Editing and Compiling Whole Programs

Keith D. Cooper
Ken Kennedy
Linda Torczon
Alan Weingarten
Mary Wolcott

Department of Computer Science†
Rice University
Houston, Texas 77251-1892

Abstract

The programming environment pays systematic attention to the problems entailed in developing, debugging, and optimizing large programs. This paper reviews the principal mechanisms provided by to support large programs. It describes the structure used to represent large programs and the tools that manipulate this structure. It also explores the way that the compiling system handles whole programs: performing interprocedural analysis, removing redundant modules, and generating customized procedure linkages.

1. Introduction

An environment designed to support large software systems must address several problems of scale. Programmers construct large systems from smaller subparts. System designers need tools for describing the interfaces between subparts and for specifying how to combine these subparts to produce an executable program. Given a structural description of the program, the system should automatically construct an executable that consistently and efficiently implements the specification. The system must be flexible enough to deal with problems like naming conflicts and multiple versions of subparts. Finally, the environment should use its knowledge of the program to provide both powerful debugging tools and excellent optimization of compiled code.

The programming environment attempts to provide coordinated support for programmers building large numerical software systems in Fortran [HoKe˜85, CoKT˜85]. It includes a structure editor for Fortran, called the *module editor*, an optimizing *module compiler*, a *debugging interpreter* and an *execution manager* all of which interact with the user under control of a window-based *monitor*. Each tool produces information used by other tools. The principal mechanism for passing such information is the environment's database.

Because numerical software systems are often large, the environment also provides tools that help manage the development of large systems. To create a program in the environment, the software developer builds a concise structural description of the program, called the program's *composition*. The composition serves as a basis for performing consistency and completeness checks on the program's design, as a set of rules to direct construction of an executable image, and as an aid to debugging. All of the environment's tools collect information about the program and its components. The composition also serves as a focal point for program-specific information; all such information is stored with the composition.

† This research has been supported by the National Science Foundation through grants MCS 81-21844, MCS 83-03638, and DCR 85-03608 and by IBM Corporation.

Many of the environment's tools use the composition and its associated information. The module editor uses the composition to provide the user with call templates that document the name, type, and dimension of each parameter at a call site. The optimizing compiler examines this structural representation of the whole program when looking for opportunities for cross-procedural optimizations. Using information found in the composition, the debugger can prepare a program for hybrid execution, running most of the program as compiled code but interpreting those procedures being tested.

In this paper we discuss the facilities provided in for editing and compiling whole programs. These facilities are provided by two of the environment's tools. The *composition editor* permits the user to interactively define the contents of a program while checking to insure that the program is well-formed. The *program compiler* prepares a program for execution. We describe these facilities and show how they cooperate with tools in the environment to help produce whole programs.

2. Goals

The environment supports the construction of *whole programs* with tools for creating, modifying, and manipulating composite objects. These tools address the following issues:

- *system contents* — The system designer must be able to specify which components make up a composite object and where each of them can be found.
- *system structure* — The description of the system must display its structure. Notationally, it must be clear which component provides each resource and which resources each component uses.
- *system consistency* — The programmer must be able to define interfaces to component subsystems so that the environment can check whether a proposed systems satisfies such specifications.
- *system documentation* — The tools for manipulating composite objects should provide convenient ways for entering and updating documentation in order to encourage builders and maintainers to write it.
- *system efficiency* — The compiling system should improve the efficiency of compiled code because of its detailed knowledge about program structure.

The program's composition provides the unifying structure for the tools that specify, modify, and compile programs in . The composition editor allows the user to browse, create, and modify compositions for programs, libraries, and composite modules. The program compiler analyzes a composition and its associated source modules to detect opportunities for interprocedural optimization and collect information that aids in performing purely intraprocedural optimization. Like other tools in the environment, they produce information used by other tools.

3. Fundamental Notions

This section introduces the structures and notation used to describe programs in the environment.

3.1. Units

A *unit* is the fundamental object in the environment. In , all the familiar components of a large system, like programs, libraries, and source code modules, are represented as units. A unit has an *implementation* and two sets that define its name scoping properties:

(1) its *requires* set contains all entry points referenced but not defined in its implementation.

(2) its *provides* set contains all entry points defined in its implementation that are available for external reference.

The *requires* and *provides* sets correspond to *imports* and *exports* lists. Thus, a unit can be viewed as an implementation along with sets that describe its interaction with the surrounding name space.

