A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies

Craig Chambers, Jeffrey Dean and David Grove

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195 USA

Technical Report 94-09-07
September 1994

{chambers,jdean,grove}@cs.washington.edu
(206) 685-2094; fax: (206) 543-2969

A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies

Craig Chambers, Jeffrey Dean, and David Grove

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195
(206) 685-2094; fax: (206) 543-2969
{chambers,jdean,grove}@cs.washington.edu

Technical Report 94-09-07

Abstract

Compilers and other programming environment tools derive information from the source code of programs; derived information includes compiled code, interprocedural summary information, and call graph views. If the source program changes, the derived information needs to be updated. We present a simple framework for maintaining intermodule dependencies, embodying different tradeoffs in terms of space usage, speed of processing, and selectivity of invalidation, that eases the implementation of incremental update of derived information. Our framework augments a directed acyclic graph representation of dependencies with *factoring nodes* (to save space) and *filtering nodes* (to increase selectivity), and it includes an algorithm for efficient invalidation processing. We show how several schemes for selective recompilation, such as smart recompilation, filter sets for interprocedural summary information, and dependencies for whole-program optimization of object-oriented languages, map naturally onto our framework. For this latter application, by exploiting the facilities of our framework, we are able to reduce the number of lines of source code recompiled by a factor of seven over a header file-based scheme, and by a factor of two over the previous state-of-the-art selective dependency mechanism without consuming additional space.

1 Introduction

Programming environment tools typically derive information from the source code of programs. For example, a call graph browser examines the procedures in the program and derives a call graph from the modules. As another example, a compiler derives compiled object code from the program's source code. Modern optimizing compilers often derive additional kinds of information from the program, such as interprocedural summary information, to support optimizations.

When using such tools in an interactive programming environment striving for fast turnaround after programming changes, the tools need to be able to keep the information they derive up to date as parts of the program's source code changes. If the derived information is expensive to regenerate from scratch, as is compiled code and some kinds of interprocedural summary information, then the system must be able to selectively regenerate only the out-of-date derived information. To provide this incrementality, some sort of dependency mechanism that links source modules (or pieces of modules) to information derived from the source modules is needed.

Ideally, the dependency mechanism should be *concise* (use little storage space) and *speedy* (take little time to update the derived information). Speediness is dependent on *rapid* and *selective* calculation of out-of-date derived information: we wish a dependency mechanism that quickly identifies the information that needs to be recomputed and does not recompute more than necessary. Depending on the kind of derived information and the speed of recalculation, different tradeoffs can be made among conciseness, rapid invalidation, and selectiveness when engineering a cost-effective dependency structure. For example, for a

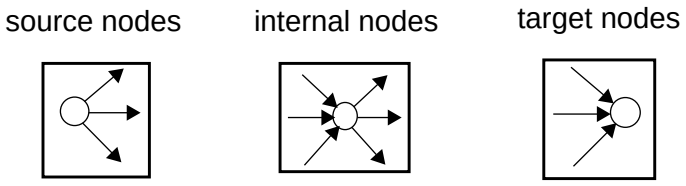
compiler, a coarse-grained dependency structure can be maintained, say at the file-level granularity as in UNIX `make` [Feldman 79], which would be space-efficient but not very selective. Alternatively, a finer-grained dependency structure can be maintained that is selective but space-consuming. Or a medium-grained dependency structure augmented with procedural checks can be maintained that can be both very selective and moderately space efficient, at a cost in time to identify out-of-date compiled code.

In this paper we present a simple framework for representing and processing intermodule dependencies modeled on a dependency graph. Our framework supports several kinds of dependency nodes, including *filtering nodes* which check whether a change actually affects some derived information and *factoring nodes* which save space in the graph representation. We present a simple algorithm to traverse, in topological order, the minimum number of nodes in the graph when calculating what derived information is invalid. We show how to automatically introduce factoring nodes to reduce the number of edges in the dependency graph and thereby save space. Using our framework and algorithms, system builders can implement incremental invalidation mechanisms fairly easily that embody the desired tradeoffs among conciseness, precision, and speed. Section 2 describes our framework in detail.

In section 3, we show how our dependency framework can be applied to achieve selective recompilation of object-oriented languages in the face of interprocedural analyses and optimizations. We use a trace of a month's worth of program development to compare several schemes for selective recompilation, including two header-file-based schemes, the previous best selective dependency framework, and two variations of our dependency mechanism. The results indicate that our dependency system, exploiting the facilities of our framework, is seven times more selective than the header-file-based systems, and is twice as selective as the previous best fine-grained dependency mechanism with no additional space requirements.

2 Dependency Framework

In our framework, intermodule dependencies are represented by a directed, acyclic graph (dag) structure. Nodes in this graph represent information, and an edge from one node to another indicates that the information represented by the target node is derived from or depends on the information represented by the source node. Depending on the number of incoming and outgoing edges, we classify nodes into three categories:



- *Source nodes* have only outgoing dependency edges. They represent information present in the source modules comprising the program, such as the source code of procedures and the class inheritance hierarchy.
- *Target nodes* have only incoming dependency edges. They represent information that is an end product of some tool, such as compiled code.
- *Internal nodes* have both incoming and outgoing edges. They represent information that is derived from some earlier information and used in the derivation of some later information. Interprocedural summary information and the procedure call graph are examples of information that could be represented by internal nodes. The information represented by an internal node may be both an end product (such as a call graph for a call graph browser) and an input for other information (such as a call graph used during interprocedural optimizations).

The graph is constructed incrementally as information is computed. Whenever a process uses a piece of information that might change, it adds an edge to the dependency graph from the node representing the used information to the node representing the client. In our implementation, there is a notion of the "current dependence node" representing the active client. Whenever a piece of information is queried that might change, the provider of the information automatically adds a link from its representative node to the "current dependence node." In this manner, construction of the dependency graph is largely transparent to client code.

