The Cost of Selective Recompilation and Environment Processing

ROLF ADAMS, WALTER TICHY, and ANNETTE WEINERT
University of Karlsruhe

When a single software module in a large system is modified, a potentially large number of other modules may have to be recompiled. By reducing both the number of compilations and the amount of input processed by each compilation run, the turnaround time after changes can be reduced significantly.

Potential time savings are measured in a medium-sized, industrial software project over a three-year period. The results indicate that a large number of compilations caused by traditional compilation unit dependencies may be redundant. On the available data, a mechanism that compares compiler output saves about 25 percent, smart recompilation saves 50 percent, and smartest recompilation may save up to 80 percent of compilation work.

Furthermore, all compilation methods other than smartest recompilation process large amounts of unused environment data. In the project analyzed, the average environment of a compilation unit is up to 1.9 times the size of that unit, but less than 20 percent of the environment symbols are actually used. Reading only the actually used symbols would reduce total compiler input by about 50 percent.

Combining smart recompilation with a reduction in environment processing might double to triple perceived compilation speed and double linker speed, without sacrificing static type safety.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.2.9 [**Software Engineering**]: Management—*software configuration management*, D.3.4 [**Programming Languages**]: Processors—compilers

General Terms: Languages, Measurement

Additional Key Words and Phrases: Empirical analysis, environment processing, selective recompilation, separate compilation, smart recompilation, software evolution

1. INTRODUCTION

Separate compilation is an important technique for reducing the time programmers must wait after changing software. It is beneficial even with today's fast workstations, because our software systems have grown dramatically in size and our programming languages have become more expensive to translate. In particular, the facilities for intermodule type checking of modern programming languages carry a price: A single, small change in a frequently

Authors' current addresses: R. Adams and W. Tichy, Informatics, University of Karlsruhe, D-76128 Karlsruhe, Germany; A. Weinert, ALSYS GmbH, D-76199 Karlsruhe, Germany.

used interface may cause the recompilation of nearly the entire system, which may take hours, days, and, in extreme cases, weeks.

A number of mechanisms have been proposed to reduce the cost of separate compilation, while retaining the advantages of performing intermodule type checking during compilation. To quantify the potential savings of these mechanisms, we analyze the history of a medium-size, industrial software system. The last configuration of the system consists of 161 compilation units and 63,000 lines of code, excluding comments.[1] The history available includes the day-to-day changes over a period of three years. The system is written entirely in Ada. Although our results are specific to Ada, we conjecture that they would not differ much for other languages with separate compilation such as C or Modula.

The main variable of interest is the compilation cost, measured in both number of input units and number of input lines. We also report various size attributes, the number of changes, and the change distribution. This study is in the spirit of the seminal work by Lehman and Belady [1985] on program evolution. However, while Lehman and Belady make their observations on the time scale of months and years, we make ours on the scale of days.

The mechanisms for reducing compilation costs considered in this paper all reduce the amount of input the compiler must process. There are two nearly independent methods: *selective recompilation* and *selective environment processing*. Selective recompilation reduces the number of compilations performed after a change, while selective environment processing reduces the amount of input per compilation unit by abbreviating its environment. The environment of a compilation unit consists of the set of all symbols (types, constants, variables, macros, subprogram headers, etc.) that are external to the unit, but that the unit may access. For each such external symbol, the environment provides various attributes to be used for type checking and code generation. Processing the environment may make up a significant portion of the total compilation cost for each unit.

Our results show that *smart recompilation* [Tichy 1986] is the most effective of the methods that do not sacrifice type safety during compilation. It can save over half of all compilations, compared to the conventional, "cascading" method based on compilation unit dependencies. Smartest recompilation can save even more, but requires a potentially expensive consistency check at linkage time.

Environment reading can also be improved noticeably: Reading only the used symbols may shorten environments by as much as 80 percent. Since we observed that the average environment is 1.4 to 1.9 times the size of a compilation unit, the savings regarding environment reading translates into a second net input reduction of about 50 percent. Combining smart recompilation with an efficient environment reading technique would yield a total reduction in input bulk by a factor of four and would significantly speed up both compiling and (incremental) linking.

