Value Dependence Graphs: Representation Without Taxation

Daniel Weise Roger F. Crew Michael Ernst Bjarne Steensgaard

Microsoft Research*

Abstract

The *value dependence graph* (VDG) is a sparse dataflow-like representation that simplifies program analysis and transformation. It is a functional representation that represents control flow as data flow and makes explicit all machine quantities, such as stores and I/O channels. We are developing a compiler that builds a VDG representing a program, analyzes and transforms the VDG, then produces a control flow graph (CFG) [ASU86] from the optimized VDG. This framework simplifies transformations and improves upon several published results. For example, it enables more powerful code motion than [CLZ86, FOW87], eliminates as many redundancies as [AWZ88, RWZ88] (except for redundant loops), and provides important information to the code scheduler [BR91]. We exhibit a one-pass method for elimination of partial redundancies that never performs redundant code motion [KRS92, DS93] and is simpler than the classical [MR79, Dha91] or SSA [RWZ88] methods. These results accrue from eliminating the CFG from the analysis/transformation phases and using *demand dependences* in preference to control dependences.

1 Introduction

Program analysis is a prerequisite for important program transformations performed by compilers. The classical program representation for analysis is the *control flow graph* (CFG). CFG-based analyses and transformations suffer from two burdens: they must maintain CFG invariants to produce a semantically equivalent CFG after each code motion or other transformation, and they are sensitive to the names given to values. CFGs overspecify a computation by totally ordering its operations, requiring the addition of extra nodes (such as entry pads and exit pads [RWZ88]) to a CFG before performing code motion. CFGs are statement-based and name all values; the names, usually provided by the programmer, get in the way of analyzing the underlying computation.

During the last decade, new program representations have been proposed to alleviate these problems and make analysis simpler, faster, or more thorough. Static single assignment (SSA) form [CFR+89] gives each value a distinct name, which improves the efficiency of constant propagation and other analyses [WZ85, AWZ88, RWZ88, WZ89]. The dependence flow graph (DFG) improves the efficiency of analyses by ignoring irrelevant sections of code and linking definitions to uses [JP93]. These representations can be viewed as augmentations to the CFG that allow more rapid traversal of the CFG. The program dependence graph (PDG) eliminates some control artifacts by linking together operations with the same *control dependence*, and builds local data dependence graphs to simplify analysis and transformation [Ott78, FOW87]. The program dependence web (PDW) [BMO90, CKB93] makes value flow in the PDG more explicit by using gated single assignment (GSA) form.

The next step in solving the problems of CFGs is to eliminate the CFG as the basis of analysis and transformation. We represent a computation as a *value dependence graph* (VDG), which specifies only the value flow through a computation. The VDG does not depend upon information about values' names, the locations in which they reside, or when they are computed. The VDG is a parallel representation that specifies a partial order on the operations in the computation based solely upon data dependences, rather than some arbitrary total order specified by a CFG. VDG semantics are demand-based; a value is computed only if it is needed by another computation. Evaluation of the

*Authors' address: One Microsoft Way, Redmond, WA 98052. Email: {dweise,rfc,mernst,rusa}@research.microsoft.com.
This is a slightly revised version of a paper that appeared in the Conference Record of the Twenty First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, January, 1994, pp.297-310.

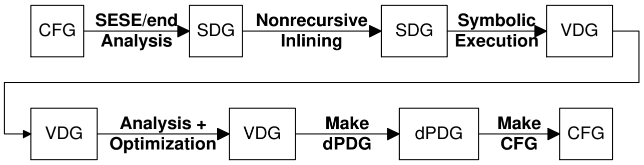

Figure 1: The stages in the translation from the initial control flow graph (CFG) to the final CFG. The parser creates the initial CFG, and the back end generates machine code from the final CFG.

VDG may terminate even if the original program would not, because CFG values that are not demanded do not appear in the VDG.

Code generation from CFGs is straightforward, but the CFG's limitations—the requirement to maintain a total ordering on program statements and the CFG's statement-based nature—complicate analyses and transformations. We claim that is the wrong tradeoff, since an optimizing compiler contains many analyses and transformations but only one conversion from its intermediate representation into (say) CFG form in the back end. Analysis and transformation is simpler to implement, understand, and express formally, and frequently faster, when using a VDG rather than a CFG. While simple-minded code generation from the VDG may result in poor code, the representation provides important information to the code scheduler [BR91] which makes code movement and scheduling considerably easier. We believe the code generator is the right place for the complexity of generating sequential code, rather than taxing every analysis and transformation with the burden of maintaining structures that are irrelevant to the operation being applied.

Figure 1 shows the steps in the translation from CFG to VDG form and then, after analysis and optimization, back to CFG form. We use the `example` program of Figure 2 to illustrate the resulting transformations. Figures 3–5 show these transformations in the CFG framework, and Figures 6, 8, and 9 show intermediate steps in the translation from CFG to VDG, which is discussed in detail later.

Optimizations that can be performed independently of the final order of evaluation are performed on the VDG. For example, global constant and copy propagation, constant folding, global name-insensitive common subexpression elimination, and dead code elimination are performed directly on the VDG. Code motion optimizations are decided when the *demand dependence graph* is constructed from the VDG. A demand dependence graph is similar in spirit to a control dependence graph (CDG) [FOW87], except that the demand dependence graph is constructed using the predicates that lead to a computation contributing to the output of program, whereas the predicates in a CDG are those

```
int example(int a, int b, int c, int d)
{
    int acopy, bcopy, lp_inv1, lp_inv2;
    int down, cse, epr, dead;
    do {
        bcopy   = b;
        lp_inv1 = c + bcopy;
        lp_inv2 = d - b;
        a       = a * lp_inv1;
        down    = a % c;
        dead    = a + d;
        if (a > d) {
            acopy = a;
            a     = down + 3;
            cse   = acopy << b;
        }
        else
          cse = a << bcopy;
        epr = a << b;
    } while (a > cse);
    return lp_inv2 + epr;
}
```

Figure 2: Example procedure containing loop invariant computations, common subexpressions, partial redundancies, and dead computations. Figures 3–5 show the transformations performed on this procedure's CFG by translation into VDG form and back or by VDG manipulations.

that lead to a computation being performed. Optimizations that depend upon the final placement of code are performed on the VDG after the demand dependence graph is constructed. Figure 4 demonstrates the results of several transformations performed on the VDG.

VDGs are the basis of powerful, precise, and efficient program slicing algorithms. Efficient algorithms for interprocedural slicing, slicing unstructured programs, and interactive slicing operate directly upon the VDG [Ern93].

The next section of this paper describes the VDG and the translation from CFGs into VDGs (the first three steps of Figure 1). Analyses and transformations are discussed in Section 3. Section 4 discusses one aspect of producing CFGs from VDGs, the construction of the *demand-based program dependence graph* (dPDG). ([Ste93a] describes the translation from the dPDG to the final CFG.) Section 5 shows that our representation affords an algorithm for elimination of partial redundancies (EPR) that is simpler and faster than the classical or SSA formulations. This algorithm uses the dPDG that was constructed for code generation. Section 6 compares our research to related work, and the last section discusses future research directions.