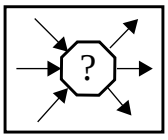
Invalidation proceeds by propagating `invalidate` messages through the dependency graph. Some tool, such as an editor or a `make`-like tool, detects changes in parts of the source program. The corresponding source nodes are sent the `invalidate` message. In the simplest case, when a node receives an `invalidate` message, it updates the information it represents and then resends `invalidate` to each of its successors (if any) in the dependency graph. In this manner, all information derived from out-of-date source code will be updated.

Our framework includes several refinements to this simple scheme, including *filtering nodes* that incorporate tests into the invalidation protocol, an algorithm for graph traversals that visits the minimum set of nodes exactly once, and an algorithm for introducing *factoring nodes* into the dependency graph to conserve space. The remainder of this section discusses these three refinements and illustrates how our framework can model several existing selective recompilation facilities.

2.1 Filtering Nodes

When a node receives an invalidation message, some information on which it depended has changed. However, that does not always imply that the information representing by the node itself has changed. For example, in an optimizing compiler, one node might represent the program's call graph structure. This node depends on the source code for all procedures in the program. If a procedure's body is changed, its call graph might have changed, but it might not have changed. We wish to avoid assuming that the call graph, and all information derived from the call graph, has changed just because a procedure has been edited, i.e., we wish to be more selective in our invalidations. One method for being more selective is to refine the granularity of the dependence graph to represent more precisely the information that is being depended upon. However, for the example of the program's call graph, it seems difficult to refine the graph structure in such a way that edits to a procedure do not lead to invalidations of the program's call graph unnecessarily. And even if it were possible, the space costs for the more refined dependency structure might be unacceptable.

filtering nodes



Filtering nodes resolve this dilemma and support more selective invalidation by allowing a programmatic test to be performed by a node when it is invalidated. Only if the test succeeds will the node be considered out-of-date. For the call graph example, each source procedure could have a dependent filtering node representing the set of callees of that procedure. The call graph node would be dependent on each of these callee set nodes. If a source procedure changes, invalidation of its dependency node will send the `invalidate` message to the callee set node. This filtering node will first recompute the set of callees from the changed procedure's source code, compare it to the old set, and only consider itself changed if the two sets differ. In this manner, invalidations will propagate only as far as derived information really has changed.

Filtering nodes provide an important new point in the space/invalidation time/recompilation time trade-off. Filtering nodes are a space-efficient alternative to finer-grained dependency structures for providing selective invalidation. For some kinds of dependencies, such as the program call graph example, they appear to be able to provide finer selectivity than can be achieved by any purely data-structure-based dependency mechanism, regardless of granularity. The primary cost of filtering nodes is additional checking time during graph traversal. Additionally, filtering nodes may require supplemental state to be able to evaluate their checks. However, the checking cost for filtering nodes is often "free" in the sense that if downstream nodes were invalidated, they would most likely need to recompute the information represented by filtering nodes anyway; filtering nodes have the effect of shifting some of the recompilation time to graph traversal time, and sometimes eliminating the rest of the recompilation time.

A common use of filtering nodes is to represent caches of information, such as the program call graph, other kinds of interprocedural summary information, and the cached results of queries of the underlying program. To make using our dependency framework more convenient, our implementation includes a user-defined control structure `with_dependence(`*dependence*`,` *action*`)` that switches the "current dependence" to *dependence* (typically the dependence representing the cached information), performs *action* (typically a closure that computes some information and adds it to the cache, accumulating dependency links as it goes), and then restores the current dependence to its original value.

2.2 Traversing the Dependency Graph

When information at the roots of the dependence graph changes, invalidation messages must be propagated from the out-of-date dependence nodes to their downstream dependents. Since we expect only a small part of the dependency graph to be affected by incremental programming changes, we wish the traversal to take time proportional to the size of the affected dependency graph, not the whole dependency graph. In the presence of filtering nodes that can be expensive to visit, we wish the traversal to visit each node at most once, even if the node is reachable along multiple paths. In the presence of nodes that "guard" caches of information, we require that the traversal be in some topological order, so that information guarded by a node is updated before any clients of that information examine it. Together, these requirements imply the need for an algorithm that examines the minimum set of nodes during the traversal, visiting each node exactly once and in topological order. Computing a topological order of the dependency graph and then processing invalidations in this order does not meet this need, since it examines the entire dependency graph to compute the topological order. And computing the affected subgraph and then sorting topologically does not work, since in the presence of filtering nodes the affected subgraph cannot be identified until the traversal is complete. Finally, in many environments, a global fixed topological ordering cannot be precomputed, since the dependency graph is dynamic: as source and derived information changes, the dependency patterns change.

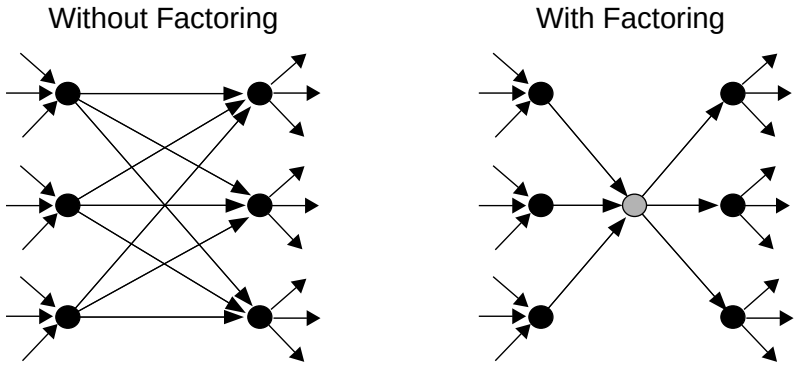
In our framework, we maintain a topological ordering of the nodes in the dependency graph incrementally. Each dependency node is assigned a number, under the constraint that a node has a number greater than all its predecessors. To traverse the graph, when a node is sent the `invalidate` message it is added to a priority queue, with nodes ordered by their assigned number. The traversal occurs in topological order by removing a minimally-numbered node from the queue, processing its invalidation (and possibly adding successors to the priority queue as a result), and repeating until the queue is empty. This algorithm is guaranteed to visit the minimum set of nodes, each node exactly once, in topological order, in time $O(K \cdot N \log N)$, where N is the number of nodes visited by the traversal and K is the maximum time to evaluate the test of any filtering node traversed.

