective for large procedure bodies, Our algorithm keeps track of how large the loops are growing. We suppress peeling or unrolling when the loop becomes too large. This is made possible because prefetch instructions are only hints, and we need not ksue those and only those satisfying the prefetch predicate, For temporal locality, if the loop is too large to peel, we simply drop the prefetches. For spatial locality, when the loop becomes too large to unroll, we introduce a conditional statement. When the loop body has become this large, the cost of a conditional statement is relatively small.

Scheduling Prefetches Prefetches must be issued early enough to hide memory latency. They must not be issued too early lest the data fetched be flushed out of the cache before they are used. We choose the number of iterations to be the unit of time scheduling in our algorithm. The number of iterations to prefetch ahead is

$$\left\lceil \frac{l}{s} \right\rceil$$

where l is the prefetch latency and s is the length of the shortest path through the loop body.

In our example in Figure 165(a), the latency is 100 cycles, the shortest path through the loop body is 36 instructions long, therefore, the j loops are software-pipelined three iterations ahead. Once the iteration count is determined the code transformation is mechanical.

Since our scheduling quantum is an iteration, this scheme prefetches a data item at least one iteration before it is used. If a single iteration of the loop can fetch so much data that the prefetched data may be replaced, we suppress issuing the prefetch.

23 Parallelism and Dependence Theory

23.1 DATA DEPENDENCE

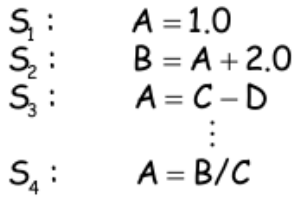


Figure 166: An example.

- Flow (true) dependence: a statement S_i precedes a statement S_j in execution and S_i computes a data value that S_j uses. $S_i \delta^+ S_j$. In Figure 166, $S_1 \delta^+ S_2$ and $S_2 \delta^+ S_4$
- Anti dependence: a statement S_i precedes a statement S_j in execution and S_i uses a data value that S_j computes. $S_i \delta^a S_j$ In Figure 166, $S_2 \delta^a S_3$
- Output dependence: a statement S_i precedes a statement S_j in execution and S_i computes a data value that S_j also computes. $S_i \delta^o S_j$, In Figure 166, $S_1 \delta^o S_3$ and $S_3 \delta^o S_4$

The dependence is said to flow from S_i to S_j because S_i precedes S_j in execution. S_i is said to be the source of the dependence. S_j is said to be the sink of the dependence. The only "true" dependence is flow dependence; it represents the flow of data in the program. The other types of dependence are caused by programming style; they may be eliminated by re-naming as seen in Figure 167.

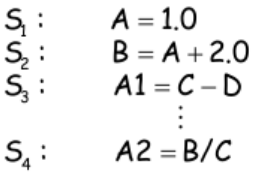


Figure 167: Dependence other than flow dependence can be eliminated by re-naming for Figure 166.

23.2 DATA DEPENDENCE DIRECTIONS

23.2.1 =

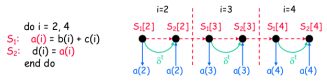


Figure 168

In Figure 168, There is an instance of S_1 that precedes an instance of S_2 in execution and S_1 produces data that S_2 consumes. S_1 is the source of the dependence; S_2 is the sink of the dependence. The dependence flows between instances of statements in the same iteration (loop-independent dependence). The number of iterations between source and sink (dependence distance) is 0. The dependence direction is =. $S_1\delta^{\pm}_{=}S_2$ or $S_1\delta^{+}_{0}S_2$

23.2.2 <

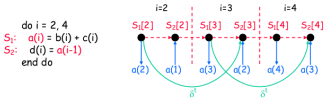


Figure 169

In Figure 169,there is an instance of S_1 that precedes an instance of S_2 in execution and S_1 produces data that S_2 consumes. S_1 is the source of the dependence; S_2 is the sink of the dependence. The dependence flows between instances of statements in different iterations (loop-carried dependence). The dependence distance is 1. The direction is positive (¡). $S_1\delta^{+}_{<}S_2$ or $S_1\delta^{+}_{1}S_2$

23.2.3 >

```
do i = 2, 4
  do j = 2, 4
S:   a(i,j) = a(i-1,j+1)
  end do
end do
```

Figure 170

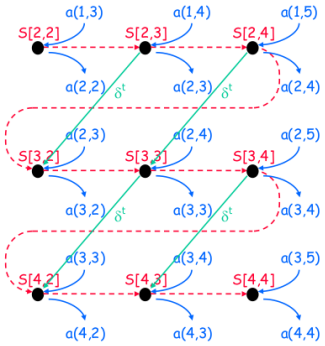


Figure 171

An instance of S precedes another instance of S and S produces data that S consumes. S is both source and sink. The dependence is loop- carried. The dependence distance is (1,-1). $S_1\delta^+_{(1,-1)}S_2$

23.3 LOOP TRANSFORMATIONS USING DIRECTION VECTORS

23.3.1 Loop Parallelization

The iterations of a loop may be executed in parallel with one another if and only if no dependences are carried by the loop.