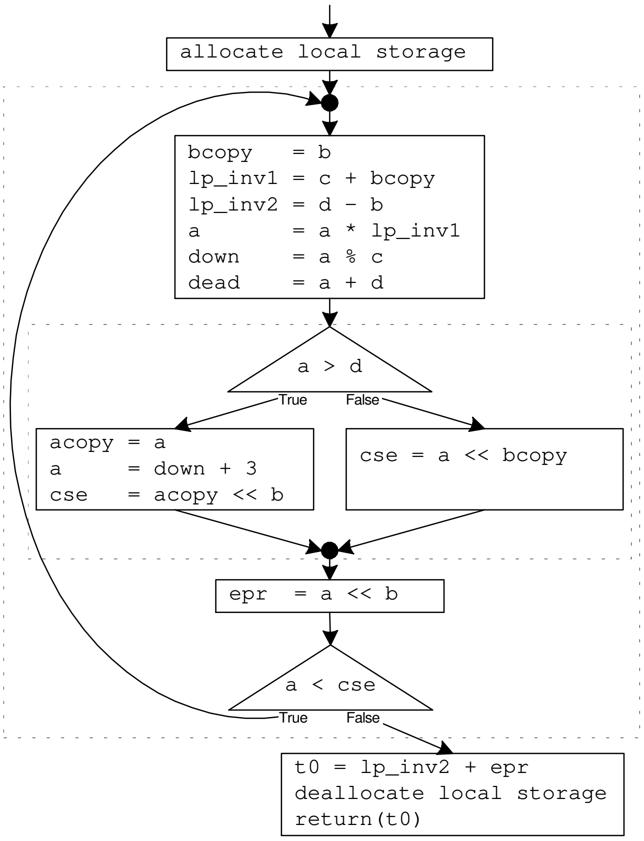

Figure 3: Original CFG for `example` (Figure 2). Our CFGs, unlike those of [ASU86], explicitly represent branches and joins. Dotted boxes enclose irreducible SESE regions (see Section 2.1).

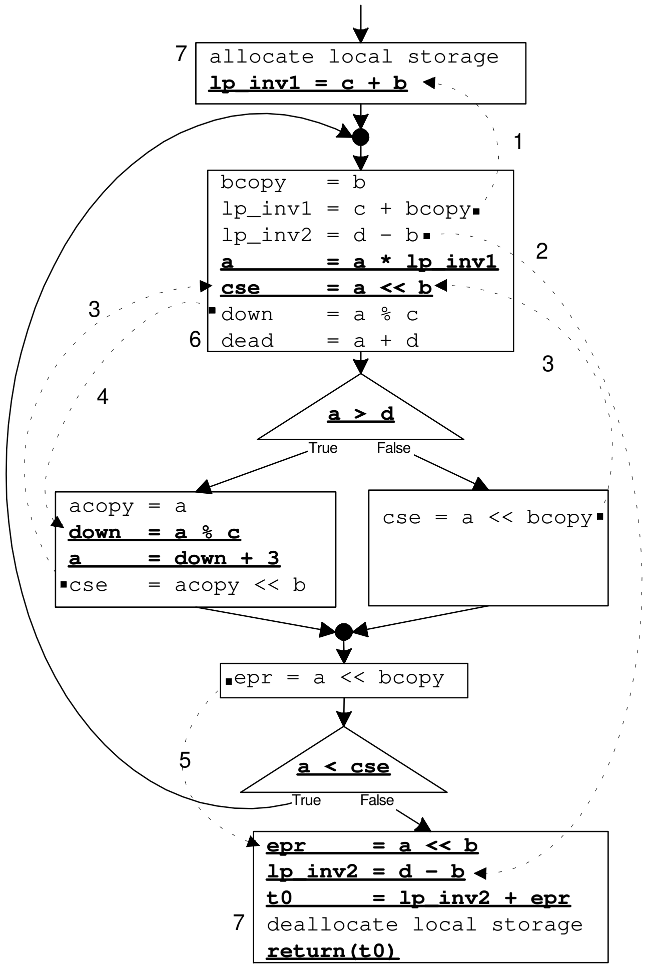

Figure 4: Transformations on `example` done by our analysis, shown in the CFG framework. These transformations are not performed on the CFG, but occur as a result of translations into VDG form and back, or are done on the VDG. Statements in the light font are moved or removed by the transformations; they do not appear in the final CFG (Figure 5). Statements in the heavy underlined font do appear in the final CFG. The transformations that occur in this example are: 1) Hoisting a loop invariant above the loop; 2) Lowering a loop invariant below the loop; 3) Name insensitive global common subexpression elimination; 4) Code motion into the arm of an `if` statement; 5) Lowering a non-loop invariant expression below (outside) a loop (the value of this expression is not consumed by the loop); 6) Dead code elimination; and 7) Decoupling values from the store. The analysis also reveals that `example` can be run in parallel with other functions that do not depend on its result.

2 Value Dependence Graphs

The VDG is a functional representation that expresses the computation of a procedure solely as value flow. The lack of control flow has three immediate consequences. First, all machine quantities that are usually implicit in other representations, such as store contents, stack and heap allocators, input and output channels, and C volatile locations, must be explicitly represented as value flow to ensure that state-changing operations occur in the correct order. To simplify the exposition, we will consider the machine state to consist solely of a store. Second, operators that choose among control paths (*e.g.*, `if` and `switch` statements) are represented by *selectors* that choose among possible values; these are essentially the γ nodes of [BMO90]. Third, looping is represented via function calls: the CFG backedge finds its analog in the VDG recursive call.

A VDG consists of

1. A directed bipartite graph whose vertices are *nodes* representing computations and *ports* representing values. The graph arcs connect from nodes to their

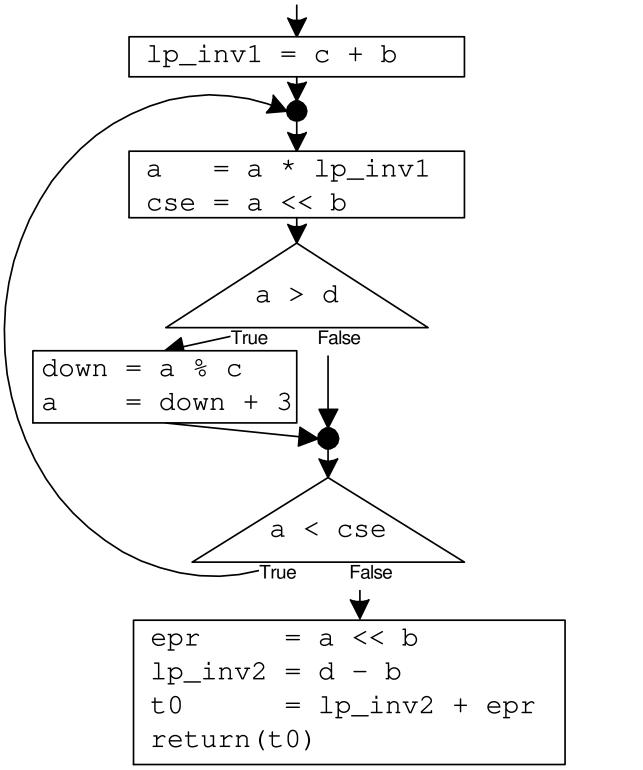

Figure 5: The final CFG of `example`, which results from translation of the VDG of Figure 9 into a CFG.

operand values (ports) and from ports to the computations (nodes) producing them as *results*. Each port is either produced by exactly one node or is a *free value* not produced by any node. Nodes are of the following kinds:

- primitive operation (*primop*) nodes, including basic arithmetic and data structure operations, constants, etc.
- boolean (or integral) selector (γ) nodes representing conditional expressions, which produce a single result from a predicate (integer) operand and two (or more) alternative value operands;
- closure (λ) nodes that produce function values;
- function application (*call*) nodes taking a function operand and actual parameter operands corresponding to the formal parameters of that function's VDG (see 3 below) and producing result ports corresponding to the function's VDG return nodes;
- formal *parameter* nodes with no operands whose result is a parameter value; and
- *return* nodes whose operand is a return value, and which produce no results—every VDG contains at least one return node, one for each value it returns.

All cycles in the graph include at least one λ node.