The correctness of the traversal algorithm relies on being able to assign numbers to dependency nodes that induces a legal topological ordering on the nodes. In some kinds of dependency nodes, it is possible to assign a fixed number to each dependency node when it is created. For nodes that are always source nodes, with no predecessors, the number 0 can be assigned statically. For nodes that are always targets, with node successors, a number larger than the maximum depth of the dependency graph can be assigned. Even internal nodes can sometimes be assigned fixed numbers, if the structure of the dependency graph follows a regular pattern. For example, if the callee set dependence node is always dependent only on the procedure source node, it can be assigned the number 1 (one greater than the number assigned to the procedure source node, 0), and similarly the call graph dependence node can be assigned the number 2.

In other situations, nodes cannot be assigned fixed numbers when created. In this case, a simple adaptive strategy can be used. A node *n* is initially assigned the number 0. Whenever the node receives a new predecessor *p*, the node's assigned number *n.num* is compared against the predecessor node's number *p.num*. If $p.num \geq n.num$, then *n.num* is changed to $p.num + 1$, and all the successors of *n* are visited recursively to ensure that they have numbers greater than the new *n.num*. In the worst case, this algorithm could visit all nodes downstream of *n*, which could be of size $O(N)$ where N is the number of nodes in the dependency graph. A similar algorithm for incremental maintenance of a topological ordering was described by Marchetti-Spaccamela, Nanni, and Rohnert [MS et al. 93].

2.3 Factoring Nodes

Fine-grained dependencies can support more precise invalidation than coarser-grained dependency representations. However, a naive representation of fine-grained dependencies can consume expansive amounts of storage. To reduce the size of the dependence graph, our framework provides the notion of factoring. Factoring is a transformation that introduces intermediary *factoring nodes* in complete bipartite regions of the dependency graph, as illustrated by the following figure:



Opportunities for factoring are likely to be fairly common in most environments. For example, many compiled modules are likely to depend on similar sets of standard library routines. Factoring can replace a plethora of edges linking each library routine to many compiled modules with a few factoring nodes that "bundle together" a group of dependencies that often occur as a group. Factoring slows down invalidation processing somewhat, however, by increasing the depth in the dependence graph. Factoring also increases the number of nodes in the graph, so a significant reduction in the number of edges is required to pay back the increased space cost for the additional factoring nodes.

Factoring nodes could be introduced into the dependence graph manually. Some kinds of internal nodes serve as natural factoring points. Even better, however, would be an automatic mechanism for determining where factoring is possible and for modifying the graph as edges are inserted to keep the graph maximally factored at all times. We have developed an algorithm that detects and factors complete bipartite subgraphs

when factorization reduces the number of edges in the graph. Figure 1 illustrates the behavior of our algorithm for several edge insertions, with inserted edges shown as dotted arrows and affected edges shown as bold arrows.