[1]Including comments, the number of lines is 99,000 Unless noted otherwise, our measure of lines of code excludes comments

The rest of this paper is organized as follows: The remaining paragraphs in this section define the selective recompilation and environment processing methods. Section 2 describes the design of the experiment, including the metrics used and how they were computed. Section 3 presents the main results, while Section 4 interprets them.

1.1 Selective Recompilation

The simplest method to restore consistency after changes is to recompile everything. This method, called *big bang*, is clearly too expensive to be practical for large systems. *Selective recompilation* refers to methods that may compile less than big bang. The main such methods are defined below:

(1) *Cascading recompilation.* A dependency relation is imposed upon compilation units. When a unit changes, its directly and indirectly dependent units are compiled also. A single change can spread recompilations down multiple chains of dependencies, hence, the name of the method.[2]

To improve the efficiency of cascading recompilation, programming languages often differentiate between specification units and implementation units. The specification units contain no procedure bodies or hidden data structures; those appear in implementation units alone. With the rule that a given unit may only depend on specification units, cascading recompilations occur only when specification units change. This approach saves compilation time, if specifications change less frequently than implementations. (Our data do not bear this out, however.) The languages Mesa [Mitchell et al. 1978], Modula 2 [Wirth 1985], and Ada [U.S. Department of Defense 1983] include such a scheme; the program MAKE [Feldman 1979] can add it to programming languages that lack it, including C and Pascal. The language CHILL [CCITT 1988] also uses this mechanism, but requires no separation into specifications and implementations.

Most implementations of cascading recompilation also permit the recompilation of individual units, without triggering recompilations of dependent units. To establish full type consistency, the recompilations have to be performed later, for example, prior to linking.

While cascading recompilation is a vast improvement over big bang (see Borison [1989] for a comparison), a number of techniques have evolved that reduce the number of compilations even further. All of these techniques have cascading recompilation as their basis, but cut short some of the cascades.

(2) *Surface change.* This method determines whether a set of changes is semantically irrelevant, that is, whether the changes are confined to comments and whitespace. If so, recompilations of dependent units are unnecessary. This method was implemented for SR [Olsson and Whitehead 1989]. Borison [1989] discussed several of its variants.

(3) *Cutoff recompilation.* This method initiates recompilations in the same way as cascading recompilation, but compares old and new compiler outputs.

[2]Also known as "trickle-down recompilation."

If the old and new compiler outputs for a unit do not differ, then no change can propagate out of that unit. It is therefore safe to cut off further compilations of dependents of that unit. The Mary-2 system [Rain 1984] uses this method.

Note that cutoff recompilation and surface change do not necessarily produce the same set of compilations. Where surface change performs a simple lexical comparison, cutoff recompilation must perform a full compilation and compare the output. On the other hand, cutoff recompilation takes into account semantically relevant changes that happen not to propagate. Thus, the units processed by cutoff recompilation are a subset of the units compiled or analyzed by surface change.

(4) *Semi-smart recompilation.* This method classifies the changes in specification units according to added declarations, modified procedure headings and variables, and other changes. Added declarations are, of course, unused in unchanged units and therefore cause no recompilations. In some languages, changes to procedure headings and variables propagate at most one level to the directly dependent units; hence, no further cascading recompilations are triggered. Other changes, such as changes of types and constants, trigger the compilation of all directly and indirectly dependent units. The method is due to Kamel, who implemented it for BNR-Pascal [Kamel 1987].

(5) *Smart recompilation.* This method refines the dependency relation to record the use of individual declarations. Whenever a unit is processed, the compiler stores the directly and indirectly used declarations from other units. If any of the other units change, the intersection of the set of changed, added, or deleted declarations with the set of used declarations indicates whether the dependent unit must be recompiled. Variants of this method have been implemented for Pascal, C, and CHILL [Tichy 1986; Schwanke and Kaiser 1988; Eidnes et al. 1989]. Integral C [Ross 1987] achieves a similar effect by permitting the incremental update of so-called *fragments*, which typically correspond to individual declarations. The most sophisticated system for selective recompilation is perhaps the Rational programming system for Ada [Feiler et al. 1988]. It allows the programmer to choose between cascading recompilation and smart recompilation by selecting the granularity of change (compilation unit, individual declaration, or even individual statement).