When that loop does not have a loop-carried dependence, it is safe to run a loop in parallel, which means outermost loop with a direction other than "="

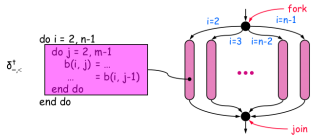


Figure 172: Iterations of loop j must be executed sequentially, but the iterations of loop i may be executed in parallel. Outer loop parallelism.

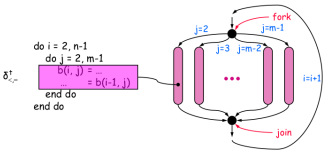


Figure 173: Iterations of loop i must be executed sequentially, but the iterations of loop j may be executed in parallel. Inner loop parallelism.

23.3.2 Loop Interchange

When all dependences remain lexicographically positive, it is legal to interchange or block/tile loops, which means outermost direction other than "=" must be "¡" (positive)

23.4 DATA DEPENDENCE DECISION ALGORITHM

23.4.1 Lamport's Test

Lamport's Test is used when there is a single index variable in the subscript expressions, and when the coefficients of the index variable in both expressions are the same.

$$A(\cdots, b*i+c_1, \cdots) = \cdots\cdots = A(\cdots, b*i+c_2, \cdots)$$

The dependence problem: does there exist i 1 and i 2 , such that $L_i \le i_1 \le i_2 \le U_i$ and such that

$$b*i_1 + c_1 = b*i_2 + c_2$$

or

$$i_2 - i_1 = \frac{c_1 - c_2}{b_2}$$

There is integer solution if and only if $\frac{c_1-c_2}{b_2}$is integer. The dependence distance is $d = \frac{c_1-c_2}{b_2}$ if $L_i \le |d| \le U_i$

- $d > 0$: true dependence.
- $d = 0$: loop independent dependence.
- $d < 0$: anti dependence.

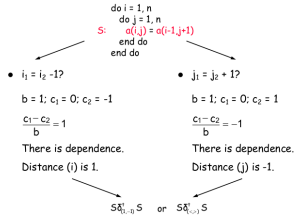


Figure 174

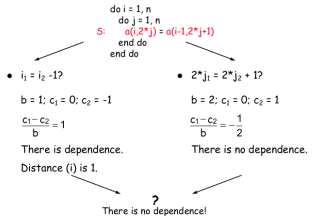


Figure 175

23.4.2 GCD Test

A greatest common divisor (GCD) test is a test used in computer science compiler theory to study of loop optimization and loop dependence analysis to test the dependency between loop statements. [48]

It is difficult to analyze array references in compile time to determine data dependency (whether they point to same address or not). A simple and sufficient test for the absence of a dependence is the greatest common divisor (GCD) test. It is based on the observation that if a loop carried dependency exists between $X[a * i + b]$ and $X[c * i + d]$ (where X is the array; a, b, c and d are integers, and i is the loop variable), then GCD (c, a) must divide (d - b). The assumption is that the loop must be normalized : written so that the loop index/variable starts at 1 and gets incremented by 1 in every iteration. For example, in the following loop, a=2, b=3, c=2, d=0 and GCD(a,c)=2 and (d-b) is -3. Since 2 does not divide -3, no dependence is possible.

```
for (i=1; i<=100; i++)
{
  X[2*i+3] = X[2*i] + 50;
}
```

24 Compiler Optimizations for Thread-Level Speculation

While using this multithreaded hardware to improve the throughput of a workload is straightforward, using it to improve the performance of a single application requires parallelization. The ideal solution would be to convert sequential programs into parallel programs automatically, but unfortunately this is difficult (if not impossible) for many general-purpose programs due to their use of pointers, complex data and control structures, and run-time inputs.

Thread-Level Speculation (TLS) [1, 6, 14, 15, 16, 20, 21, 26, 30, 34] is a potential solution to this problem since it allows the compiler to create parallel threads without having to prove that they are independent. The underlying hardware ensures that interthread dependences through memory are satisfied, and re-executes any thread for which they are not.

The key to extracting parallelism from these programs and hence improving performance is in the efficiency of speculative execution. While recent research has investigated hardware optimization for TLS [6, 20, 22, 31, 24], there has been relatively little work on compiler optimization in this area. One potential opportunity for optimization focuses on data dependences between speculative threads that occur frequently: if the compiler is able to identify the source and the destination of a frequent inter-thread data dependence, then it is beneficial to insert synchronization and forward that value explicitly to avoid failed speculation. Figure 1(a) shows an example loop that the compiler has speculatively parallelized by partitioning the loop into speculative threads (aka epochs). Since the variable A is read and written in every iteration, the compiler decides to synchronize and forward A by inserting a wait operation before the first use of A, and a signal operation after the last definition of A—we describe, implement, and evaluate this algorithm in Section 3. The synchronization results in the partially-parallel execution shown in Figure 1(a), where each epoch stalls until the value of A is produced by the previous epoch. The flow of the value of A between epochs serializes the parallel execution, and so we refer to it as a critical forwarding path. In the next section, we show that the overall performance of speculation is limited by the size of this critical forwarding path.