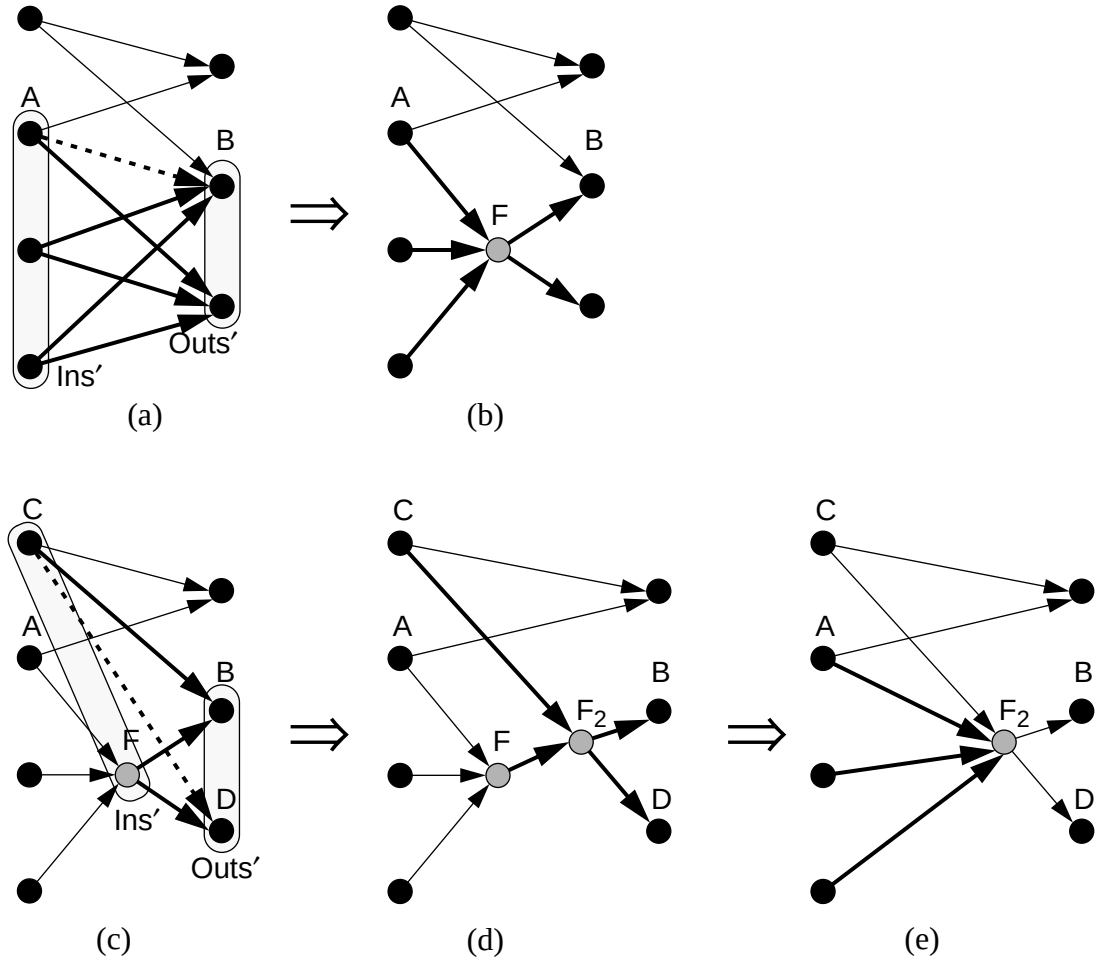


Figure 1: A sequence of factoring transformations

In figure 1(a), the edge (A,B) has been inserted, creating a 3×2 complete bipartite subgraph. A factoring node F is inserted, as shown in figure 1(b). Figure 1(c) shows an edge being added between C and D, creating a 2×2 subgraph between C, F, B and D. Normally, inserting a factoring node in a 2×2 subgraph is not useful, since the resulting graph contains the same number of edges and one additional node (F_2, as shown in figure 1(d)). However, if, after factoring, the new factoring node is the only predecessor or successor of an existing factoring node, then these two factoring nodes can be merged, as shown in figure 1(e). The algorithm for performing these transformations is shown in figure 2. The algorithm runs in $O(e^2)$ time per edge inserted (including factoring edges), where e is the maximum number of edges entering or exiting a node.

Newbery describes a related algorithm to factor bipartite graphs, with the goal of reducing the number of edge crossings when the graph is displayed [Newbery 89]. The algorithm differs from ours because it is not

Let *ins*(*n*) and *outs*(*n*) be the predecessors and successors, respectively, of node *n*

-- Compute sets ins' and outs', the vertices of a complete bipartite subgraph that is amenable to factoring.
-- Factor located subgraph if large enough
AddEdge(*A*,*B*) ≡
    ... add edge (A,B) *to dependency graph.*
    -- Pick element from outs(A) different from B and look for large predecessor set common to o and B
    foreach $o \in outs(A), o \neq B$
        -- Compute set of predecessors o and B have in common
        $ins' := \{ i \in ins(B) \mid i \in ins(o)$ **and** $i \in ins(B) \}$
        if $|ins'| \geq 3$ **or** $|ins'| \geq 2$ **and** $\exists f \in ins'$ s.t. *f* is a factoring node **then**
            -- Found a graph that will save space with factoring: make outs' as large as possible, given ins'
            $outs' := \{ o_2 \in outs(A) \mid \forall i_2 \in ins', o_2 \in outs(i_2) \}$
            InsertFactoringNode(*ins'*,*outs'*)
            return
        endif
    endfor
    ... symmetric code to pick element from ins(B) and look for large common successor set ....
    ... otherwise, no factoring candidates found...
    return

-- Replace edges between V_{ins} *and* V_{outs} *with a factoring node F and edges from* V_{ins} *to F and F to* V_{outs}
InsertFactoringNode(V_{ins},V_{outs}) ≡
    ... delete edges between V_{ins} *and* V_{outs} *...*
    ... create factoring node F, and add edges from V_{ins} *to F and from F to* V_{outs} *...*
    if *any of the newly added edges are between F and another factoring node* F_2 **and**
      that edge is the only edge leaving or entering F_2 *in that direction* **then**
        merge F and F_2 *by redirecting* F_2*'s edges to F and deleting* F_2
    endif
    .. call **FactorEdge** *recursively for each edge added to the graph when inserting F ...*

Figure 2: Factoring Algorithm

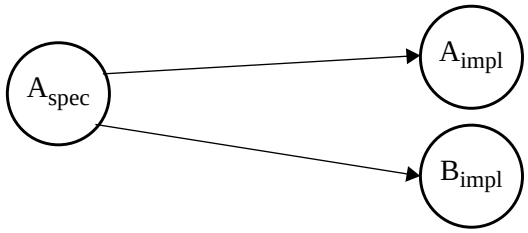
incremental and depends on having a more rigid structure in order to factor the graph. Our algorithm can also introduce multiple levels of factoring nodes automatically.

2.4 Examples

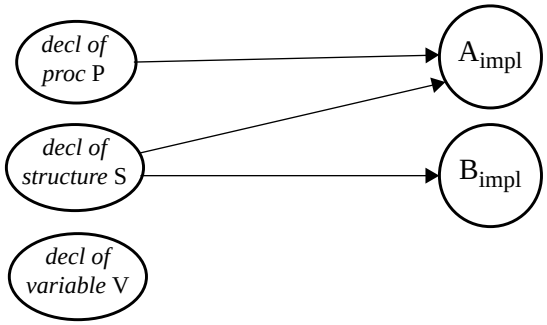
Our framework can be utilized to implement existing solutions to selective compilation. Selective recompilation attempts to minimize the amount of code that must be recompiled to maintain the correctness of the generated code with respect to a program's source code. Adams *et al.* provides an overview of several techniques and a quantitative comparison of their effectiveness in reducing compilation time [Adams et al. 94]. We illustrate how our framework can be applied to model several of these strategies, using as an example a specification for a module A (A_{spec}) that declares a procedure P, a structure S, and a global variable V, and implementations for modules A (A_{impl}) and B (B_{impl}) that both import A_{spec}.