(6) *Attribute recompilation.* This method is a refinement of smart recompilation. It refines the dependency relation to record the use of individual attributes of declarations. For example, suppose a record type is changed in such a way that its size attribute remains invariant (e.g., by reordering or renaming fields). A unit referencing that record type need not be recompiled if it uses only the record's size attribute. See Dausmann [1985] for a more detailed discussion of this technique.

(7) *Smarter recompilation.* Schwanke and Kaiser [1988] observed that smart recompilation is too conservative: It requires full type consistency, even though programmers often wish to leave harmless inconsistencies in a system until current changes have been tested. Consider, for instance, a

change that redefines a type T. As long as the old and new versions of T are used in different partitions of a system, and the interface between the partitions does not involve T, then it is perfectly acceptable to recompile only one partition of the two. Smarter recompilation automatically determines these partitions. More important than reducing compilation time is the fact that this method may save programming time: After changing T, a programmer need only update those references to T that are relevant for testing. If the change has to be discarded or reworked, the programmer has not lost time establishing full consistency in a transient situation.

(8) *Smartest recompilation.* This method requires no dependencies among units, because it reads no environment at all. Each unit is compiled in isolation. The compiler employs type inference to derive the most general possible (parametric) types for undeclared identifiers. The derived types are unified with the actual types during a linking phase. The method was proposed for Pascal [Levy 1984] and later refined for ML [Shao and Appel 1993].

Although compiling quickly, smartest recompilation may be counterproductive because it slows error removal. Error reports regarding the interfaces between modules are delayed until integration time, long after programming work on a module has been completed. Error messages this late appear out of context and are therefore difficult to remedy. Programmers generally prefer to receive error messages as early as possible.

This paper quantifies the compilation costs of cascading recompilation, surface change, cutoff recompilation, smart recompilation and smartest recompilation. The figures for smartest recompilation do not include the cost of the consistency check to be performed later. However, they are still useful as lower bounds on compilation work. The costs of semi-smart and attribute recompilation are approximated. Analysis of smarter recompilation is impossible with our data, due to a lack of information about transient, inconsistent configurations.

1.2 Environment Processing

While selective recompilation seeks to reduce the total number of unit compilations, environment processing attempts to reduce the amount of environment data that must be processed in each compiler run. The environment of a given unit simply consists of those units on which the given unit depends. Reading the environment has been nicknamed *big inhale* for the usually large amount of information that must be scanned before actual compilation commences. Conradi and Wanvik [1985] reported that big inhale may account for 30–50 percent of compilation time.

Gutknecht [1986] classified the methods for environment reading as follows:[3] Call the units making up an environment *environment units*. Gutknecht's first distinction is that environment units may be read in *source*

[3]We changed Gutknecht's terminology, but preserved the concepts.

or in *compressed* form. In source form, environment units are character strings according to the syntax of a programming language, usually stored as text files. The compressed form is an encoding that can be loaded quickly into a compiler's symbol table. The compressed form must be generated, for example, during the first time its source is encountered by the compiler. If environment units change less often than they are read, then compression saves time at the expense of space.

The second dimension of Gutknecht's classification distinguishes whether an environment unit is *unresolved* or *resolved*. An unresolved environment unit is one that contains only pointers to its (sub-) environment units, whereas a resolved environment unit has been extended with the information contained in all its (sub-) environments. A resolved environment unit has the advantage that it can be loaded all at once, without looking up additional units. However, resolved environment units may consume dramatically more space, because they duplicate information.

Environment units in source form are usually unresolved, while resolved environments can easily be compressed during construction. The other combinations are also possible. Some examples follow:

(source, unresolved): C compilers and Foster's Modula-2 compiler [Foster 1986];

(compressed, unresolved): many Ada and Modula compilers, for example, Wirth's [Wirth et al. 1982];

(source, resolved): Tichy's compiler for INTERCOL [Tichy 1981]; and

(compressed, resolved): Gutknecht's Modula-2 compiler [Gutknecht 1986].