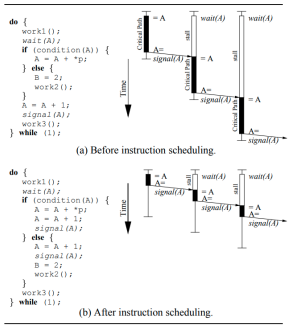


Figure 176: Impact of scheduling on the critical forwarding path.

24.1 What Can the Compiler Do to Improve Performance?[49]

Although synchronization is better than speculation for a data dependence that occurs frequently, the resulting serialization can still limit performance. In fact, the performance of many applications that exploit TLS is limited by the critical forwarding path. Figure 2 shows the poten-

tial impact of reducing the critical forwarding path on a four-processor chip multiprocessor—we will explain the details of this experiment later in Section 2. The U bars show the unscheduled TLS version of each application run speculatively in parallel. Each bar is normalized to the execution time of the original sequential version, such that bars less than 100 are speeding up. The best we can possibly do with scheduling is to eliminate the critical forwarding path altogether. We measure this ideal behavior with a model that can perfectly predict all forwarded values such that there is no synchronization (P). We see that for most applications, removing the synchronization bottleneck results in great performance improvements.

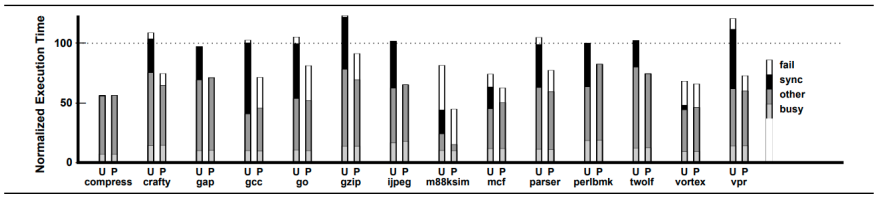


Figure 177: Potential impact of reducing the critical forwarding path. For each benchmark we show execution time on four processors for the speculatively parallelized regions of code normalized to that of the original sequential versions. U is the unscheduled speculative version and P shows the impact of perfect prediction of forwarded values.

What can the compiler do to shrink the critical forwarding path? The key idea is to reduce the number of instructions between each wait/signal pair. However, this becomes more difficult in the presence of conditional control flow. Figure 1(b) shows the example loop after the compiler has scheduled the code to reduce the critical forwarding path. The scheduling algorithm has duplicated the computation of A=A+1 as well as the signal and moved them into the conditional structure. If the condition on A is rarely true, then less work will be performed before reaching each signal (by deferring the computation of B=2 and work2()). As shown in the figure, this reduces the stall time for each epoch, thereby improving overall execution time.

24.1.1 Inserting Wait/Signal

24.1.2 Scheduling Instructions

24.1.3 Scheduling Instructions Speculatively

25 Profile Guided Optimizations

25.1 Efficient Path Profiling

25.2 Improved Basic Block Reordering

Improved Basic Block Reordering [50] is published by Andy Newell and Sergey Pupyrev from Facebook.

Given a directed control flow graph comprising of basic blocks and frequencies of jumps between the blocks, find an ordering of the blocks such that the number of fall-through jumps is maximized. This is the maximum directed TRAVELING SALESMAN PROBLEM (TSP). Solving TSP alone is not sufficient for constructing a good ordering of basic blocks. It is easy to find examples of control flow graphs with multiple different orderings that are all optimal with respect to the TSP objective. Consider for example a control flow graph in Figure 178 in which the maximum number of fall-through branches is achieved with two orderings that utilize a different number of I-cache lines in a typical execution. For these cases, an algorithm needs to take into consideration non-fall-through branches to choose the best ordering. However, maximizing the number of fall-through jumps is not always preferred from the performance point of view. Consider a control flow graph with seven basic blocks in Figure 179. It is not hard to verify that the ordering with the maximum number of fall-through branches is one containing two concatenated chains, B0→B1→B3→B4 and B5→B6→B2 (upper-right in Figure 179). Observe that for this placement, the hot part of the function occupies three 64-byte cache lines. Arguably a better ordering is the lower-right in Figure 179, which uses only two cache lines for the five hot blocks, B0, B1, B2, B3, B4, at the cost of breaking the lightly weighted branch B6→B2.