2. Indexings[1] for the following sets: parameter nodes, return nodes, operand arcs for each node, and result arcs for each node.

3. A *body* for each λ node, namely, a VDG and a bijection between the free values of that VDG and the operands of the λ (in practice, we simply identify the free values with the λ operands; note that the λ operands are *not* its parameters).

Figures 8 and 9 show VDGs.

The free values of a VDG can be viewed as analogous to the free variables in a lambda term. If a given VDG is the body of some λ node, its free values denote external values (*e.g.*, external functions, addresses of global variables) that all calls of that λ node have in common. The top-level VDG for a complete program, which takes the form of a function producing a final machine state (in this paper, a store) from an initial machine state, has no free values.

While VDGs are slightly larger than CFGs, we do not anticipate that they will prove too large for practical use. Experiments with *thinned gated single assignment* (TGSA) form [Hav93] indicate that in practice, TGSA can be constructed in time and space linear in the number of variable references for programs "satisfying reasonable assumptions." We would be surprised if the same result did not hold for VDG form, which is similar to TGSA form.

Sections 2.1 through 2.3 describe the three steps in constructing a procedure's VDG. The first step, SESE/end analysis, constructs a *store dependence graph* (SDG), which is a translation of the CFG into VDG form with basic blocks left uninterpreted.[2] Unstructured control flow is modeled via procedure calls. The second step, inlining of nonrecursive calls, introduces extra γ nodes that allow inlining without code duplication. The third step is symbolic execution of the SDG (and each closure therein), which interprets the basic blocks to produce a VDG.

2.1 Construction of the Store Dependence Graph

A store dependence graph (SDG) is a VDG having the following node kinds in place of primop nodes:

[1] For an indexing of a set S, a bijection $\{1, 2, \ldots, |S|\} \leftrightarrow S$ suffices. This establishes the correspondence of call operands with λ parameters, call results with λ returns, selector branches with predicate values, etc.

[2] The construction of the SDG is an expository fiction. The implementation finds the information necessary to construct the SDG but uses this information to guide symbolic execution rather than to construct the SDG.

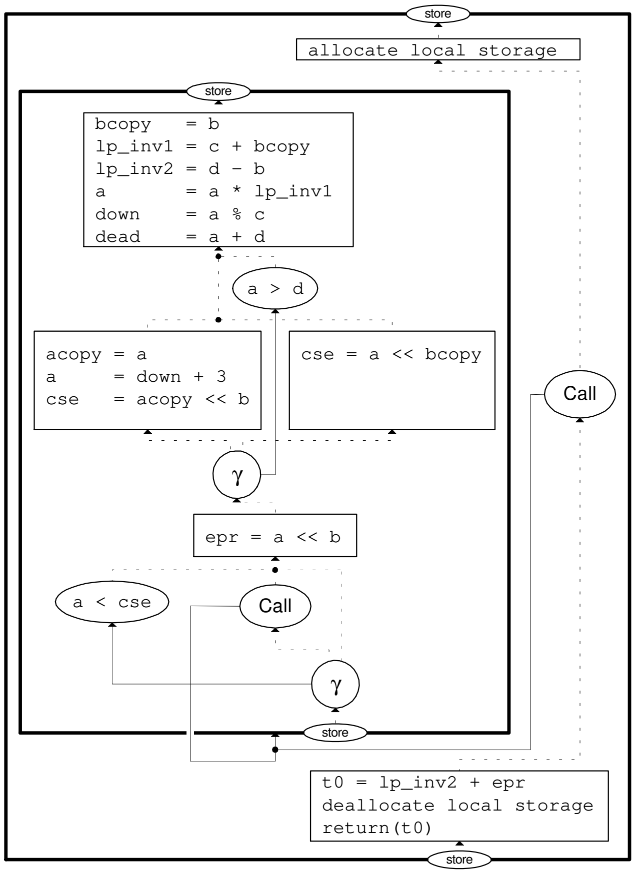

Figure 6: SDG for `example`, which was produced from the CFG of Figure 3. Our SDG and VDG diagrams suppress the ports, since the computations here are single-valued. Each heavy box denotes a λ node, which produces a function as its result. Arcs leaving such a box (*e.g.*, the function operand arc from the inner Call node) denote the λ's free value operands. Parameter and return nodes are embedded in its upper and lower boundaries respectively. Dotted lines indicate store values; solid lines are scalar or function values.

- *block nodes* which produce a next-store from a previous-store operand, and
- *predicate nodes* which produce a predicate value from a store operand.

The initial CFG is assumed to have explicit join nodes and entry and exit nodes. In addition, statements reside in CFG basic block nodes having just one predecessor and one successor; branches contain only the predicate (switch) value and joins have no content whatsoever. Each block or predicate node in the resulting SDG corresponds to a particular CFG basic block or branch.

Given a CFG node b and one of its postdominators e, consider the function giving the store prior to e as a function of the store prior to b. The SDG is a representation of this function for the case where b is the entry and e is the exit. In this way, control flow is replaced by (store) value flow. Figure 6 shows the SDG for `example`.

The problem thus reduces to constructing the SDG fragment corresponding to a $b \to e$ "traversal" of the CFG.

- A basic block is replaced by a corresponding basic block SDG node applied to the store operand. CFG traversal proceeds from the block's successor.
- To translate a branch node, one of its postdominators is designated as the *branch-end* (see below). A sequence of stores is obtained by traversing the CFG from each branch successor in turn to the branch-end, then the stores are combined via a γ node. CFG traversal proceeds from the branch-end.
- A join indicative of looping or unstructured control (see below) translates to a λ node with a Call node for each join predecessor. Such λ nodes are called *internal* λ nodes, to distinguish them from functions explicitly defined by the programmer; the corresponding Calls are eventually converted into `goto`s by the code generator. As with branch translation, join translation requires the designation of a postdominator, the *join-end*, both to designate where the CFG traversal continues after the Call and to produce the λ body SDG by traversing the CFG from the join successor to the join-end.
- The remaining joins (*e.g.*, the lower join in Figure 3) are merely mergings of distinct paths from a branch; no λ or Call nodes are necessary.

Whether a particular join needs a corresponding internal λ and how to designate branch/join-ends are resolved by *single-entry-single-exit (SESE) analysis* [JP93]. A pair of distinct CFG nodes a, b *encloses a SESE region* iff there exist arcs α ending at a and β beginning at b such that α dominates β, β postdominates α, and α and β are cycle-equivalent (*i.e.*, every CFG graph cycle contains both or neither). The SESE region itself consists of a, b, and all nodes that are dominated by a and postdominated by b. Merging all consecutive joins in the original CFG prevents artifacts resulting from ordering of joins. Unless otherwise specified, "SESE region" refers to a nontrivial SESE region, one which is neither a single basic block nor the composition of other SESE regions. A node d is a *SESE-bottom* if, for some node c, c and d enclose a SESE region. Finally, for each CFG node a, let $bottom(a)$ be the postdominator of the smallest SESE region containing a; $bottom(a)$ can be found for all nodes in $O(E)$ time [JPP93]. A join is symptomatic of looping or unstructured control iff it is not a SESE-bottom.

End analysis associates with each branch (join) node a *branch-end* (*join-end*). The analysis depends on whether the node is a SESE-bottom or not. If a branch is a SESE-bottom node, the branch-end is that branch's unique successor that is outside the SESE region. If a join is a SESE-bottom node, it is its own join-end. When determining the end for a non-SESE-bottom branch or join n, let z be the end for $bottom(n)$. Nodes in contained SESE regions (that is, nodes a for which $bottom(a) \neq bottom(n)$) are ignored. The branch-end for a non-SESE-bottom branch node n is its nearest postdominator that is either a join node or z. The join-end of a non-SESE-bottom join n is its nearest postdominator that is either a branch-end or z. End analysis depends only on postdominator trees and takes linear time [Har85].