In general, a unit's implementation is a list of other units that it includes. The degenerate case, a *simple* unit, has a Fortran source file as its implementation. In , a simple unit is also called a *module*. A *compound* unit has a list, called its *members* list, as its implementation. Elements of a compound unit's *members* list are defined either directly by sets, in the case of an *internal* subunit, or by a named reference to an *external* subunit. A compound unit is also called a *composition*.

3.2. Name Scoping

The notion of name scoping used in follows naturally from this structural view of units. Each unit constitutes a distinct name scope; its *provides* set defines the set of internally defined names available to a unit that includes it. Similarly, a unit's *requires* set defines the set of names the unit inherits from its immediately enclosing scope. Either set may specify a binding of an internal name to an external name.[1] This scheme has reasonable flexibility. For example, a single compound unit can contain multiple implementations of a single entry point, provided the implementations occur in different subunits and the appropriate *requires* and *provides* sets prevent both names from reaching a common scope.

The scheme also has a clean simplicity. The set of names visible inside a given scope named C is simply

$$requires(C) \cup \bigcup_{x \in C} provides(x).$$

Throughout this paper, we use the notation $x \in C$ in place of $x \in members(C)$ whenever the meaning is clear.

3.3. Consistency and Completeness

A well formed composition satisfies certain constraints. A composition C is *consistent* if it fulfills the following two criteria. First, its name space must be well formed, that is, for any $x \in C$ and $y \in C$ such that $x \neq y$,

(1) $provides(x) \cap provides(y) = \emptyset$,

(2) $provides(C) \subseteq \bigcup_{x \in C} provides(x)$, and

(3) each of its subunits is consistent.

Second, all of its call sites must be type consistent with the corresponding entry points. This means that each call site in the composition agrees with the corresponding entry point in terms of the number of parameters and the type of each.[2]

A compound unit may also be *complete*. Formally, a composition C is *complete* if

(1) $\bigcup_{x \in C} requires(x) - \bigcup_{x \in C} provides(x) \subseteq requires(C)$, and

(2) each of its subunits is complete.

A complete unit has no unimplemented entry points, except those named in its *requires* set. In , a program is simply a consistent, complete composition with an empty *requires* set, along with a designated main entry point.

3.4. A Sample Composition

These ideas are best illustrated through an example. Suppose we are implementing a program that requires an efficient sorting routine, say *quicksort*, and we wish to use the linear-time median finding algorithm to select the pivot element. The median finder needs another sorting routine for small sets, for example *bubblesort* to deal with sets of fifty or fewer elements.

Figure 1 shows an instance of the composition editor. Its leftmost pane, the *composition pane*, shows one possible implementation of our sorting program. The innermost composition, named *median*, contains one of the two modules needed for the linear median algorithm, bubblesort. It doesn't yet include an implementation for the entry point *median*. This composition is included, in turn, in a composition named *quicksort*, which is one of the subunits of the outermost composition, named *example*.

The example illustrates the facilities for managing name scopes. Because the *provides* set of the composition *median* doesn't contain *sort*, the entry inside module *bubblesort* can't be seen outside that composition. Thus, it doesn't conflict with the entry *sort* provided by the module *quicksort*. The composition *quicksort*, in turn, makes the entry *sort* available externally under the name *qsort* by renaming it in the *provides* set. The example also illustrates the power of this system for structural description. The median finder is encapsulated into one composition while the main sorting procedure is encapsulated into another. Throughout, comments are used to elucidate the program map.

4. Composition Editor

The composition editor is a structure editor for a module interconnection language. Because the language was designed in concert with the editor, it has a simple, graphically-oriented, concrete syntax. The editor allows the user to specify

---

[1] For the sake of simplicity, equations in this paper that involve *requires* or *provides* sets ignore renaming effects.

[2] To facilitate enforcing this requirement, the *requires* and *provides* sets store the number of parameters and the type of each parameter with each entry.

Figure 1. Sample Composition Editor Display

which modules are incorporated in a composition and where their source is to be found. It subsumes the functions of the linkage editor input language in traditional systems and the module interconnection and configuration management languages of modern programming systems [DeKr˜76, MiMS˜79, Tich˜82, LaSc˜83, LeMc˜85].