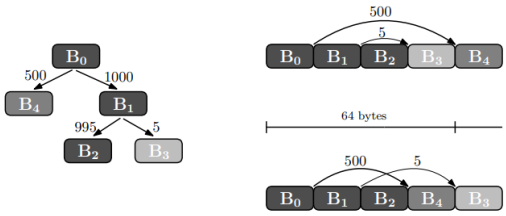


Figure 178: Two orderings of basic blocks with the same TSP score (1995) resulting in different I-cache utilization. All blocks have the same size of 16 bytes and colored according to their hotness in the profile.

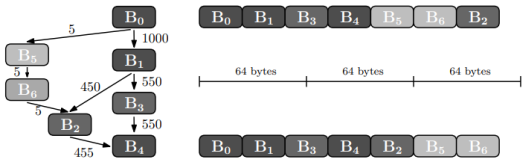


Figure 179: A control flow graph with jump frequencies (left) and two possible orderings of basic blocks (right). All blocks have the same size (in bytes) and colored according to their hotness in the profile. An optimal TSPbased layout (upper right) utilizes three cache lines for the hot code, while an arguably better layout (lower right) can be built with a new EXTTSP model.

25.2.1 Contribution

The contributions of the paper are the following.

- Identify an opportunity for improvement over the classical approach for basic block reordering, initiated by Pettis and Hansen [51]. Then they extend the model and suggest a new optimization problem with the objective closely related to the performance of a binary.
- Develop a new practical algorithm for basic block reordering. The algorithm relies on a greedy technique for solving the optimization problem.
- Propose a Mixed Integer Programming formulation for the aforementioned optimization problem, which is capable of finding optimal solutions on small functions

25.2.2 New ideas

In their study, they consider the following features.

- The length of a jump impacts the performance of instruction caches. Longer jumps are more likely to result in a cache miss than shorter ones. In particular, a jump with the length shorter than 64 bytes has a chance to remain within the same cache line.
- The direction of a branch plays a role for branch predicting. A branch s→t is called forward if s $<$ t, that is, block s precedes block t in the ordering; otherwise, the branch is called backward.
- The branches can be classified into unconditional (if the out-degree is one) and conditional (if the out-degree is two). A special kind of branches is between consecutive blocks in the ordering that are called fall-through; in this case, a jump instruction is not needed.
- They introduce a new score that estimates the quality of a basic block ordering taking into account the branch characteristics. In the most generic form, the new function, called EXTENDED TSP (EXTTSP), is expressed as follows:

$$\text{ExtTSP} = \sum_{(s,t)} w(s,t) \times K_{s,t} \times h_{s,t}(\text{len}(s,t))$$

where the sum is taken over all branches in the control flow graph. Here $w(s,t)$ is the frequency of branch s→t and $0 \leq K_{s,t} \leq 1$ is a weight coefficient modeling the relative importance of the branch for optimization. We distinguish six types of branches arising in code: conditional and unconditional versions of fall-through, forward, and backward branches. Thus, we introduce six coefficients for EXTTSP. The lengths of the jumps are accounted in the last term of the expression, which increases the importance of short jumps. A non-negative function $h_{s,t}(len(s,t))$ is defined by value of 1 for zero-length jumps, value of 0 for jumps exceeding a prescribed length, and it monotonically decreases between the two values. To be consistent with the objective of TSP, the EXTTSP score needs to be maximized for the best performance. Notice that EXTTSP is a generalization of TSP, as the latter can be modeled by setting $K_{s,t} = 1, h(len(s,t)) = 1$ for fall-through branches and $K_{s,t} = 0$ otherwise.

They use machine learning methods to find parameters for EXTTSP that have the highest correlation with the performance of a binary in the experiment.

$$\text{ExtTSP} = \sum_{(s,t)} w(s,t) \times \begin{cases} 1 & \text{if } \text{len}(s,t) = 0, \\ 0.1 \cdot \left(1 - \frac{\text{len}(s,t)}{1024}\right) & \text{if } 0 < \text{len}(s,t) \leq 1024 \\ & \text{and } s < t, \\ 0.1 \cdot \left(1 - \frac{\text{len}(s,t)}{640}\right) & \text{if } 0 < \text{len}(s,t) \leq 640 \\ & \text{and } t < s, \\ 0 & \text{otherwise.} \end{cases}$$

. Intuitively, EXTTSP resembles the traditional TSP model, as the number of fall-through branches is the dominant factor. The main difference is that EXTTSP rewards longer jumps. The impact of such jumps is significantly lower and it linearly decreases with the length of a jump. Next we summarize our high-level observations regarding the new score function.