The simplest selective recompilation rule, sometimes referred to as *cascading recompilation* and exemplified by the UNIX `make` utility, is that whenever a module specification changes, all module implementations that import that specification must be recompiled. Under cascading recompilation, both

A_{impl} and B_{impl} would be recompiled whenever A_{spec} changes, a situation modeled with the following simple dependency graph:



Smart recompilation [Tichy & Baker 85] improves selectivity by maintaining dependencies on individual declarations within a specification, rather than on whole specifications.* Since modules frequently utilize only a subset of the declarations in an imported specification, recompilation is avoided when declarations outside of this subset are changed. Suppose that module A uses both procedure P and structure S, but that module B uses only structure S. A reasonable application of our framework would construct a fine-grained dependency structure, where root nodes are associated with individual declarations in a specification module:

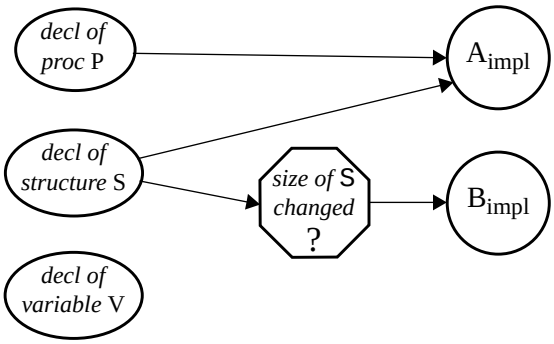


This framework would require more space to represent than the basic framework for cascading recompilation, but it supports much better selectivity. Moreover, since it is likely that many implementations will depend on the same groups of declarations, automatic factoring nodes inserted by our framework can help reduce the space cost for the fine-grained dependency graph.

Attribute recompilation [Dausmann 85] extends smart recompilation to allow dependencies to depend on particular attributes of declarations, rather than depending on the whole declaration. For example, suppose B_{impl} depends only on the size of structure S, not on the individual elements of the structure. This can be

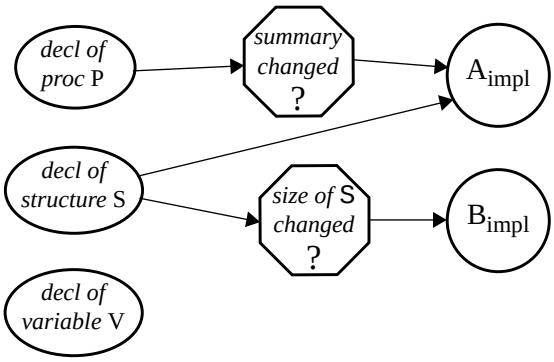
* In *smartest recompilation* [Shao & Appel 93], modules are compiled independently to avoid creating intermodule dependencies, and so smartest recompilation has no need of an intermodule dependency mechanism.

modelled in our framework with a filtering dependence that propagates invalidations only when some attribute of the declaration changes, in this case when the size of structure S changes:



Using our dependency framework, these dependency mechanisms become simple to express and implement.

Compilers that perform interprocedural optimizations generate quite subtle intermodule compilation dependencies. Examples of such optimizations include cross-module inlining and using the results of interprocedural summary information, such as available constants or aliasing relationships, from one module to justify optimizations performed in another module. Burke and Torczon describe several schemes for performing selective recompilation in the face of common interprocedural optimizations for procedural languages [Burke & Torczon 93]. Under their compilation model, each procedure has several associated interprocedural summary sets. The simplest form of selective recompilation is to recompile a procedure whenever the body of the procedure changes or when an interprocedural set used during the compilation of the procedure changes. This can easily be modelled by creating dependency graph edges from interprocedural summary sets to the compiled code for the procedures that used the summary sets. Since the compilation of a procedure P need not depend on the exact contents of the summary set, but often only some property of the summary set, a filtering node mediating between a summary set and a compiled procedure can model more precisely the part of the summary set on which the compilation really depended.



The system described by Burke and Torczon performs a global pass after each programming change in which interprocedural summary information is recomputed from scratch, after which their recompilation tests are used to determine what procedures need to be recompiled as a result of changes in the interprocedural information. The FIAT system improves on this by computing interprocedural summary information lazily [Hall et al. 93]. Using our dependency framework, however, the computation and update of the interprocedural summary information could be made incremental, with the framework only invalidating and recomputing those summary sets that are out of date.

3 Interprocedural Optimization of Object-Oriented Languages

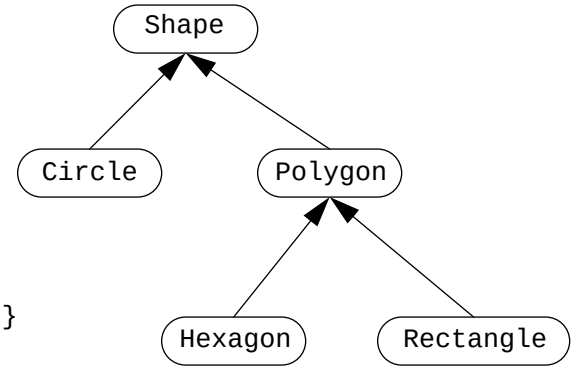
Selective recompilation of object-oriented programs in the face of interprocedural optimizations is difficult to achieve. The kinds of analyses and optimizations performed introduce a large number of intermodule dependencies, and moreover a change to some piece of the source program may or may not actually invalidate all the compiled code that depends on it. We used our framework to develop a highly-selective dependency mechanism for the Cecil object-oriented language [Chambers 93]. This section describes our dependency system and compares its selectivity and space requirements to that of several alternative mechanisms.