There are several methods to reduce the amount of environment information to be processed, independent of Gutknecht's classification. The reduction of environment input is important, because programmers, perhaps either out of ignorance, carelessness, or lack of time, tend to make environments larger than necessary. The minimal environment input would be achieved if the compiler selected only those declarations that it actually needed. Such a method, called *selective embedding*, was proposed by Cashin et al. [1981]. The basic idea is to interleave syntax analysis, environment reading, and semantic analysis. Syntax analysis, besides building a syntax tree, determines the set of unknown identifiers. The environment reading phase then looks up the unknown symbols in the environment and enters their attributes into the symbol table. This process may add additional unknown symbols, so that it recurses. After all unknown symbols have been resolved, the environment is complete, and semantic analysis can commence.

Clearly, selective embedding saves space in the compiler. Time is saved only if the individual declarations can be accessed quickly, without having to read whole environment units.

A less selective, but simpler method is *environment pruning*. Here, environment units are skipped entirely if none of their declarations are used. The method can be implemented in the compiler or as a separate program. If implemented in the compiler, all external symbols must be qualified with the name of the environment unit in which they are defined. Thus, a compiler only reads those environment units whose identifiers appear as qualifiers in the program text. If there are unqualified external symbols, the compiler must process all units in the environment.

An environment analyzer separate from the compiler can afford a more thorough analysis. It can process all units to detect spurious unit dependencies and delete them. The program Incl [Vo and Chen 1992] performs this analysis for C programs. It is run periodically to "clean up" a dependency architecture that might have undergone some deterioration during maintenance. Incl does not require qualification of external symbols.

This paper reports on the cost of environment reading for the following methods: unresolved source environments, selective embedding, and environment pruning. Data about compressed environments are not available.

2. DESIGN OF THE EXPERIMENT

We analyzed two subsystems of a software configuration management system called BiiN[4] SMS, developed at Siemens Corporate Research in Princeton, New Jersey [Schwanke et al. 1989]. The two subsystems considered were CM (configuration management) and VM (version management), both written entirely in Ada. CM and VM are independent of each other and were therefore analyzed separately. Both depend heavily on the operating-system interface and another subsystem called SUPPORT. Neither SUPPORT nor the operating system were analyzed, because of insufficient space in the analysis program.

The entire daily development history of SMS is available, because programmers stored their daily changes into an archive managed by the version control system RCS [Tichy 1985]. Compilations occurred in a batch run at night. The batch job incorporated the changes of the day into a consistent baseline to be tested the next day. If a new revision caused a compilation error, then it was replaced with the one in the previous configuration. The compilations performed for each configuration were exactly those prescribed by the Ada language definition.

In essence, the RCS archive records a snapshot configuration of SMS at the end of each day. Programmers performed some preliminary testing during the day, and only the end results of these tests appear in the archive. The preliminary tests consisted mainly of syntax checking and some small unit tests. Because of the high cost of recompilation, programmers actually had to apply an unsafe form of selective recompilation for this part of their work: They guessed the proper compilation order and then invoked the Ada compiler on individual units, but turned off any propagated compilations. Full consistency could only be reestablished during the batch run at night. The compiler ran at only 1–5 lines/s on a time-sharing system, and the complete dependency analysis with all recompilations took 5–6 hours. Programmers simply could not afford to wait this long during the day.

We analyzed the daily snapshots with a special program written in YACC

[4]BiiN is a trademark of Siemens.

and C. This program consists of a syntax analyzer, a symbol table manager, and a cross-reference generator for Ada. The analyzer successively processes several configurations. The initial configuration builds up the symbol and cross-reference tables, while later configurations update this information. Ada's rules for visibility and overloading made the symbol table management quite complicated. An additional complication was the need to process incomplete configurations, since both the operating-system interface and the subsystem SUPPORT were excluded from the analysis.