25.2.3 Algorithm

Algorithm 16 Basic Block Reordering

Input: control flow graph $G = (V, E, w)$, the entry point $v^\star \in V$
Output: ordering of basic blocks $(v^\star = B_1, B_2, \ldots, B_{|v|})$
function ReorderBasicBlocks
  for $v \in V$ **do**
    $Chains \leftarrow Chains \cup (v)$
  end for
  while $|Chains| > 1$ **do** ▷ chain merging
    for $c_i, c_j \in Chains$ **do**
      $gain[c_i, c_j] \leftarrow$ ComputeMergeGain(c_i, c_j)
    end for
    src, $dst \leftarrow \arg\max_{i,j} \text{gain}\,[c_i, c_j]$ ▷ find best pair of chains
    $Chains \leftarrow Chains \cup Merge(src, dst) \backslash \{src, dst\}$; ▷ merge the pair and update chains
  end while
  return ordering given by the remaining chain;
end function
function ComputeMergeGain(src, dst)
  for $i = 1$ **to** blocks(src) **do** ▷ try all ways to split chain src
    $s_1 \leftarrow src[1 : i]$ ▷ break the chain at index i
    $s_2 \leftarrow src[i + 1 : blocks(src)]$
    score $_i \leftarrow \max \begin{cases} \text{ExtTSP}\,(s_1, s_2, dst) & \text{if } v^* \notin dst \\ \text{ExtTSP}\,(s_1, dst, s_2) & \text{if } v^* \notin dst \\ \text{ExtTSP}\,(s_2, s_1, dst) & \text{if } v^* \notin s_1, dst \\ \text{ExtTSP}\,(s_2, dst, s_1) & \text{if } v^* \notin s_1, dst \\ \text{ExtTSP}\,(dst, s_1, s_2) & \text{if } v^* \notin src \\ \text{ExtTSP}\,(dst, s_2, s_1) & \text{if } v^* \notin src \end{cases}$ ▷ try all valid ways to concatenate
  end for
  return $\max_i$ score $_i$ − ExtTSP(src) − ExtTSP(dst) ▷ the gain of merging chains src and dst
end function

26 Destructive Compiler Optimizatios

In (say) the early 1990s, compilers did fewer optimizations, in part because there were fewer compiler writers and in part due to the relatively small memories of that era. Nevertheless, problems did arise, as shown in Listing 4.14, which the compiler is within its rights to transform into Listing 4.15. As you can see, the temporary on line 1 of Listing 4.14 has been optimized away, so that global_ptr will be loaded up to three times.

Given code that does plain loads and stores,[10] the compiler is within its rights to assume that the affected variables are neither accessed nor modified by any other thread. This assumption allows the compiler to carry out a large number of transformations, including load tearing, store tearing, load fusing, store fusing, code reordering, invented loads, invented stores, store-to-load transformations, and dead-code elimination, all of which work just fine in single-threaded code. But concurrent code can be broken by each of these transformations, or shared-variable shenanigans, as described below.

26.1 Load tearing

Load tearing occurs when the compiler uses multiple load instructions for a single access. For example, the compiler could in theory compile the load from global_ptr (see line 1 of Listing 4.14) as a series of one-byte loads. If some other thread was concurrently setting global_ptr to NULL, the result might have all but one byte of the pointer set to zero, thus forming a "wild pointer". Stores using such a wild pointer could corrupt arbitrary regions of memory, resulting in rare and difficult-to-debug crashes.

Worse yet, on (say) an 8-bit system with 16-bit pointers, the compiler might have no choice but to use a pair of 8-bit instructions to access a given pointer. Because the C standard must support all manner of systems, the standard cannot rule out load tearing in the general case.

26.2 Store tearing

Store tearing occurs when the compiler uses multiple store instructions for a single access. For example, one thread might store 0x12345678 to a four-byte integer variable at the same time another thread stored 0xabcdef00. If the compiler used 16-bit stores for either access, the result might well be 0x1234ef00, which could come as quite a surprise to code loading from this integer. Nor is this a strictly theoretical issue. For example, there are CPUs that feature small immediate instruction fields, and on such CPUs, the compiler might split a 64-bit store into two 32-bit stores in order to reduce the overhead of explicitly forming the 64-bit constant in a register, even on a 64-bit CPU. There are historical reports of this actually happening in the wild.

26.3 Load fusing

Load fusing occurs when the compiler uses the result of a prior load from a given variable instead of repeating the load. Not only is this sort of optimization just fine in single-threaded code, it is often just fine in multithreaded code. Unfortunately, the word "often" hides some truly annoying exceptions. For example, suppose that a real-time system needs to invoke a function named do_something_quickly() repeatedly until the variable need_ to_stop was set, and that the compiler can see that do_something_ quickly() does not store to need_to_stop. One (unsafe) way to

[10] That is, normal loads and stores instead of C11 atomics, inline assembly, or volatile accesses.

code this is shown in Listing 4.16. The compiler might reasonably unroll this loop sixteen times in order to reduce the per-invocation of the backwards branch at the end of the loop. Worse yet, because the compiler knows that do_ something_quickly() does not store to need_to_stop, the compiler could quite reasonably decide to check this variable only once, resulting in the code shown in Listing 4.17. Once entered, the loop on lines 2–19 will never exit, regardless of how many times some other thread stores a non-zero value to need_to_stop. The result will at best be consternation, and might well also include severe physical damage.