The composition editor allows the user to edit a composition's *members* list and its *requires* and *provides* sets. The user can add and delete subunits from the *members* list. Subunits can be modules, external compositions referenced by name, or internal compositions. The details of internal compositions are also available for editing. Additionally, the user can add or delete entries from the *requires* and *provides* sets and specify renamings. The editor supports searching for specific entries in single compositions or collections of libraries. It also provides a mechanism for automatically completing a composition through library search.

Like most tools in , the composition editor produces information needed by other tools. In particular, for a given composition C, it produces:

(1) *members*(C), *requires*(C) and *provides*(C).

(2) *Makefile*(C) — the input to the [3] utility *make* required to produce an executable for C using standard tools.

(3) *CallGraph*(C) — information to help the program compiler construct the program's call graph.

(4) *EntryTable*(C) — a map from each entry in $\bigcup_{x \in C}$ *provides*(x) to the subunit of C that provides it.

It also performs the necessary checks to ensure that the composition is consistent.

4.1. The Display

To simplify the editor's display, the editor window is divided into several panes. Each pane hierarchically displays its data using an elision mechanism like that found in the module editor. The user can choose from several different elision settings. For example, the programmer can display all the entries in an unfamiliar library while showing only the names of other libraries.

An instance of the composition editor appears in Figure 1. The uppermost pane in the window is the *title pane*; it identifies the window as a composition editor. Directly below the *title pane* is the editor's *button pane*; it contains buttons for functions applied to the entire editor. There are also buttons associated with individual panes; their functions apply only to the contents of that pane. Section 4.2 describes the function of the various buttons.

The remaining space divides into four separate panes. The leftmost pane, called the *composition pane*, contains the current composition. Each unit is represented by a header and an indented implementation. The header displays the unit's name along with its *requires* and *provides* sets. The implementation is either a module or a *members* list. The example composition has two primary units, *driver*[*versn12*] and *quicksort*. The latter is a compound unit. Its internal components and their *requires* and *provides* sets are shown. The internal subunit *quicksort* provides the *qsort* entry by redefining the entry point *sort* provided by *quicksort*[*working*]. The programmer could hide these internal details using the elision mechanism.

The upper right pane, called the *library pane*, displays the current set of libraries in their search order. Libraries are displayed hierarchically, with several elision options. Libraries may be inserted or deleted from the list, but their internals may not be edited in this pane.

Immediately below the *library pane* is the *entries needed pane*. It displays all the unresolved entries in the composition. In a composition C, an entry is unresolved if it is called by some unit in C and is neither provided by a subunit nor contained in *requires*(C). Thus, the *entries needed pane* displays all the members of the set given by:

$$\mathit{provides}(C) \cup \bigcup_{x \in C} \mathit{requires}(x) - \bigcup_{x \in C} \mathit{provides}(x) - \mathit{requires}(C).$$

If this set is non-empty, the composition is incomplete.

Below the *entries needed pane* is the *copy buffer pane*. It displays subunits that have been clipped for copying from one pane to another.

4.2. Editor Functions

An element is selected by clicking the mouse on it. Editing functions are invoked using one of two mechanisms: clicking on a button or clicking on a selected item.

Clicking the mouse on a selected item brings up a menu of editing commands. These commands include operations to insert, cut, and paste objects. Using these familiar functions, the user can interactively build compositions. The menu includes a visit selection; if a module or an external composition is selected, this function invokes the appropriate editor on the selected item.

The editor's *button pane* contains three functions. The **database** button appends a pane to the right hand side of the window and invokes the database browser inside that pane. The **save** button writes the current composition back to the database; it includes options for saving the composition under a new name. The **run** button invokes the execution monitor to prepare an

[3] is a trademark of AT&T Technologies.

executable for the current composition and run it.

There are two buttons associated with each of the *composition pane* and the *library pane*. They provide equivalent function in each pane. The **mgmt** button provides a menu to control the elision mechanism. The **search** button provides facilities for textually searching the pane's current contents.

The *entries needed pane* has three buttons. The **resolve** button invokes a library searching mechanism. The editor searches, in order, the current members of the library list for each selected entry and adds the first implementation of each entry to the composition. Of course, adding implementations to the composition can add entries to the list of unresolved entries. If the entire list is selected, the editor attempts to resolve any entries added as a result of its own searching. The **query** button invokes the search mechanism to locate the next implementation of a selected entry in the library list. By repeating this, the user can step through all the implementations of an entry. As in the other panes, the **search** button provides a textual search of the pane.