The analyzer estimates the costs for cascading recompilation, surface change, cutoff recompilation, smart recompilation, and smartest recompilation, as well as the environment reading techniques for unresolved source environments, environment pruning, and selective embedding. The recompilation costs were measured in both number of units processed and lines of input. The first measure reflects the number of files to be opened, while the second determines total input bulk. Lines processed may be somewhat more meaningful, especially because specification units are typically shorter than implementation units. Furthermore, compiler speed is commonly measured in lines per time unit. Other factors that influence compilation speed, such as the quality of the compiler (optimizing vs. nonoptimizing), the operating system, and the hardware are not measured. As a by-product, the analyzer also collects statistics about the number and class of changed declarations per specification unit.

2.1 Configuration Selection

Configuration selection is the process of choosing the revisions for a consistent configuration. With the SMS database, this is a simple process: Since only compilations, but no update activity, occurred at night, the configurer simply chooses the latest version of each unit at a given date. Occasionally, several revisions of a unit were stored in RCS on a single day, presumably when programmers found errors. We ignore all but the last of multiple revisions per day.

If our analyzer discovers a syntax error in a unit, it backs up to the revision used in the previous configuration, just as the actual development environment would have. However, our analyzer does not detect semantic errors. Thus, it is possible that our analyzer processes some configurations that were never actually compiled successfully. The potential errors introduced in this way include overestimation of the number of compilations that were actually performed and perhaps slight inaccuracies in the dependency hierarchy. However, since programmers compiled and tested their programs before depositing them into RCS, and because they were expected to correct any semantic errors in the next revision, we believe the number of undetected semantic errors to be quite small and their effect on compilation costs negligible.

2.2 Estimating Compilation Costs

The basis for computing the compilation costs for the various methods is the dependency relation at unit and declaration level. These are defined for Ada as follows:

Unit level. Compilation unit U2 *USES* compilation unit U1, iff

—U2 mentions U1 in a WITH-clause,
—U2 is an implementation unit and the corresponding specification unit mentions U1 in a WITH-clause,
—U2 is an implementation unit and U1 the corresponding specification, or
—U2 is a subunit and U1 an ancestor unit of U2.

Declaration level. Declaration D2 *USES* a declaration D1, iff D2 refers to D1 or to D1's subordinate declarations.

The following clarifications are in order: First, the definitions ignore implicit dependencies caused by generics and in-line subprograms. These may cause further dependencies on both unit and declaration level. Fortunately, the data do not contain any generics, since the Ada compiler in use at the time did not support them. There are only a few in-line subprograms (six in the final configuration); our analyzer simply treats these as normal subprograms.

Second, a compiler may introduce additional dependencies. For instance, a compiler may employ procedure numbers or offsets for variables to resolve intermodule links, even though a linker is a better tool to handle such links. Interprocedural optimizations may also introduce dependencies. Burke and Torczon [1993] described algorithms that reduce the number of compilations in such situations. Here we assume that the compiler does not produce any dependencies not captured by name use.

Third, our definition of declaration-level *USES* simplifies the treatment of subordinate declarations, such as enumeration literals, record components, and formal parameters. Rather than performing a dependency analysis down to subordinate declarations, we treat each use of a subordinate declaration as the use of the parent declaration. Our results indicate that a more thorough analysis would achieve only minor additional savings in recompilations.

Fourth and finally, overloading of identifiers is handled properly. The overloading problem is a special case of the more general problem that adding or changing declarations can lead to semantic errors in unchanged units. The analyzer does not, however, detect semantic errors caused by overloading. The resulting inaccuracy is extremely small, because overloading was rare and most of the configurations available at the end of a day may be assumed to be correct.

Based on the above definitions and assumptions, the analyzer builds up symbol tables recording the *USES* relations and the number of lines in the units. For any two successive configurations, the analyzer pairs corresponding specification units, determines added, deleted, and modified declarations for each pair, and then computes the various compilation costs by consulting the *USES* relations. The recompilation set, that is, the set of recompiled units, is computed as follows:

Cascading recompilation. The recompilation set is simply determined by computing the reflexive, transitive closure of the touched units under unit-level *USES*.