The important property of this construction is that each CFG basic block and predicate appears exactly once within the resulting SDG, as a consequence of each CFG node being visited exactly once in the CFG traversal. The rules for determining branch/join-ends were chosen to minimize the number of Call nodes generated, though there are numerous variations worth exploring.

2.2 The λ-γ Transform and Inlining Nonrecursive Calls

Initially, the SDG models looping and unstructured control via Call and λ nodes. To simplify analysis and transformation, the λ-γ *transform* consolidates Calls corresponding to unstructured control flow, ensuring that each recursive λ has only one external Call. This permits inlining without code duplication, after which Call and λ only model looping (and programmer-specified procedures and procedure calls). The result of this process is similar to *thinned gated single assignment* (TGSA) [Hav93] except that the γ trees take stores, rather than values, as inputs. (Later passes produce more TGSA-like structures in which the γ trees operate on values rather than upon stores.)

Before the λ-γ transform, call loops are discovered, which identifies internal λs that are loop entry points and identifies recursive Calls, as well as finding loops with multiple entry points. The λ-γ transform for a particular internal λ proceeds in three steps. First, the control dependence tree is projected to contain only paths leading to nonrecursive Calls of the λ. (Symbolic execution tags all operations with their control dependences.) Second, a γ tree is generated by converting predicates both of whose branches are contained in the projection into γ nodes and omitting other predicates. The final step depends on whether the λ is called recursively. If the λ is called recursively, a new call is created whose input is the γ tree, with the operands of the original calls at its leaves. This new call replaces all the old non-recursive calls, so that there is only one non-recursive call to each internal λ. If the λ is not called recursively, the third step inlines the λ body by making consumers of its store input consume the γ tree instead and making consumers of the Calls of the λ consume the outputs of the λ body. The λ-γ transform handles irreducible loops in the same fashion, collecting and treating together all calls into the loop.

For example, consider the simple unstructured program CFG of Figure 7. Its upper join point is not a SESE-bottom. The corresponding SDG for this CFG contains a λ and two Calls to that λ, one for each possible path from the entry of the CFG to the unstructured join point. The λ-γ transform produces a new γ node that selects between the stores on the two paths to the join, predicates the γ on the fork that initiates the two separate paths, then inlines the body of the λ by replacing its store parameter node by the just constructed γ node. The λ-γ transform is illustrated by the two SDGs of Figure 7.

The γ nodes introduced by this pass represent additional representational overhead of our form over SSA form. Experiments have shown that this overhead is a linear function of the size of programs in SSA form [Hav93].

2.3 Symbolic Execution

Symbolic execution expands the SDG's unevaluated basic blocks into VDG nodes. During the expansion, global value numbering, copy propagation, and constant propagation that can be performed without recourse to fixed point analysis are automatically performed. Figure 8 shows the VDG that results from symbolically executing `example`'s SDG.

Symbolic execution occurs independently for the body of each λ node, each basic block of which is visited in turn after all of its predecessors have been processed. Symbolic execution uses a data structure called a *symbolic store* that maps locations to VDG ports representing the values currently known to be stored there. Essentially, the symbolic store represents what is known about the store operand of the basic block SDG node being executed.

The first basic block of the SDG is supplied with a symbolic store that contains information about globally allocated locations, but not necessarily about their contents.[3] Symbolic execution of a basic block updates the symbolic store, so that it represents what is known about the result store that gets passed on to the basic block's successors. Symbolic execution produces VDG nodes according to the following rules:

Variable Lookup If the variable's location in the symbolic store contains a value that was placed there by a previous update (see below), return that value. If the location contains no explicit value, build a VDG lookup node representing the corresponding runtime load operation and return its result port. If the store is actually the result of a γ node, then

[3] [Ste93b] describes our handling of store operations when the location being looked up or modified is not yet known due to pointer operations.

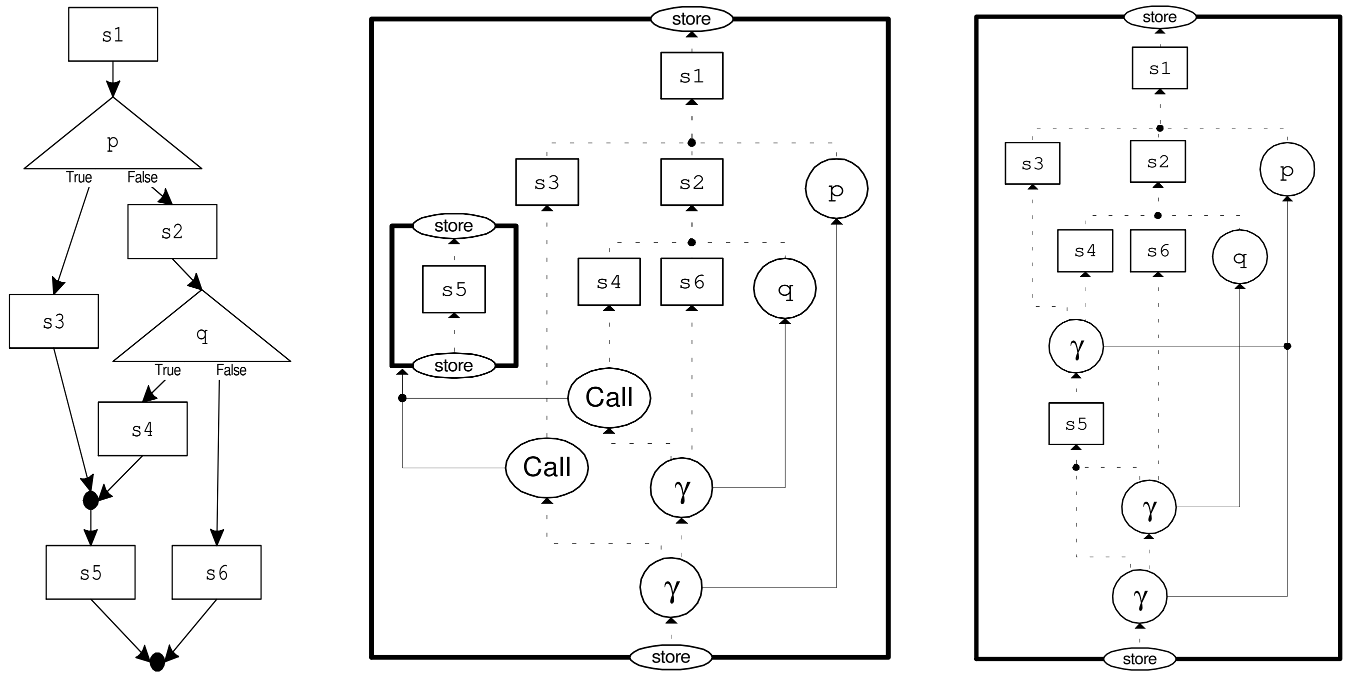

Figure 7: The CFG on the left contains unstructured control flow, which results in non-recursive λ nodes in its corresponding SDG (middle figure). These non-recursive λs prevent accurate analysis because of the lack of γ nodes to indicate within the λ which parameter values come from which paths through the program. That is, there is no way to track which Call supplied a particular parameter value. We solve this problem by constructing γ trees that indicate where the inputs come from, and then inlining the λ using the γ as its input (right illustration).

create a γ node whose "then" ("else") operand is the result of looking up the variable in the "then" ("else") store. Constant folding is performed here: if the "then" and "else" values are the same port, simply return it.