The *copy buffer pane* has only one button. The **drop** button deletes the current element from the copy buffer.

4.3. Composite Modules and Libraries

Since a library is also a composition, it may be viewed as a composite module and may be incorporated into another composition. This facility is particularly useful because the program compiler doesn't include modules containing unused entries in executables. Thus, if a library contains several needed modules that must be used together, the programmer can simply include the entire library. If some entry in the library conflicts with an entry in the composition, the offending entry can be hidden using the normal name scoping facilities.

When a composition appears in the *library pane*, the user can select any individual subunit of its *members* list, but cannot directly select elements of those subunits. This facility allows a library designer to package a set of entries that must be used together as a single subunit, ensuring that any search which includes one entry will add all the related entries.

4.4. Autoinclude Libraries

The linkage editor, *ld*, provides the *autoinclude library*, a useful feature that cannot be duplicated using the scheme presented so far. Using *ld*, the programmer can provide one or more default libraries that behave in the following fashion: if any entry has no implementation, *ld* searches the libraries, at linkage time, for one. In a later compilation, if an implementation is expressly provided for the entry, the autoincluded module is no longer included.

The pervasive use of this feature in leads us to believe that it is worth providing an equivalent mechanism in . Thus, a library may be added to a composition in *autoinclude* mode, so that modules from the library are automatically added as needed to provide entries and deleted when they become either unnecessary or duplicates.

This feature differs from the wholesale inclusion of libraries. While including whole libraries is useful, it does not permit the inclusion of two libraries that have conflicting entries. The autoinclude feature permits this, deleting autoincluded entries as explicit duplicates are added to the composition.

4.5. Program Updates

If a system like the environment is to be truly useful, consideration must be given to the problem of program updates. For example, the system must deal with the situation that arises when a programmer edits and changes one of the component modules of a program in a way that makes the program incomplete, or worse, inconsistent.

In , we have taken a middle ground. The system prevents the user from making a change to a module when that change might make the program inconsistent. While this sounds restrictive, in practice it isn't. The only change that can make a program inconsistent is adding a new entry to a module. Thus, the system prevents addition of entries to a module once that module has been incorporated into a program. This is not a severe restriction since the programmer need only define a new module containing the changed source and build a new program composition around it.

On the other hand, changes to modules which would only make the program incomplete, such as adding a call or deleting an entry, are permitted since the environment allows incomplete

programs in the database. If a user makes such a change, the system will notify them and mark the incorporating program incomplete. This gives the programmer the flexibility to continue editing the module and fix the program composition later.

In all cases, program updates are bound by the basic guidelines for permissions in the environment [HoKe˜85]. In particular, public programs may not be modified, nor may any modules incorporated in a public program.

5. Program Compiler

Fortran users expect their compilers to generate excellent code. In , we plan to improve the efficiency of compiled code through systematic use of interprocedural analysis. Indeed, one of the primary goals of the project is to mount a concerted attack on the problems of performing interprocedural analysis and optimization in a compiling system [CoKT˜86a]. The compilers in the environment attempt to optimize whole programs instead of independent subroutines. A recent estimate by Conradi suggests that ambitiously optimizing whole programs can lead to improvements of up to twenty percent in run-time efficiency [Conr˜83].

To provide effective support for optimizing whole programs, we have divided the compilation process between two compilers: a *module compiler* and a *program compiler*. The *module compiler* consists of an optimizer and code generator for single modules. It corresponds to the back end of a traditional optimizing compiler. The *program compiler*, the principal innovation of the optimization strategy in , manages the optimization of the whole program.

The program compiler is an *incremental compilation system* for whole programs, where the single source module is the incremental unit. The program compiler is invoked after a change to either a module or the program composition. To construct an executable, it

- analyzes the program to discover what interprocedural information has changed,
- uses a special technique called *recompilation analysis* to determine which source modules need to be recompiled [CoKT˜86b],
- invokes the module compiler to perform the individual recompilations, and
- builds a new executable for the program.

Using its knowledge about the entire program, the program compiler can perform optimizations that are difficult to implement in a traditional independent compilation system. Two examples that we will examine are eliminating redundant module implementations and replicating definitions to improve optimization.