Surface change. The recompilation set consists of all touched units plus the units in the reflexive, transitive closure of the semantically changed units. Including all touched units actually underestimates the potential savings. However, only 3 percent of the recompiled units are touched but semantically unchanged, so this error is slight.

Cutoff recompilation. The recompilation set of this method is approximated closely by observing that for semantically changed units, cutoff recompilation compiles one dependency link further than smart recompilation. Thus, we first compute the smart recompilation set for the semantically changed units (see below), then supplement this set with those units that directly import any unit of this set, and, finally, add the touched, but semantically unchanged units.

This computation is correct under two assumptions: (1) The recompilation of any unit that is not affected by a given change produces output identical to the output before the change (otherwise, cutoff recompilation would hardly work at all); (2) the recompilations enforced by smart recompilation are really necessary; that is, smart recompilation causes no redundant recompilations. Unfortunately, this assumption is not always met in practice. Smart recompilation may cause a redundant compilation if a unit uses attributes of a changed declaration that happen to be invariant despite the change. For example, the size attribute of a record type might remain invariant if the order of its fields changes. If a unit relies only on the size attribute, smart recompilation would cause an unnecessary recompilation. However, when evaluating our results in the last section, we shall see that the number of those cases is negligibly small.

Semi-smart recompilation. The semi-smart recompilation technique does not work well for Ada, since the assumptions about nonpropagation of the relevant classes of changes are not met. For example, new declarations may cause visibility errors, and thus, the directly dependent units must be recompiled. Ada also permits the use of attributes of variables and subproprograms when defining new types. In general, the performance of semi-smart recompilation lies between surface change and smart recompilation. For Ada, we estimate it to be close to that of cutoff recompilation, since over half of all changes involve procedure headers, which tend to propagate dependencies only to the directly calling units.

Smart recompilation. The recompilation set is computed exactly. It consists of the set of changed units plus the units containing declarations that use a changed declaration directly or indirectly.

Attribute recompilation. Only lower and upper bounds can be given. The upper bound is smart recompilation; the lower bound is the set of changed units.

Smartest recompilation. The recompilation set is the set of changed units, excluding all specification units. Smartest recompilation is infeasible for Ada, but the figures serve as lower bounds.

2.3 Estimating the Cost of Environment Processing

For each selective recompilation technique investigated, the analyzer sums the number of source environment lines and units inhaled. It also determines the savings achievable with environment pruning and selective embedding. The gain of environment pruning is easy to compute by analyzing declaration-level *USES*. Selective embedding is measured by Borison's RUV (Ratio of Use to Visibility) metric. This metric is computed for a given set S of n declarations as follows:

$use(i)$ = number of compilation units where declaration i is used,

$visible(i)$ = number of compilation units where declaration i is visible,

$$RUV(S) = \frac{\sum_{i=1}^{n} use(i)}{\sum_{i=1}^{n} visible(i)}.$$

The set S may be all or a subset of the declarations in specification units of a given program. RUV for the entire program is an indicator for the performance of selective embedding, since it is the fraction of actually used declarations.

We computed the RUV metric for the declarations in the specifications of the last configuration and for the declarations in all changed specifications. Furthermore, we broke RUV down into separate declaration classes (subprograms, variables, types, constants, etc.).

3. RESULTS

This section reports on the quantitative results with little interpretation; a more thorough interpretation appears in the succeeding section.

3.1 Size and Growth

The first revision of SMS was checked into the archive in September 1985, and the last one in March 1989. With the last revision, the development of the alpha test release of SMS was complete. Maintenance data are not available, because further development was halted. The last release of SMS consists of about 140,000 lines of code (140 KLOC), of which CM and VM contain about a total of 63 KLOC (all counts exclude comments and empty lines). Details about the number of units, revisions, and configurations are given in the Appendix.

Our analyzer successfully processed 687 out of the 706 configurations. The rest contain syntax errors.