Expression Evaluation If a given expression can be evaluated statically, then do so and return the value; otherwise build a VDG node that represents the runtime execution of the expression and return its result port. Nodes are cached, so attempts to construct a node whose operation and operands match an existing node will merely retrieve that existing node. Caching, also known as *global value numbering* [Coc70, CS70], implements common subexpression elimination (CSE)[4] by representing multiple (not necessarily identical) source expressions by a single node.

Variable Update Given an assignment statement, update the location in the symbolic store to contain the value (VDG port) obtained by symbolically executing the expression part of the assignment statement. The updated store is used to symbolically execute the program after the assignment, which makes the assigned value available to subsequent lookups.

[4]We follow [ASU86] in defining two expressions to be common subexpressions if they compute the same value and one dominates the other.

Symbolic execution performs no interprocedural analysis of a Call's effect on, or use of, the store; later passes perform that work. Therefore, symbolic execution of a Call results in a store that has no information about the contents of locations. Lookups in this store during symbolic execution create lookup nodes. Additionally, all values must be *homed* (placed in their single globally-known locations, not in temporary storage) before each call and return. This is done by constructing an Update Store node whose arguments are all of the (changed) values and (corresponding) locations in the symbolic store.

3 Analysis and Transformation of VDGs

The VDG produced by symbolic execution is too coarse for good program analysis because too much of the computation depends upon store operations, which inhibit (instruction level) parallelism. The first several analyses/transformations reduce the reliance on the store, making the VDG sparser and more parallel. Then standard transformations, such as dead code elimination, are performed. Transformations for arity raising, both on parameter and return values, are also performed.

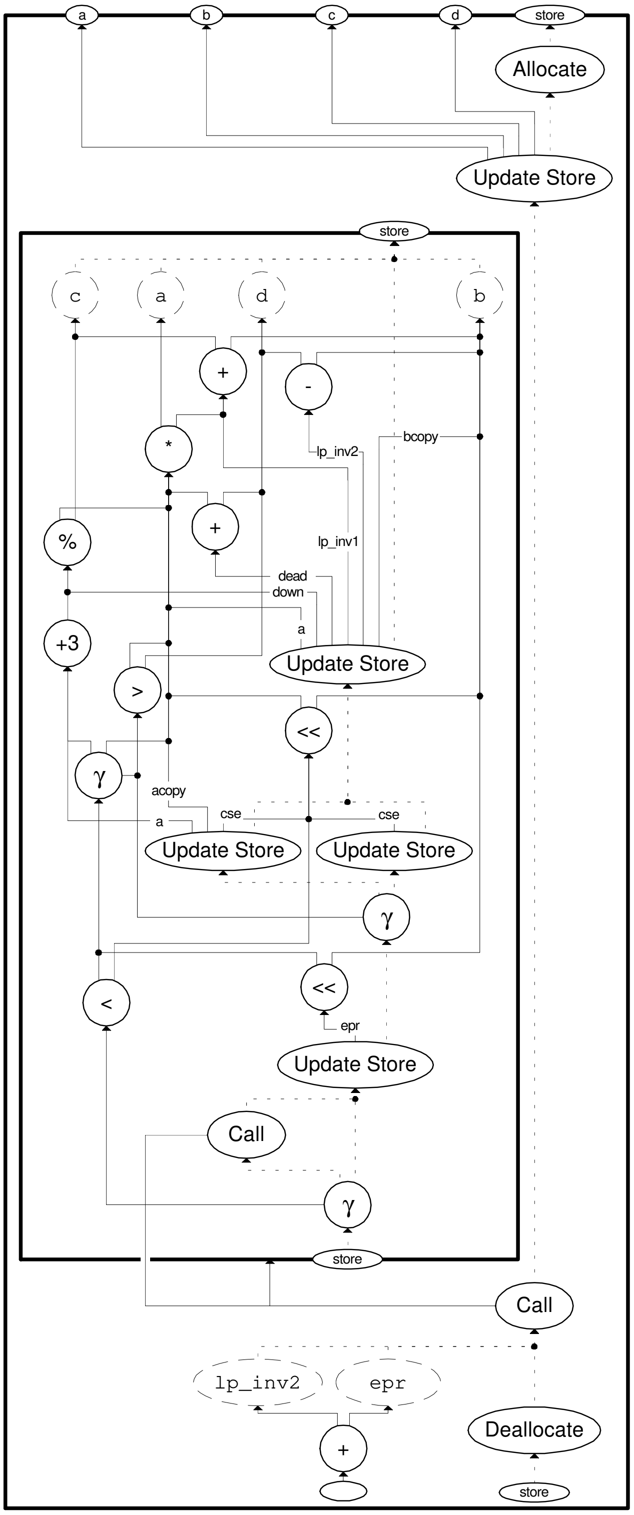

Figure 8: The VDG of `example` (Figure 2) after symbolic execution. The dashed circles taking a store operand are lookup (load) operations. For clarity, the location arguments to lookup and Update Store nodes are suppressed, but their values (which are produced by the Allocate node) are indicated by labels on the lookup nodes and on the value operand arcs for Update Store nodes.

The VDG representation makes many transformations simpler to state and to prove correct, and their implementations more efficient.

Our formulation does not admit a standard dataflow model for proving properties of program points because our model has no program points. Instead, analysis proves invariants about runtime values. Each node is given an abstract interpretation, and a fixpoint engine finds a fixpoint of the equations specified by the VDG and the abstract functions. This framework accommodates both forward flow problems and backward flow problems; the latter are all variants of the question, "What information does the rest of the program demand?" For example, dead update elimination, which eliminates operations on the store, is a backwards flow analysis that determines which elements of the store are demanded by the rest of the program. The VDG is called a sparse representation for analysis because information flows directly to where it is consumed.

Loop Dependences

This analysis pass annotates each internal λ node with the locations the node either demands or mutates. When the λ represents a loop, locations that are both demanded and mutated by the loop are called *loop dependent locations*. In the `do-while` loop of `example` (Figure 2), variable `a` is both demanded and changed, `b` is demanded but not changed, and `epr` is changed but not demanded.

This pass employs a fixed-point finding algorithm to determine this information. In a particular strongly connected component (SCC) of the VDG each λ demands and changes the same locations. Our algorithm performs an inside-out traversal of SCCs by visiting inner SCCs and internal λs first, so that inner loops get more detailed information than outer ones.

A traversal of a λ reveals which locations are demanded and changed by it. This traversal only needs to follow store arguments, because only operations that perform lookups or modifications are of interest, not the values that are retrieved or stored. In Figure 8, only the dotted arcs need be followed, so the traversal is quicker than a full traversal of the VDG.

After all components of an SCC have been traversed, each λ in the SCC is annotated with the demand and modification information, which later transformation passes depend on.

Accounting for the Effects of Calls on the Store

This analysis/transformation is a constant folding pass whose major goal is to make the program as independent of the store as possible by symbolically executing lookup nodes to pull values out of the store. The construction of VDGs left all lookups of the store produced by a Call as nodes in the graph. Likewise, nothing was known about the store parameters of λs, which produced many lookup nodes on an internal λ's store parameter. This pass uses the information found by loop dependence analysis to approximate the stores on entry and exit of each Call.

The effects of Calls are accounted for in a single top-down pass through the VDG. A node is processed only after all its inputs have been processed. Processing propagates a symbolic store, much like the one used during symbolic execution, through the graph. Processing a Call "blanks" out those locations mutated by the λ being called and leaves the others unchanged. Processing a lookup node attempts a lookup on its incoming symbolic store. If the lookup is successful, then the lookup node is replaced by what was found; otherwise, it remains.

An internal λ node is processed after its (single) external Call is processed. Its incoming store is approximated by the symbolic store at the Call, less the loop dependent locations of the λ. Many lookups are symbolically executed, and thereby eliminated. For example, for the VDG for `example` (Figure 8), this pass replaces the lookups on `b`, `c`, and `d` within the internal λ with direct pointers to the inputs of the procedure.