The compiler can fuse loads across surprisingly large spans of code. For example, in Listing 4.18, t0() and t1() run concurrently, and do_ something() and do_something_else() are inline functions. Line 1 declares pointer gp, which C initializes to NULL by default. At some point, line 5 of t0() stores a non-NULL pointer to gp. Meanwhile, t1() loads from gp three times on lines 10, 12, and 15. Given that line 13 finds that gp is non-NULL, one might hope that the dereference on line 15 would be guaranteed never to fault. Unfortunately, the compiler is within its rights to fuse the read on lines 10 and 15, which means that if line 10 loads NULL and line 12 loads &myvar, line 15 could load NULL, resulting in a fault. 8 Note that the intervening READ_ONCE() does not prevent the other two loads from being fused, despite the fact that all three are loading from the same variable.

26.4 Store fusing

Store fusing can occur when the compiler notices a pair of successive stores to a given variable with no intervening loads from that variable. In this case, the compiler is within its rights to omit the first store. This is never a problem in single-threaded code, and in fact it is usually not a problem in correctly written concurrent code. After all, if the two stores are executed in quick succession, there is very little chance that some other thread could load the value from the first store.

However, there are exceptions, for example as shown in Listing 4.19. The function shut_it_down() stores to the shared variable status on lines 3 and 8, and so assuming that neither start_shutdown() nor finish_ shutdown() access status, the compiler could reasonably remove the store to status on line 3. Unfortunately, this would mean that work_ until_shut_down() would never exit its loop spanning lines 14 and 15, and thus would never set other_task_ready, which would in turn mean that shut_it_down() would never exit its loop spanning lines 5 and 6, even if the compiler chooses not to fuse the successive loads from other_ task_ready on line 5.

And there are more problems with the code in Listing 4.19, including code reordering.

26.5 Code reordering

Code reordering is a common compilation technique used to combine common subexpressions, reduce register pressure, and improve utilization of the many functional units available on modern superscalar microprocessors.

It is also another reason why the code in Listing 4.19 is buggy. For example, suppose that the do_more_work() function on line 15 does not access other_task_ready. Then the compiler would be within its rights to move the assignment to other_task_ready on line 16 to precede line 14, which might be a great disappointment for anyone hoping that the last call to do_ more_work() on line 15 happens before the call to finish_shutdown() on line 7.

It might seem futile to prevent the compiler from changing the order of accesses in cases where the underlying hardware is free to reorder them. However, modern machines have exact exceptions

and exact interrupts, meaning that any interrupt or exception will appear to have happened at a specific place in the instruction stream. This means that the handler will see the effect of all prior instructions, but won't see the effect of any subsequent instructions. READ_ONCE() and WRITE_ONCE() can therefore be used to control communication between interrupted code and interrupt handlers, independent of the ordering provided by the underlying hardware. 9

26.6 Invented loads

Invented loads were illustrated by the code in Listings 4.14 and 4.15, in which the compiler optimized away a temporary variable, thus loading from a shared variable more often than intended.

Invented loads can be a performance hazard. These hazards can occur when a load of variable in a "hot" cacheline is hoisted out of an if statement. These hoisting optimizations are not uncommon, and can cause significant increases in cache misses, and thus significant degradation of both performance and scalability.

26.7 Invented stores

Invented stores can occur in a number of situations. For example, a compiler emitting code for work_until_shut_down() in Listing 4.19 might notice that other_task_ready is not accessed by do_more_work(), and stored to on line 16. If do_more_work() was a complex inline function, it might be necessary to do a register spill, in which case one attractive place to use for temporary storage is other_task_ready. After all, there are no accesses to it, so what is the harm?

Of course, a non-zero store to this variable at just the wrong time would result in the while loop on line 5 terminating prematurely, again allowing finish_shutdown() to run concurrently with do_more_work(). Given that the entire point of this while appears to be to prevent such concurrency, this is not a good thing.

Using a stored-to variable as a temporary might seem outlandish, but it is permitted by the standard. Nevertheless, readers might be justified in wanting a less outlandish example, which is provided by Listings 4.20 and 4.21.

A compiler emitting code for Listing 4.20 might know that the value of a is initially zero, which might be a strong temptation to optimize away one branch by transforming this code to that in Listing 4.21. Here, line 1 unconditionally stores 1 to a, then resets the value back to zero on line 3 if condition was not set. This transforms the if-then-else into an if-then, saving one branch.

Finally, pre-C11 compilers could invent writes to unrelated variables that happened to be adjacent to written-to variables [Boe05, Section 4.2]. This variant of invented stores has been outlawed by the prohibition against compiler optimizations that invent data races.