3.1 Compile-Time Method Lookup

In object oriented languages, dynamic dispatch (introduced by message sends in Smalltalk [Goldberg & Robson 83] and virtual function calls in C++ [Stroustrup 91]) is often a major source of runtime inefficiency. Effective optimization of programs that use dynamic dispatch heavily requires the elimination of as many of the dispatches as possible. One proven approach applies static class analysis to determine the possible class(es) of the receiver of a message and then performs method lookup at compile-time in an effort to replace the expensive dynamically-bound message send with a cheaper statically-bound procedure call [Chambers & Ungar 91]. Subsequent optimizations such as interprocedural analysis or inlining can further speed the call. These optimizations can improve performance by a factor of 5 to 20 in programs that send messages frequently [Chambers 92]. To perform compile-time method lookup, the compiler examines portions of the class inheritance graph, thereby introducing non-local dependencies between selected attributes of class declarations and pieces of optimized code. To enable incremental recompilation, the compiler must track these relationships between source code and optimized code. For example, consider the following program fragment, written in C++ syntax:

```
class Shape {
   virtual void draw() = 0;
}
class Circle:public Shape {
   void draw() { .... }
}
class Polygon:public Shape {
   void draw() { .... }
}
class Hexagon: public Shape {...}
class Rectangle: public Shape {...}

Polygon* p = ...;
... p->draw(); ...
```



Based on static type information (or on static class analysis in a dynamically-typed language), the compiler can determine that the variable `p` holds some subclass of the class `Polygon`. By examining the complete class hierarchy of the program, the compiler can deduce that `p` will point to a direct instance of `Polygon`, `Hexagon`, or `Rectangle`. The compiler can look up the implementation of `draw` for each of these classes and determine that `Polygon::draw` is invoked in each case. Consequently, the compiler can replace the dynamically-dispatched `draw` message with a direct call to `Polygon::draw` and then attempt to further optimize the call. (Note that the `draw` message could not be optimized this way without examining the entire class hierarchy below `Polygon`.) The correctness of this optimization depends on several things, including the existence of `Polygon::draw`, the set of subclasses of `Polygon`, and the

lack of a `draw` method defined on any of these subclasses. For example, if a `draw` method is added to `Rectangle`, or if a `Triangle` class with its own `draw` method is added, the optimizations of the `draw` call site will be incorrect. A dependency mechanism must be able to detect these kinds of changes while simultaneously striving to ignore other changes that do not affect the outcome of the method lookup.

3.2 Possible Dependency Mechanisms

3.2.1 Header Files

One possible dependency architecture would build upon a standard `make` and header file-based system, such as is common in C++ environments. For example, the implementation file containing the `draw` call site could be made dependent on the header file that contains the definition of class `Polygon`. To support the kind of whole-program optimizations performed for `draw`, an implementation file would need to include (and be dependent on) the header files for all *subclasses* as well as superclasses of classes used in the implementation file. Whenever any of these header files is changed, the implementation file is recompiled. While simple and space-efficient, this approach is likely to have poor selectivity; our experimental results support this conclusion.

3.2.2 Fine-grained Dependencies

To achieve better selectivity than a header file system, the optimizing compiler for the SELF object-oriented language [Ungar & Smith 87] uses a fine-grained dependency graph structure linking each compiled procedure to the individual pieces of declarations, such as method declarations and class inheritance links, on which the method's compilation and optimization depended [Chambers et al. 89]. Additionally, the SELF system includes a special no-other-methods pseudo-declaration with each class that represents the *absence* of any other methods in the class. During method lookup, if a class is searched for a method without finding one, a link is added between the class's no-other-methods pseudo-declaration and the compiled code. If a method later is added to the class, then the compiled methods linked to its no-other-methods declaration are invalidated. (It would be possible to maintain separate method-absent declarations for each different message name, but this was deemed to require far too much space to be practical.) This system supports good selectivity for changing or removing a method, but it is fairly coarse when adding a new method to a class, particularly when adding a method close to the roots of the inheritance graph.[*] It also tended to invalidate lots of code whenever the programmer reorganized the inheritance graph without actually changing the outcome of many method lookups. For example, the `Shape` hierarchy might be reorganized with a `Quadrilateral` class inserted between `Rectangle` and `Polygon`; this insertion would not affect the outcome of the draw message, but SELF's dependency mechanism would still consider the method lookup out of date. Finally, it is fairly space consuming, with most compiled methods depending on many pieces of the program's inheritance graph. The SELF compiler does not support the kind of whole-program optimizations that require examining the entire inheritance graph, but to support them its dependency model could be extended with a no-other-descendants pseudo-declaration for each class which is invalidated whenever a new child is added to a class or any of its descendants.

3.2.3 Adding Filtering Nodes

To address SELF's problems with selectivity and space consumption, we developed the dependency framework described in this paper and used filtering nodes to represent the outcome of compile-time method lookup (as well as similar kinds of derived information). The structure of a large portion of our dependency

[*] Even worse, in SELF, the global name space is represented by a collection of classes inherited by all client code, and consequently defining a new global variable or class has the effect of invalidating nearly all compiled code.

graph is illustrated in figure 2. Each node in the figure stands for all the dependency nodes of a particular type, and an edge between two nodes in the figure represents all the edges in the dependency graph from a dependency node of the first type to a dependency node of the second type. The numbers in the figure indicate how many nodes and edges of a particular type are present in the application described in section 3.3.