Input Arity Raising

Lookups can be eliminated in λ bodies by passing the desired values directly, instead of (or in addition to) passing a store parameter. This is profitable if it releases the caller from the obligation to update the store before the call.

Every λ node takes a store as an argument and produces a store as a result; internal λ nodes have no other arguments or results. The store argument is too coarse; only the locations whose contents are both demanded (*i.e.*, used before being changed) and modified within the λ (*i.e.*, its loop dependent locations) need be its arguments. All λ invocations are sequentialized, which inhibits transformations, especially code motion. (Aliased variables, the requirement not to reorder output, and similar constraints may introduce dependences that force such sequentialization.)

Arity raising of the store argument changes the contract between λs and their Calls. Those lookup nodes within a λ that operate on its incoming store can be replaced by arranging for the Calls of the λ to provide those values as arguments by executing the lookup prior to calling the λ. For example, arity raising is how the lookup of `a` within the internal λ of `example` is replaced by a new formal parameter. (The store parameter node is then eliminated by dead code removal because after this transformation, it is not accessed within the internal λ.)

A more aggressive input arity raiser could use a fixed point algorithm to find even more constant values. However, it may be cheaper to discover these additional constants after the input arity raising pass.

Output Arity Raising

Just as an internal λ can have its input arity increased, so can its output arity be increased. Originally, each λ returns a store. External lookup operations then retrieve values from the store. We can move these lookup operations into the λ by having the λ return the value of the lookup as as additional return value that is used instead of the original lookup. For example, this transformation is applied to the `epr` lookup of the Call of the internal λ in (Figure 8), causing that value to be returned by the internal λ. (The store return node is then eliminated by dead code removal since, after this transformation, it is dead, *i.e.*, none of the λ's Calls have any consumers for their store result ports).

Dead Update Elimination

Due to arity raising and symbolically evaluated lookups, there will be many dead update operations on stores. Dead update elimination removes these now extraneous operations. First, a bottom-up traversal determines at each store mutating site (*e.g.*, update nodes and Calls) those locations whose contents are demanded (*i.e.*, may affect the computation's final value) after the store mutator. (This is also called live variable analysis.) If the locations that are changed by the operation are not demanded by the rest of the computation, then the operation is deleted, and its consumers are changed to accept the store that was the input to the store mutator.

It would seem that more direct methods could be used for transforming the VDG, in particular, for determining the values demanded and modified by loops. Other sparse representations, such as dependence flow graphs [PBJ+90, JP93] and SSA form, are constructed directly from the CFG. They can collect in one pass the variables used or modified within a loop. We do not do so because we are actively extending the system to handle data structures and pointers, which requires a uniform and incremental method for determining the locations used or modified within a loop.

Incremental Transformations

Because the VDG models (only) data dependences explicitly, transformations occur without the classical problem of data flow and control flow information getting out of sync and without the need for a global analysis to put them back in sync. For example, consider selector deletion, which occurs when static analysis determines the value of the predicate of a selector (γ) node. We snip out all γ nodes with that predicate

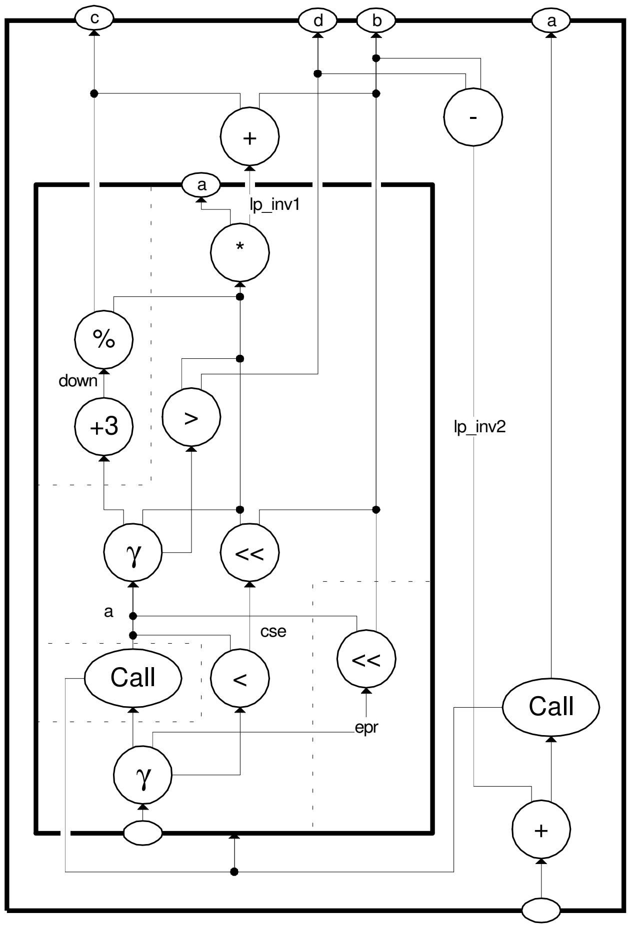

Figure 9: The VDG for `example` (Figure 2) after the VDG transformations of Section 3. Dotted lines separate regions of equivalent demand dependence; there are no store values.

by directly connecting their outputs to the relevant inputs and transitively rebuilding successor nodes as necessary (which performs common subexpression elimination, constant folding, etc.). We safely ignore the issue of whether some variable was defined on the deleted selector, or some definition was allowed to pass through on the deleted selector, as all the dataflow ramifications are directly accounted for in the VDG. Nodes that were demand dependent solely on the predicate (or its negation, whichever is false) are garbage collected without the need to employ a separate dead code elimination pass. After local changes propagate, a scan determines if any enclosing λ formals may now be removed, or if return values are no longer computed within the λ. If so, the relevant Calls are updated, with local transformations proceeding outward. The same effect can be achieved with PDG methods [FOW87, Section 5.1], but at higher cost in conceptual complexity, overhead, and analysis passes.

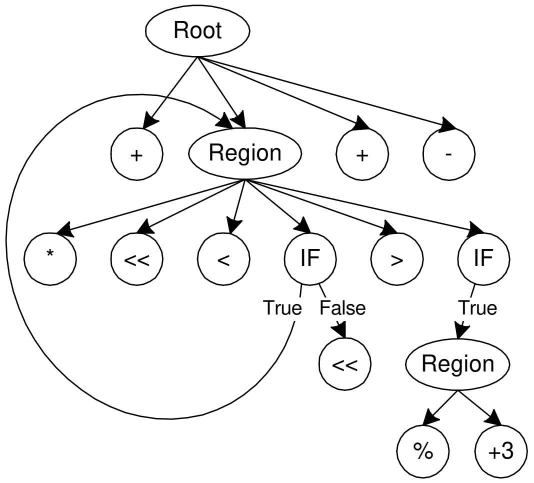

Figure 10: The demand dependence graph constructed from the VDG of `example` (Figure 9).

4 Code Generation

Generating code from the VDG is done in two stages. The first stage transforms the VDG into a *demand-based program dependence graph* (dPDG). A standard PDG [FOW87] consists of a data dependence graph and a control dependence graph (CDG). In the dPDG the VDG operand arcs provide the data dependences and the CDG is replaced by a *demand dependence graph*.

The second stage transforms the dPDG into a CFG from which code may be generated directly. This stage is described in [Ste93a], which extends earlier work on sequentializing PDGs [SF93].