Figures 1 and 2 illustrate the growth of CM and VM combined. Figure 1 plots the average number of revisions per unit over time. The gap between specification and implementation units widens with the age of the system, apparently as the architecture stabilizes. Figure 2 shows the number of

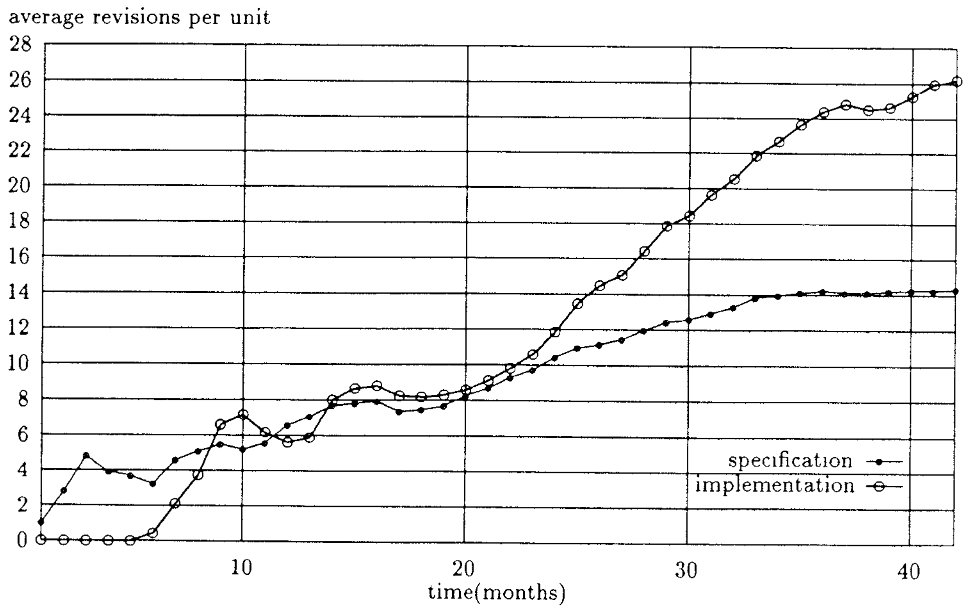


Fig. 1. Average number of revisions per compilation unit in a configuration.

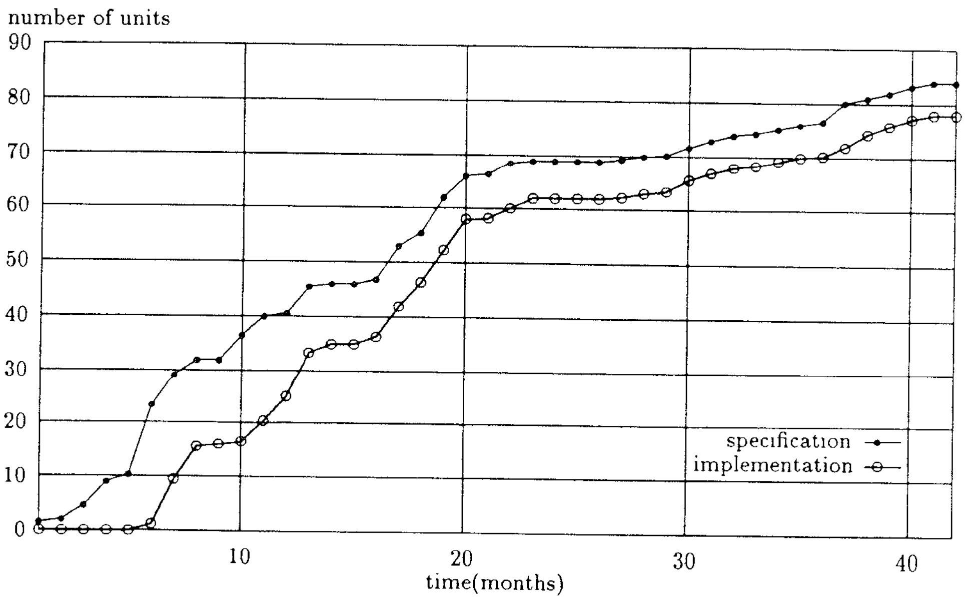


Fig. 2. Number of compilation units in a configuration over time.

compilation units per configuration. Both graphs separate specifications and implementations, and plot the variables in one-month intervals. Implementations trail specifications in Figure 2, with the gap narrowing over time. Drops in the averages in Figure 1 occur when new units, with no revisions initially, are added (compare Figure 2).