5.1. Incremental Updates

To improve the quality of the code it generates, the program compiler computes various types of interprocedural information. As part of the design work for the environment, we have developed the fastest known technique to compute flow-insensitive interprocedural summary and aliasing information. The algorithm requires $\mathbf{O}(E\alpha(E,N))$ time, where E is the number of call sites in the program, N is the number of procedures in the program, and α is an extremely slow growing function bounded above by the functional inverse of Ackermann's function. The algorithm and incremental updating techniques for the information it produces are described in detail in other papers [CoKe˜84, CoKe˜86].

In response to an editing change, either to the program's composition or the source of one of its components, the program compiler updates the program's interprocedural information. Because programs can be large, the program compiler chooses the most cost-effective mechanism to achieve this goal. Two different techniques are available to it: recomputing the summary and aliasing information, or updating the information with an incremental technique. The literature dealing with analogous global problems suggests that the latter is preferable. Unfortunately, the interprocedural data flow problems solved by the program compiler are more complex than typical global problems. In some cases, complete recomputation may be less expensive.

To see this point, consider the algorithm for computing summary information. We cannot directly compute the summary information efficiently because of complications introduced by the combination of call-by-reference parameter binding and name scoping rules. Instead, the Cooper-Kennedy technique constructs a mapping that describes the parameter binding behavior of the program; the mapping can be used to directly construct the desired interprocedural sets. The incremental update techniques involve either an

update to the mapping relation, a recalculation of the desired sets from the mapping, or both. The updates fall into two categories: informational changes or structural changes.

In response to an informational change, the program compiler can simply change the initial fact sets for the changed procedure and redo the direct computations using the old mapping relation. These updates require **O**(1) bit-vector steps. For both types of update, the length of the bit-vectors is proportional to the number of procedures in the program.

In response to a structural change, the program compiler must update the mapping relation. This requires far more work than an informational change. Each structural change requires an update that takes **O**(N) bit-vector steps in the worst case, where N is the number of procedures in the program.

Thus, given a set of changes, the program compiler must decide whether to apply the batch algorithm or the incremental algorithm. While the constant of proportionality on the batch algorithm is large, for a sufficient number of structural changes the batch algorithm becomes an inexpensive alternative to the incremental update. So, the program compiler uses the number of call site additions and deletions as the key factor in deciding how to construct consistent interprocedural information.

This section has discussed only the updating techniques for summary and aliasing sets. We are currently investigating incremental techniques for updating sets describing interprocedural constants [CCKT˜86].

5.2. Eliminating Redundant Modules

While the hierarchical style of composition supported in provides the user with tools for abstracting away details about the implementation of modules, it can also give rise to very large executables due to multiple inclusions of a single module in different subunits. Rather than implement mechanisms to eliminate redundancy in the composition itself, we rely on the program compiler to determine the minimal number of copies of each module required to correctly implement the composition. By applying this space optimization, the environment allows the user the organizational expressivity of trees coupled with the storage efficiency of acyclic graphs. As a side effect, this optimization deletes any module not reachable from the main program entry point. This allows the programmer to include whole libraries in a composition without concern for the size of the resulting executable.

Ignoring the linkage tailoring considerations discussed in the next section, a module can generally be implemented with a single copy of its body. To understand the exception to this rule, consider a module m that calls entry point e. If m appears in two subtrees of the composition and e is provided by different modules in each subtree, then a single copy of m doesn't suffice to correctly implement the composition. The compiler could avoid this problem by using a complex calling sequence. Instead, generates two copies of such a module, allowing the program compiler to determine when different instances of such a module can be coalesced.[4]

To determine when it is safe to coalesce multiple copies of a module, the program compiler examines an annotated version of the program's call graph. Each node in the graph represents a module in the program and is labeled by the module's name. The edges represent calls from within one module to entries within another and are labeled by the name of the called entry point. The program compiler constructs the initial call graph by starting at the root and creating nodes for module instances as it encounters calls to entries they contain. If it finds duplicate instances of a module, it creates a node for each instance. This creates a graph that contains only modules actually reachable from the program's main entry; at the same time, the graph may contain multiple nodes with the same name.

The minimization procedure is shown in Figure 2. It operates by examining sets of nodes that can be correctly implemented by a single copy of the procedure. In a set S, it looks for two nodes x and y that have arcs labeled e going to nodes that are members of different sets. If such nodes exist, we say that x and y have a *call conflict*. Two nodes can be coalesced only if they have no call conflicts. The algorithm uses a worklist of sets, *Candidates*, that may need to be partitioned. The algorithm stops when *Candidates* is

[4] To preserve the semantics of Fortran's SAVE statement, the two code bodies share a single static storage area.