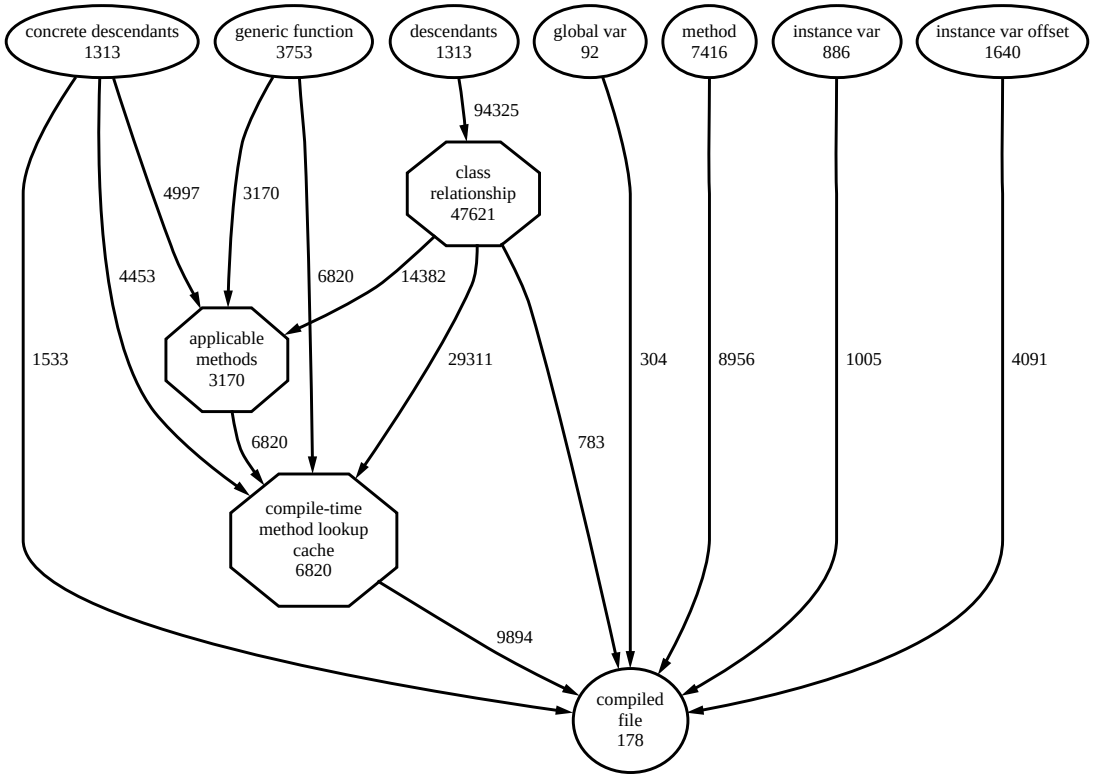


Figure 3: Cecil Dependency Graph Structure

Each compile-time method lookup filtering node depends (directly or indirectly) on all parts of the source program that might impact the outcome of the method lookup. If any of these properties change, the filtering node re-executes the compile-time method lookup. Only if the outcome of the method lookup changes, however, are any dependent compiled methods invalidated and recompiled. In this way we gain selectivity and insensitivity to superficial source code changes or class reorganizations that do not materially affect the outcome of method lookup. Moreover, the compile-time method lookup nodes act like factoring nodes, saving space by grouping together dependencies on pieces of source code that are shared by all optimized methods that perform the same method lookup. As a bonus, the compile-time method lookup nodes serve as persistent caches of method lookup results, speeding up compilation by performing a given method lookup at most once per compilation.

Our dependency graph takes longer to traverse than the SELF-style dependency graph, but the time spent checking compile-time method lookup filtering nodes in our system would be spent during recompilation of invalidated methods in a SELF-style system, so there is no real overhead to traversing our dependency graph.

3.3 Experiments

We performed several experiments to assess the selectivity and space costs of our dependency mechanism, relative to a header-file-based system and a SELF-like dependency mechanism.

3.3.1 Methodology

We have implemented our dependency framework in the context of the Cecil compiler and programming environment. Since we are writing the Cecil programming environment in Cecil, the programming environment itself serves as a reasonably large (40,000 lines) and rapidly developing benchmark application for evaluating our dependency mechanism. We modified the compiler to log source changes each time it was invoked; by replaying these logs we are able to recreate the original sequence of source modifications and monitor the subsequent recompilations with a variety of dependency mechanisms. The logs used to drive the simulations in this section cover a period of several weeks, during which 86 compilations occurred. The changes that occurred during this period are summarized below.

Table 1: Summary of Source Changes over 86 Compiles

Program Structure	Initial	Changed	Added	Removed	Final
Class Declarations	903	0	12	42	873
Methods	7272	714	455	341	7386
Instance Variables	889	2	34	23	900
Global Variables	73	8	21	3	91
Lines	39,697	n/a	n/a	n/a	40,874

Utilizing this history of changes, we simulated the following dependency systems:

- Header per File refers to a system where a single header file is simulated for each Cecil source file (Cecil does not require header files).
- Header per Class refers to a system where each class is assumed to have its own header file.
- SELF-like refers to a fine-grained dependency mechanism similar to the one described for SELF in section 3.2.2 that has the same structure as Cecil's except that no method lookup caches or other internal caches are used.
- Cecil refers to our system as described in section 3.2.3.

3.3.2 Selectivity Measurements

The expected benefit of a more complicated dependency mechanism is a reduction in the amount of recompilation after changes are made to the application's source files. The following table reports both the total number and the median number per compile of lines of source and files recompiled:

Table 2: Amount of Recompilation over 86 Compiles

Dependency System	Total lines recompiled	Total files recompiled	Median lines recompiled	Median files recompiled
Header per File	1,432,492	5939	26,888	108
Header per Class	1,328,766	5485	21,241	91
SELF-like	399,055	1409	1384	5
Cecil	189,662	645	1146	4

Using either header file-based system, approximately seven times as many total lines would have been recompiled than were recompiled under the Cecil dependency mechanism. The SELF-like dependency mechanism performs substantially better than the two header file systems, but it still would recompile twice as many total lines of code as Cecil did. The gap in the median number of lines compiled is much narrower. This is a result of a SELF-like dependency mechanism usually being selective for method edits, but being much less selective for other kinds of changes such as adding a method or reorganizing the inheritance graph.