The demand dependence of a VDG node is characterized by the γ nodes encountered on paths from a return node, much as the control dependence of a CFG node is characterized by branch nodes encountered on paths from an entry node. That is, any path from a return node to a VDG node yields a sequence of γ selector ports and a corresponding sequence of selector/predicate values required in order that the given node actually be demanded along that path. Very loosely, node n is demand dependent upon γ node g if n dominates, within the VDG, one arm (either True or False) of g, but not the other. Space prevents us from giving the formal definitions of demand dependence and VDG dominance.

In `example`'s final VDG (Figure 9), the "`%`" and "`+3`" nodes have demand dependence "`a > d`", the recursive Call node has demand dependence "`a < cse`", the "`a << b`" node has demand dependence "not `a < cse`", and the remaining nodes are always demanded.[5] Figure 10 shows the demand dependence graph for `example`.

Viewed as predicates, the demand dependence for a

[5] Since the VDG contains no names, we should really refer to the result port of the "`>`" node rather than to some particular expression like "`a > d`".

computation implies the control dependence: in the original CFG, a computation may be computed in a program before it is used and the use may occur at a program point that doesn't post-dominate the computation. For example, the control and demand dependences of the expression `down = a % c` in `example` differ. The control dependence is vacuous (*i.e.*, "always execute"), but the demand dependence is "`a > d`". The corresponding code motion into an arm of a conditional is called the revival transformation in [FKCX94].

The use of the dPDG enables more code motion and other transformations than representations based on control dependences. Motion out of loops occurs automatically, since the dPDG reflects only when a computation is used, not its location in the original program. When the data dependences do not enforce a total ordering, the code generator is free to reorder computations and conditional constructs. For example, our system performs more strict (nonspeculative) code motion than [CLZ86]. In particular, for Figure 2 of that paper, our system would move the `a.1` and `a.5` assignments into the else clause of the `if movable1` statement on the right side of the figure,[6] resulting in a clear improvement in the code. We extend the results of [CLZ86] in other ways: because of our redundancy detection, we find at least as many (structural) common subexpressions as their COMMON algorithm finds, and statements moved by our method don't have to have the same values on all iterations (our method handles the problem case of [CLZ86, Figure 11]).

5 Elimination of Partial Redundancies

A computation is *partially redundant* at a program point if there is some path to the point that performed the same computation (the redundant path) and some path to the program point where the computation is not performed. For example, the second `<<` operation of `example` (Figure 2) is partially redundant. We will show how to remove partially redundant computations from a program via transformation.

Consider an operation node at least one of whose operands is a γ node. The operation can be distributed through the γ node, without changing the VDG's semantics, by constructing two new operation nodes (Figure 11). This transformation is equivalent to converting `a op (p ? b : c)` into `p ? a op b : a op c`, which is transformation rule 3.2.a of [SHKN76]. Our contribution is a program representation that allows this simple transformation to be the basis of efficient EPR.

[6] We assume that "unmovable" expressions are unmovable due to loop carried dependences, and not because of semantic issues such as volatile variables in C.

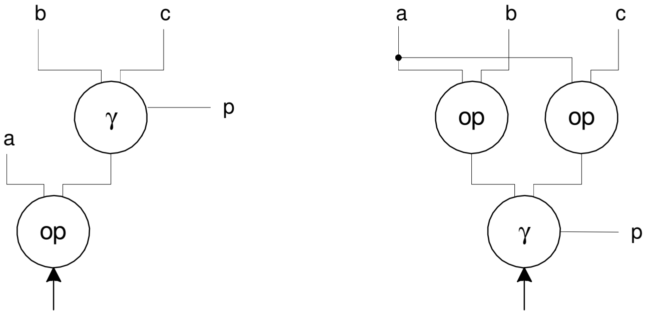

Figure 11: Distributing an operation through a γ node.

Distribution through γs duplicates code and increases the size of the program (though not the number of operations performed at runtime). It removes redundant computations (*i.e.*, is profitable) if either of the new operation nodes matches (has the same operator and operands as) a node that it *subsumes*, or if further applications of the transform match such a node. Node n *subsumes* node m iff n's demand dependence implies m's demand dependence (*i.e.*, if n is demanded, then so is m.) In other words, if n subsumes m, then the computation of m will be available (or be costlessly made available) at the computation of n. Our EPR algorithm performs only profitable transformations, searching for a subsumed match on a candidate node's γ tree input(s). When the inputs are γ nodes with the same predicate, minor bookkeeping ensures that only possible program paths are considered during the search and transformation.

Assume that hashing (*i.e.*, matching two nodes, or determining whether a predicate has been encountered and its value) is a constant time operation and that the subsumption relation can be verified in time linear in the depth of the demand dependence graph, $|\mathrm{DDG}|$. Let $\mathit{Gtree}(n)$ denote the size of the γ tree rooted at node n. A candidate node with γ tree input g is processed in worst case time $O(|\mathrm{DDG}| \cdot \mathit{Gtree}(g))$, which is equivalent to being quadratic in the size of the procedure being optimized. If both of the node's inputs are γ trees, the time is no worse than cubic. However, empirical results [CFR+89] indicate that in practice $\mathit{Gtree}(g)$ contains at most a few elements, which implies that in practice the depth of the demand dependence graph has a small bound.

Because our EPR method employs only a partial ordering of the computation as expressed by the dPDG, it is much simpler than methods that employ total orderings expressed as a CFG. For example, as compared to the pure CFG methods [MR79, Dha91, DRZ92, KRS92, DS93], it requires no availability or global (partial) anticipatibility analysis for every expression nor any basic block by basic block code motion. Like more recent CFG based EPR methods, it does not perform redundant code motion. It is also simpler than the SSA method [RWZ88] with its CFG graph conditioning (*e.g.*, adding landing pads), assignment of ranks or orders, LCT and MCT tables, etc.

6 Related Research

Our graph structure is a direct descendant of the graph structure used in the Fuse partial evaluator [Wei90, WCRS91]. The graph structure was designed for reasoning about and transforming strict functional programs. The only accommodation made for imperative programs is the explicit presence of a store datatype (and other usually implicit machine quantities), and operations on stores. Our transformational engine is an extension of Fuse's, which, because Fuse was a specializer, only did constant folding and function specialization. Because we now use graphs to reason about imperative programs with unstructured control, the translation from the source to graph form is more complex than it was in Fuse.

PDG-based compilers simplify transformations by eliminating the CFG, at the cost of needing to reconstruct the CFG to produce serial code [FM85, FMS88, SAF90, SF93, Ste93a]. The underlying representation is still statement-oriented, however (control dependences are attached to statements) so the PDG approach does not enable as many analysis and transformation simplifications as the VDG and the dPDG, which also eliminates the CFG but attaches demand dependences to expressions.

The program dependence graph (PDG) [FOW87] consists of a control dependence graph (CDG) and a data dependence graph. The CDG is the novel contribution; it ties together elements of the program that execute under the same control conditions. CDGs provide information that enables and simplifies many transformations, such as code motion [CLZ86].

The theta graph [Cli93a, Cli93b] is a PDG for a program in SSA form [CFR$^+$89]. The theta graph is built directly from the CFG, without any need for an inlining step as in the VDG construction. Click suggests removing the control information from his program representation; if this occurs, the theta graph will probably end up very similar to the VDG.

The dependence flow graph (DFG) [PBJ$^+$90, JP93] is an extension of the standard SSA form in which, in addition to the ϕ nodes inserted at merge points, switch nodes are inserted above branch points. Switches are merge points for backwards program execution, so the DFG makes it as easy to perform backwards as forwards analyses.

MIT dataflow graphs [AN90] differ from VDGs by including tokens for control and by not needing to represent machine state, such as stores. [BJP91] suggests a dataflow-like intermediate representation that shares many of our goals and concerns. However, it uses tokens to represent control, which keeps explicit control in the program representation. The authors of [BJP91] have since abandoned the dataflow model and are now investigating DFGs [JP93].