(1) Partition the nodes into sets by label. Initialize *Candidates* to contain all sets consisting of more than one node.

(2) While *Candidates* is non-empty,

Remove an element S from *Candidates*.

Partition S into sets $T_1, T_2, \cdots, T_n$ such that nodes $m_1, m_2 \in T_i$ iff all edges leaving m_1 and m_2 have no call conflict.

For every set containing a node V_k that calls an entry in T_i and a node V_l that calls an entry in T_j, $i \neq j$, add that set to *Candidates*.

Figure 2

empty. At this point, all the nodes in a single set can be coalesced. The compiler makes a pass over the graph to coalesce nodes and relabel the arcs appropriately.

The number of sets considered during the algorithm is bounded by $2N$, where N is the number of nodes in the graph. During execution of the algorithm, a set can be on the *Candidates* list at most twice. A set may be placed on the *Candidates* list once during initialization. Once a set S is removed from the list, it can only be placed back on the list one time, the first time that splitting another set creates a call conflict between members of S. Because a set placed on the list this way is always partitioned, it no longer exists and cannot reappear on *Candidates*. Thus, a set appears on the *Candidates* list at most twice, so step 2 requires at most $\mathbf{O}(N)$ iterations.

5.3. Interprocedural Optimization

The program compiler analyzes the program and selects one of several ways to implement each procedure call. It assigns linkage styles to call sites in a way that is likely to improve the run-time performance of the resulting code. Deciding how to handle each call site is a complex problem.

At each call site, the program compiler chooses from three alternatives:

(1) it can perform *inline substitution*, replacing the call site with a copy of the called procedure's body appropriately modified to model parameter binding;

(2) it can *clone* the called procedure, binding the call site to a copy of the procedure invoked only by call sites with similar interprocedural properties; or

(3) it can generate a *standard procedure call*, assigning the call site to an instance of the procedure body shared in common by all call sites that invoke the procedure.

We call the problem of assigning call styles to call sites *linkage tailoring*.

Each call style has advantages and disadvantages. Inline substitution completely eliminates the overhead of the procedure call and results in optimizing the called procedure specifically to the run-time situation at a single call site; it can also lead to exponential growth in the size of the resulting program. Procedure cloning can prevent call sites that require full generality from inhibiting optimization at call sites where the generality isn't needed; it also carries with it a space penalty. Generating a standard linkage incurs no space penalty; unfortunately it opens up no new ground for interprocedural optimization.

To assign call styles to call sites, the program compiler uses a conceptually simple technique. First, any procedures that are called from only one call site are marked for inline substitution. Second, any procedure whose anticipated code size is smaller than the size of the procedure linkage is marked for inline substitution. These two cases remove from the analysis all call sites where inline substitution is always profitable. For each remaining call site, the program compiler estimates the anticipated improvement and code size growth and then enters this alternative in a priority queue ordered by estimated improvement.

For each remaining call site, the potential improvement must be estimated assuming inline substitution and procedure cloning. Thus, for each call site, the program compiler estimates the improvement possible with inline substitution and enters that information in the queue. Next, each procedure is examined. If the information on incoming call graph edges differs enough to make improvements from cloning likely, it enters that alternative into the queue.

Given the initialized queue, the program compiler removes the most profitable linkage from the queue, marks the call graph accordingly,

and updates its estimate of final code size. It repeats this process until the estimated final code size reaches a predetermined multiple of the original estimated code size. This condition prevents the exponential explosion of code size possible with a complete inline substitution. All remaining call sites receive standard linkages.

The one remaining issue is computing estimated improvement. In , the editor will produce an estimate of possible improvement due to interprocedural information using a technique similar to that described by Ball [Ball˜79]. To obtain the estimated improvement for a call site, it estimates the improvement possible in the called procedure based on the information known at the call site. It then multiplies that estimate by a static estimate of dynamic execution frequency. The estimated execution frequency is derived from the loop nesting depth of the statement within the procedure and an approximation of the loop nesting depth along all acyclic paths from the program's main entry point to the call site. The estimated improvement for a cloned procedure is the sum of the estimates of the effected call sites.