3.3.3 Space Measurements

Finer-grained dependency mechanisms can be substantially more selective than a coarse-grained header file-based alternative. However, they require more space to maintain. The table below reports on the number of nodes and edges required to represent the dependencies of the fully optimized Cecil programming environment program. The row labelled Cecil + Factoring reports the size of our dependency graph after applying the automatic factoring algorithm described in section 2.3.

Table 3: Space Requirements of Dependency Graphs

Dependency System	Node Count	Edge Count
Header per File	370	2146
Header per Class	1501	6857
SELF-like	45,556	369,988
Cecil	81,270	258,846
Cecil + Factoring	87,262	206,548

As expected, the finer-grained dependency mechanisms require substantially more space to represent than the coarser, header file-based systems. When compared against the previous state-of-the-art SELF system, our system consumes roughly the same amount of storage but is substantially more selective, as shown in

table 2. Automatic factoring reduces the number of edges in the graph by 20% without substantially increasing the number of nodes. Whether the space usage of highly selective dependency systems are justified by their much better selectivity depends on the relative importance of selective recompilation versus limiting space consumption. But regardless of whether efficient means *selective, concise* or a blend of both, our framework provides the tools needed to construct efficient dependency systems.

4 Conclusions

The task of supporting selective recompilation in the face of complex derived interprocedural summary information and optimizations led us to develop a framework for engineering intermodule dependencies. Our framework extends a standard dependency graph with filtering nodes, automatic introduction of factoring nodes, and automatic topological traversal of the minimum dependency subgraph. Filtering nodes increase selectivity without consuming additional space, allowing procedural tests to be embedded in the dependency graph that squash invalidation messages when derived information has not changed as a result of source code changes. Filtering nodes are particularly useful for guarding caches of summary information which presents a restricted view of the source code. Factoring nodes can be inserted manually or automatically to save space when representing the common dependencies of groups of nodes. By incrementally maintaining a global topological ordering as the dependency graph is constructed, the minimum subgraph of the dependency graph can be traversed during invalidation. Our framework assumes a persistent repository for information about programs, which may run counter to traditional Unix compilation tools but is consistent with most programming environments and compilers that perform serious interprocedural optimizations.

We applied our framework to the problem of selective recompilation of object-oriented programs in a system that performs whole-program interprocedural analysis and optimization. Several kinds of derived summary information is used in this system, most notably a compile-time method lookup cache, and by using filtering nodes our dependency mechanism can avoid many unnecessary invalidations of optimized code. Using a log of several week's worth of actual program development, we compared several possible dependency mechanisms, and found that our mechanism was by far the most selective with no more space required than the previous most selective mechanism.

Our framework was designed to make its use by a tool implementor as painless as possible. Our experience suggests this is largely true. Clients of information are nearly unchanged; they only need to set "the current dependence" to their representative dependency node before accessing any information. Providers of derived information, such as interprocedural summary information, must do a little more work. Typically, this involves defining a new type of dependency node (a new subclass in our system), implementing a few simple operations and possibly a more complex operation to determine if the filtering node is out of date, and writing wrapper functions for clients that temporarily switch "the current dependence" to the supplier's representative for the duration of processing the client's request and add a dependence from the supplier's representative to the client's representative. The framework assumes responsibility for traversing the dependence graph to propagate invalidations and for invoking the right filter out-of-date procedures in the right order. Finally, some mechanism must be implemented to detect changes in the source code of the program and invalidate the right root dependency nodes.

Despite this fairly short list of obligations, it still can be tedious to define new dependency node types to represent each kind of information in the programming environment. Moreover, filter nodes usually have two ways of computing information: when invoked by a client for the first time, and when invoked during invalidation to test whether the information has changed. We currently are investigating refinements to our framework that can smooth over these remaining hindrances.

Acknowledgments

Filtering nodes for compile-time method lookup, the initial motivation for this work, arose out of discussions with David Ungar and Urs Hölzle. Ben Teitelbaum contributed to the early design of the dependency framework.

This research is supported in part by a National Science Foundation Research Initiation Award (contract number CCR-9210990) and gifts from Sun Microsystems, ParcPlace, and Pure Software. Stephen North and Eleftherios Koutsofios of AT&T Bell Laboratories supplied us with `dot`, a program for automatic graph layout, which has been invaluable for visualizing dependency graphs.

References

[Adams et al. 94] Rolf Adams, Walter Tichy, and Annette Weinert. The Cost of Selective Recompilation and Environment Processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[Burke & Torczon 93] Michael Burke and Linda Torczon. Interprocedural Optimization: Eliminating Unnecessary Recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings OOPSLA '91*, pages 1–15, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Chambers 92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, March 1992.

[Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

[Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF – a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings OOPSLA '89*, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Dausmann 85] M. Dausmann. Informationstrukuren und Verfahren für die getrennte Übersetzung von Programmteilen. Technical report, GMD-Bericht 155, R. Oldenburg Verlag, Munich, Germany, 1985.

[Feldman 79] Stuart I. Feldman. Make–a computer program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, 1979.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Lanaguge and its Implementation*. Addision-Wesley, Reading, MA, 1983.

[Hall et al. 93] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In *The Sixth Anunual Workshop on Parallel Languages and Compilers*, August 1993.

[MS et al. 93] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. On-line Graph Algorithms for Incremental Compilation. In Jan van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science*, number 790 in Lecture Notes in Computer Science, pages 70–86, Utrecht, The Netherlands, June 1993. Springer-Verlag.

[Newbery 89] Frances J. Newbery. Edge Concentration: A Method for Clustering Directed Graphs. *ACM SIGSOFT Sofware Engineering Notes*, 17(7):76–85, November 1989.

[Shao & Appel 93] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450, Charleston, South Carolina, January 1993.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addision-Wesley, Reading, MA, 1991.

[Tichy & Baker 85] Walter F. Tichy and Mark C. Baker. Smart Recompilation. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 236–244, New Orleans, Louisiana, January 1985.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.