26.8 Store-to-load transformations

Store-to-load transformations can occur when the compiler notices that a plain store might not actually change the value in memory. For example, consider Listing 4.22. Line 1 fetches p, but the "if" statement on line 2 also tells the compiler that the developer thinks that p is usually zero. 10 The barrier() statment on line 4 forces the compiler to forget the value of p, but one could imagine a compiler choosing to remember the hint—or getting an additional hint via feedback-directed optimization. Doing so would cause the compiler to realize that line 5 is often an expensive no-op.

Such a compiler might therefore guard the store of NULL with a check, as shown on lines 5 and 6 of Listing 4.23. Although this transformation is often desirable, it could be problematic if the actual

store was required for ordering. For example, a write memory barrier (Linux kernel smp_wmb()) would order the store, but not the load. This situation might suggest use of smp_store_release() over smp_wmb().

26.9 Dead-code elimination

Dead-code elimination can occur when the compiler notices that the value from a load is never used, or when a variable is stored to, but never loaded from. This can of course eliminate an access to a shared variable, which can in turn defeat a memory-ordering primitive, which could cause your concurrent code to act in surprising ways. Experience thus far indicates that relatively few such surprises will be at all pleasant. Elimination of store-only variables is especially dangerous in cases where external code locates the variable via symbol tables: The compiler is necessarily ignorant of such external-code accesses, and might thus eliminate a variable that the external code relies upon.

26.10 A Volatile Solution

The volatile keyword can prevent load tearing and store tearing in cases where the loads and stores are machine-sized and properly aligned. It can also prevent load fusing, store fusing, invented loads, and invented stores. However, although it does prevent the compiler from reordering volatile accesses with each other, it does nothing to prevent the CPU from reordering these accesses. Furthermore, it does nothing to prevent either compiler or CPU from reordering non-volatile accesses with each other or with volatile accesses. Preventing these types of reordering requires the techniques described in the next section.

References

[1] *Live Variables Analysis*. https://isoft.acm.org/winterschool17/presentation-decks/WSSE17-Day1-2-Uday-talks/live-vars.pdf. (Accessed on 12/19/2022).

[2] *Microsoft Word - lecture4.doc*. https://homes.cs.washington.edu/~bodik/ucb/cs264/lectures/4-chaotic-notes.pdf. (Accessed on 12/20/2022).

[3] *DATAFLOW ANALYSIS*. https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html. (Accessed on 12/17/2022).

[4] John B Kam and Jeffrey D Ullman. "Monotone data flow analysis frameworks". In: *Acta informatica* 7.3 (1977), pp. 305–317.

[5] Gary A Kildall. "A unified approach to global program optimization". In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, pp. 194–206.

[6] John H Reif and Harry R Lewis. "Symbolic evaluation and the global value graph". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 1977, pp. 104–118.

[7] Ben Wegbreit. "Property extraction in well-founded property sets". In: *IEEE Transactions on software engineering* 3 (1975), pp. 270–285.

[8] Mark N Wegman and F Kenneth Zadeck. "Constant propagation with conditional branches". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.2 (1991), pp. 181–210.

[9] Etienne Morel and Claude Renvoise. "Global optimization by suppression of partial redundancies". In: *Communications of the ACM* 22.2 (1979), pp. 96–103.

[10] Bernhard Steffen. "Data flow analysis as model checking". In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1991, pp. 346–364.

[11] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. "Sparse code motion". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2000, pp. 170–183.

[12] *293S_08_PRE_LCM*. https://sites.cs.ucsb.edu/~yufeiding/cs293s/slides/293S_08_PRE_LCM.pdf. (Accessed on 11/27/2022).

[13] *Microsoft PowerPoint - L12-Lazy-Code-Motion*. https://www.cs.cmu.edu/afs/cs/academic/class/15745-s16/www/lectures/L12-Lazy-Code-Motion.pdf. (Accessed on 11/27/2022).

[14] *Microsoft PowerPoint - L12-Interval-Analysis*. http://www.cs.cmu.edu/afs/cs/academic/class/15745-s13/public/lectures/L12-Interval-Analysis-1up.pdf. (Accessed on 11/27/2022).

[15] *Points-to Analysis*. https://engineering.purdue.edu/Cetus/Documentation/manual/ch07s05.html. (Accessed on 11/30/2022).

[16] Radu Rugina and Martin C Rinard. "Pointer analysis for structured parallel programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.1 (2003), pp. 70–116.

[17] *notes07-pointers.pdf*. http://www.cs.cmu.edu/~clegoues/courses/15-819O-16sp/notes/notes07-pointers.pdf. (Accessed on 12/04/2022).

[18] *REGISTER ALLOCATION*. https://pages.cs.wisc.edu/~horwitz/CS701-NOTES/5.REGISTER-ALLOCATION.html. (Accessed on 12/06/2022).