An interesting opportunity exists for one further optimization on the call sites that have not been substituted inline. By changing the calling sequence, the program compiler can move the prologue code across the call site boundary into the calling procedure. Assuming the prologue code is reasonably short, the space growth would be proportional to the number of call sites in the program. However, in return for this growth, the module compiler would get the opportunity to optimize the prologue code with the calling procedure. For example, the module compiler could move loop-invariant prologue code out of loops that contain call sites. In particular, large parts of the code to establish the addressability of parameters is likely to be loop invariant.

Rather than applying this optimization at all call sites, the program compiler could transform only those call sites where it appears that an actual run-time improvement would occur. However, it is likely that the overhead of tracking this partial application of the optimization across the lifetime of the program would overwhelm the benefits gained from applying it. Since the space penalty for partially opening up each call is reasonably small, the program compiler will either apply this transformation to all call sites or to none, depending on the level of optimization selected by the user.

6. Implementation Status

The current version of the programming environment runs on both the IBM RT/PC and the SUN Workstation.[5] It includes stable versions of many tools, including a module editor, debugging interpreter, execution monitor, documentation editor, database browser, calculator, and help processor. The current version of the composition editor implements the features described in section 4, except for autoinclude libraries. It was installed in the environment at mid-summer, replacing an earlier prototype. A skeleton program compiler that builds interprocedural information has been part of the environment since early 1986. The program and module compilers are under construction; we expect to install them in the environment in early 1987. Currently, the environment uses the utilities *make* and *f77* to construct an executable with a makefile automatically produced by the composition editor.

6.1. Acknowledgements

The implementation team has provided us with a marvelous vehicle for experimenting with our ideas. In particular, Kim Taylor and Scott Alexander worked on earlier versions of the composition editor. The example composition for quicksort using the linear median finder is due to Robert Hood. To all these people go our heartfelt thanks.

References

[Ball˜79] J.E. Ball. Predicting the effects of optimization on a procedure body. *Proc. of the SIGPLAN 79 Symposium on Compiler Construction, SIGPLAN Notices 14(8)*, August 1979, 214-220

[CCKT˜86] Callahan, D. Cooper, K.D., Kennedy, K. and Torczon, L. Interprocedural constant propagation. *Proc. of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices 21(7)*, July 1986, 152-161.

[5] RT/PC is a trademark of IBM; SUN is a trademark of SUN Microsystems, Inc.

[Conr˜83] R. Conradi, Inter-procedural optimization of object code. TR 25/83, Division of Computer Science, University of Trondheim, Trondheim-NTH, Norway, 1983.

[CoKe˜84] K.D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. *Proc. of the SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices, 19(6)*, June 1984, 247-258.

[CoKe˜86] K.D. Cooper and K. Kennedy. Flow insensitive interprocedural problems. *In preparation.*

[CoKT˜85] K.D. Cooper, K. Kennedy and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. *Proc. of the SIGPLAN 85 Symposium on Language Issues in Programming Environments, SIGPLAN Notices 20(7)*, July 1985, 107-116.

[CoKT˜86a] K.D. Cooper, K. Kennedy and L. Torczon. Optimization of compiled code in the programming environment. *Proc. of the Nineteenth Annual Hawaii International Conference on Systems Sciences*, January 1986, 492-502

[CoKT˜86b] K.D. Cooper, K. Kennedy and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *Proc. of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices 21(7)*, July 1986, 58-67.

[DeKr˜76] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Software Engineering SE-2(2)*, June 1976, 80-86.

[HoKe˜85] R.T. Hood and K. Kennedy. A programming environment for Fortran. *Proc. of the Eighteenth Annual Hawaii International Conference on Systems Sciences*, January 1985, 625-637.

[LaSc˜83] B.W. Lampson and E.E. Schmidt. Organizing software in a distributed environment. *Proc. of the SIGPLAN 83 Symposium on Programming Language Issues in Software Systems*, San Francisco, 1983, 1-13.

[LeMc˜85] D.B. Leblang and G.D. McLean. Configuration management for large-scale software development efforts. *Workshop on Software Engineering Environments for Programming-in-the-large, Harwichport, Massachusetts, June, 1985.*

[MiMS˜79] J.Mitchell, W. Maybury and R. Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto CA, April 1979.

[Tich˜82] W.F. Tichy. A data model for programming support environments and its applications. *Automated Tools for Information Systems Design (H.-J. Schneider and A.L. Wasserman, eds.)*. North-Holland, 1982.