The gated single assignment (GSA) component of the program dependence web [BMO90, CKB93] is similar to the VDG, but transformation is hindered by the need to keep three different representations in sync, and it fails to handle irreducible programs. Thinned gated single assignment (TGSA) form [Hav93] is also similar to our representation. The major difference between the VDG and TGSA form is that VDGs represent looping via procedure call and return, whereas TGSA, like PDW form, represents looping with special nodes (ν and μ).

[Fie92] presents a framework for reasoning about and partially evaluating programs in graphical form, based on the idea of a *guarded expression*. Its rewrite rules are similar to VDG transformations, and similar techniques are used to extract values from stores, converting lookup operations into the values they would return. This work is similar to [CF89], which discusses the semantics of the program dependence graph (PDG) when considered as an executable dataflow program.

Sparse evaluation graphs [CCF91] are a program representation for efficient solution of dataflow problems. For a given dataflow problem, a program is converted to a sparse evaluation graph which is used as the data structure for analysis. VDGs can be translated into a similar data structure by removing nodes whose abstract function is the identity function, removing γ nodes whose true and false inputs are identical, and eliminating (internal) λ and Call parameters that no longer represent loop dependences. One advantage of constructing sparse evaluation graphs from VDGs is the presence of γ nodes, which allow for conditional analyses (*e.g.*, conditional constant propagation).

7 Future Work

We expect the VDG representation to simplify pointer and alias analysis. We approximate the locations to which a pointer may refer by explicitly modeling locations as the operation that allocates them. We use "mini-stores," which represent portions of the global store that are not aliased to other locations but within which aliasing may occur, in order to minimize false dependences and maximize parallelism. We eliminate not only the names that values may be bound to, but the locations that those names really denote.

Our system will give two expressions the same global value number even when those expressions have differing demand dependences. When this occurs, there may be no program point where the value can be computed without unnecessarily increasing register pressure. The system will be forced to duplicate the computation, which will be no worse than the original program, in which the computation appeared multiply. To help the system decide whether a computation with multiple uses appeared duplicated in the original program, we plan to keep track of the control dependences of each expression, and combine them when global value numbering merges computations. We anticipate that we can construct control dependences on the fly during symbolic execution and store propagation.

γ nodes with the same predicate are either all produced from the same `if` statement in the program, or from different `if` statements that used the same boolean value as their predicate. Consider the case where they come from the same `if` statement. When they all share the same demand dependence, the code generator produces only one branch operation. However, when they have different demand dependences, then the code generator may decide to place them in different parts of the program, which will require saving the test value, and replicating the `if` statement throughout the program. Doing so may or may not improve the program. We need to address this issue in the back end.

Our representation raises many other issues for a code generator. For example, a computation that is demanded at two sites in the program with different demand dependences may be duplicated to reduce register pressure [BCT92]. Similarly, γ nodes with similar predicates but dissimilar demand dependences force a "precompute vs. register pressure vs. code size" engineering tradeoff. While these issues are not trivial, they are where they belong, in the back end.

Program slicing [Wei84, Ven91] determines which elements of a program affect, or can be affected by, a given computation. This analysis can be useful in debugging and in distributing computations across processors. The VDG is a particularly convenient representation for slicing because it is sparse (operands are connected directly to inputs) and all computation is expressed as value flow; a VDG is effectively a representation of the slice for every computation in the program, simultaneously. Other authors have noticed the efficacy of the PDG for slicing [HRB90] and addressed the problems of irreducible control flow [BH92]. Our algorithms are equally as precise as the best of these, simpler to state, and often more efficient [Ern93]. Because the VDG represents only value flow, we are able to slice only on values, not on particular program points, which notion does not make sense in the VDG framework. This does not appear to be a serious limitation.

Acknowledgments

We thank Ellen Spertus, Erik Ruf, Cliff Click, Todd Knoblock, and the referees for their helpful comments.

References

[AN90] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Computer Science Series. Addison-Wesley, Reading, Massachusetts, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11. ACM Press, January 1988.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 311–321. ACM Press, June 1992.

[BH92] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. Technical Report 1128, University of Wisconsin – Madison, December 21, 1992.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271. ACM Press, June 1990.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Procedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66. ACM Press, January 1991.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 13–27, Portland, OR, June 1989.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35. ACM Press, January 1989.

[CKB93] Philip L. Campbell, Ksheerabdhi Krishna, and Robert A. Ballance. Refining and defining the program dependence web. Technical Report CS93-6, University of New Mexico, Albuquerque, March 1993.

[Cli93a] Cliff Click. Combining analyses, combining optimizations. PhD thesis proposal, April 1993.

[Cli93b] Cliff Click. From quads to graphs: An intermediate representation's journey. Submitted for publication, October 18, 1993.

[CLZ86] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, January 1986.

[Coc70] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970.

[CS70] John Cocke and Jacob T. Schwartz. Programming languages and their compilers. Technical report, Courant Institute, NYU, April 1970. Preliminary notes.

[Dha91] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[DRZ92] Dhananjay M. Dhamdere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, San Francisco, California, June 17-19, 1992. ACM Press.

[DS93] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's *lazy code motion*. *ACM SIGPLAN Notices*, 28(5):29–38, May 1993.

[Ern93] Michael Ernst. Program slicing using the value dependence graph. In preparation, 1993.

[Fie92] John Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis (preliminary report). In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, June 1992. Published as Yale University Technical Report YALEU/DCS/RR–909.

[FKCX94] Lawrence Feigen, David Klappholz, Robert Casazza, and Xing Xue. The revival transformation. In *Proceedings of the Twenty-first Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, January 1994.

[FM85] Jeanne Ferrante and Mary Mace. On linearizing parallel code. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 179–190, January 1985.

[FMS88] Jeanne Ferrante, Mary Mace, and Barbara Simons. Generating sequential code from parallel code. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 582–592, June 1988.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Har85] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. Draft — Private distribution, February 20, 1993.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, Albuquerque, NM, June 23-25, 1993. ACM Press.

[JPP93] Richard Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical Report CTC93TR141, Cornell University, July 1993.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, San Francisco, California, June 17-19, 1992. ACM Press.

[MR79] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[Ott78] Karl Joseph Ottenstein. *Data-flow graphs as an intermediate program form.* PhD thesis, Purdue University, August 1978.

[PBJ+90] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. Technical Report 90-1152, Department of Computer Science, Cornell University, Ithaca, NY 14853, September 1990.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM Press, January 1988.

[SAF90] Barbara Simons, David Alpern, and Jeanne Ferrante. A foundation for sequentializing parallel code — extended abstract. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 350–359, 1990.

[SF93] Barbara Simons and Jeanne Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR 03.465, IBM, Santa Teresa Laboratory, San Jose, California, February 1993.

[SHKN76] T. A. Standish, D. C. Harriman, D. F. Kilber, and J. M. Neighbors. The Irvine program transformation catalogue. Technical Report 161, University of California at Irvine, Department of Information and Computer Science, January 1976.

[Ste93a] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, WA, August 1993.

[Ste93b] Bjarne Steensgaard. A store algebra for graphical program representations. In preparation, November 1993.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, Toronto, Ontario, Canada, June 26-28, 1991.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 165–191, Cambridge, MA, August 26-30, 1991. ACM, Springer-Verlag.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wei90] Daniel Weise. Graphs as an intermediate representation for partial evaluation. Technical Report CSL-TR-90-421, Stanford Computer Systems Laboratory, Stanford, CA, March 1990.

[WZ85] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with condition branches. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.

[WZ89] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. Technical Report CS-89-36, IBM T.J. Watson Research Center, Yorktown Heights, NY, May 1989.