[19] Sebastian Hack, Daniel Grund, and Gerhard Goos. "Register allocation for programs in SSA-form". In: *International Conference on Compiler Construction*. Springer. 2006, pp. 247–262.

[20] Fernando Magno Quintao Pereira and Jens Palsberg. "Register allocation via coloring of chordal graphs". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2005, pp. 315–329.

[21] Gregory J Chaitin. "Register allocation & spilling via graph coloring". In: *ACM Sigplan Notices* 17.6 (1982), pp. 98–101.

[22] Keith D Cooper, Philip J Schielke, and Devika Subramanian. "An experimental evaluation of list scheduling". In: *TR98* 326 (1998).

[23] *Data dependency - Wikipedia*. https://en.wikipedia.org/wiki/Data_dependency. (Accessed on 12/07/2022).

[24] Philip B Gibbons and Steven S Muchnick. "Efficient instruction scheduling for a pipelined architecture". In: *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*. 1986, pp. 11–16.

[25] David Landskov et al. “Local microcode compaction techniques”. In: *ACM Computing Surveys (CSUR)* 12.3 (1980), pp. 261–294.

[26] Shen Lin and Brian W Kernighan. “An effective heuristic algorithm for the traveling-salesman problem”. In: *Operations research* 21.2 (1973), pp. 498–516.

[27] Monte Zweben et al. *Scheduling and rescheduling with iterative repair*. Tech. rep. 1992.

[28] Monte Zweben et al. “Learning to improve constraint-based scheduling”. In: *Artificial Intelligence* 58.1-3 (1992), pp. 271–296.

[29] John Whaley. “Partial method compilation using dynamic profile information”. In: *ACM SIGPLAN Notices* 36.11 (2001), pp. 166–179.

[30] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. “Partial escape analysis and scalar replacement for Java”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 2014, pp. 165–174.

[31] Toshio Suganuma et al. “Overview of the IBM Java just-in-time compiler”. In: *IBM systems Journal* 39.1 (2000), pp. 175–193.

[32] Michael Paleczny, Christopher Vick, and Cliff Click. “The Java {HotSpot™} Server Compiler”. In: *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. 2001.

[33] Michael G Burke et al. “The Jalapeno dynamic optimizing compiler for Java”. In: *Proceedings of the ACM 1999 conference on Java Grande*. 1999, pp. 129–141.

[34] Michał Cierniak, Guei-Yuan Lueh, and James M Stichnoth. “Practicing JUDO: Java under dynamic optimizations”. In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 2000, pp. 13–26.

[35] Toshio Suganuma et al. “A dynamic optimization framework for a Java just-in-time compiler”. In: *ACM SIGPLAN Notices* 36.11 (2001), pp. 180–195.

[36] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[37] Ken Kennedy. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division, 1979.

[38] Thomas Kotzmann and Hanspeter Mössenböck. “Escape analysis in the context of dynamic compilation and deoptimization”. In: *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. 2005, pp. 111–120.

[39] Arie Van Deursen, Paul Klint, and Joost Visser. “Domain-specific languages: An annotated bibliography”. In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.

[40] Kevin J Brown et al. “A heterogeneous parallel framework for domain-specific languages”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 89–100.

[41] Arvind Sujeeth et al. “OptiML: an implicitly parallel domain-specific language for machine learning”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 609–616.

[42] Tiark Rompf and Martin Odersky. “Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs”. In: *Proceedings of the ninth international conference on Generative programming and component engineering*. 2010, pp. 127–136.

[43] Tiark Rompf et al. “Building-blocks for performance oriented DSLs”. In: *arXiv preprint arXiv:1109.0778* (2011).

[44] *CUDA Zone - Library of Resources — NVIDIA Developer*. https://developer.nvidia.com/cuda-zone. (Accessed on 12/09/2022).

[45] Joseph A Fisher. “Walk-time techniques: Catalyst for architectural change”. In: *Computer* 30.9 (1997), pp. 40–42.

[46] Todd C Mowry, Monica S Lam, and Anoop Gupta. “Design and evaluation of a compiler algorithm for prefetching”. In: *ACM Sigplan Notices* 27.9 (1992), pp. 62–73.

[47] Monica D Lam, Edward E Rothberg, and Michael E Wolf. “The cache performance and optimizations of blocked algorithms”. In: *ACM SIGOPS Operating Systems Review* 25.Special Issue (1991), pp. 63–74.

[48] *GCD test - Wikipedia*. https://en.wikipedia.org/wiki/GCD_test. (Accessed on 12/15/2022).

[49] Antonia Zhai et al. “Compiler optimization of scalar value communication between speculative threads”. In: *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 2002, pp. 171–183.

[50] Andy Newell and Sergey Pupyrev. “Improved basic block reordering”. In: *IEEE Transactions on Computers* 69.12 (2020), pp. 1784–1794.

[51] Karl Pettis and Robert C Hansen. “Profile guided code positioning”. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, pp. 16–27.