Concerning the mixture of changes, we observed that 39 percent of the daily changes were pure implementation changes (i.e., only implementations were revised that day) and that 16 percent were pure specification changes (i.e., only specifications were revised). Most of the pure specification changes occurred at the beginning of the development. Pure implementation changes increased in number later on. Overall, about 37 percent of all unit changes were specification changes.

In the last configuration of March 1989, the average size of a specification unit was 111 lines of code; the average size of an implementation unit was 706 lines. The average unit contained an additional 236 lines of pure comments. This number was almost identical for specifications and implementations. Thus, the ratio of pure comment lines to the total number of source lines was about 0.70 for specifications and 0.25 for implementations.

3.2 Compilation Costs

Table I summarizes the cumulative cost of compilation and environment handling over all configurations analyzed, measured in KLOC processed. The table contains columns for cascading recompilation, surface change, cutoff recompilation, smart recompilation, and smartest recompilation. An extra column, labeled "Changed", tallies the number of lines to be recompiled or inhaled if only the changed units were recompiled. This column is the lower bound for methods that process all changed units, such as smart and attribute recompilation. The column for smartest recompilation is unrealistic, because it is infeasible to add type inference to Ada so that environment reading can be totally omitted. However, this column provides a lower bound on pure compilation work (without environment reading).

When compiling a given unit, we distinguish the lines that belong to that unit from the lines that the unit inhales from the environment (first and second rows, resp.); their sum appears in the third row. The fourth row provides the source lines analyzed, to determine which compilations to omit. This row is needed to estimate the overhead of the various methods and is not included anywhere else in the table. Note that the numbers in this row are small fractions of the total lines processed by cascading recompilation.

The fifth row gives the ratio of inhaled to recompiled lines; this number decreases from 1.9 for cascading recompilation to 1.4 for smart recompilation. This effect is easily explained: The units that are not compiled are those that depend on the touched units. These units are further down in the dependency hierarchy and therefore have a larger environment. Consequently, removing them from the compilation sets makes the environment smaller on average, and the average number of lines processed per compilation unit decreases from 1.42 to 1.31 KLOC.

Table I. Cost of Compilation and Environment Handling in KLOC (all configurations)

Method	Cascade	Surface	Cutoff	Smart	Changed	Smartest
Σ KLOC recompiled	3,146	2,784	2,530	1,879	1,621	1,621
Σ KLOC inhaled	6,063	5,193	4,247	2,653	2,124	0
Σ KLOC total	9,209	7,977	6,777	4,532	3,745	1,621
Σ KLOC analyzed	0	1,621	180	135	0	0
Inhaled/recompiled	1.9	1.9	1.7	1 4	1.3	0
Mean KLOC processed per unit	1.42	1.39	1.39	1.31	1.27	0.55
Lines saved relative to cascade	0 %	13 %	26 %	51 %	59 %	82 %
Σ KLOC inhaled with environment pruning	4,892	4,303	3,808	2,573	2,100	0
Lines saved relative to Σ KLOC total with environment pruning	13 %	11 %	6 %	2 %	1 %	0 %

The entries showing the savings relative to cascading recompilation are perhaps the most interesting. Surface change achieves savings of 13 percent; cutoff recompilation, 26 percent; and smart recompilation, 51 percent. These three methods provide the same type safety as cascading recompilation and face no additional, deferred costs. Smart recompilation's savings are close to the lower bound of changed (59 percent).

Smartest recompilation compiles the theoretical minimum: only the changed compilation units. However, recall that smartest recompilation gives up compile-time type safety, and the cost for consistency checking during linking is not included.

As explained previously, semi-smart recompilation would be close to cutoff recompilation. The savings of attribute recompilation lie somewhere between smart recompilation and Changed, that is, between 51 and 59 percent.

The last two rows of Table I show the savings achievable with environment pruning. Depending on the quality of the selective recompilation method, the savings range from 13 to 2 percent. On our data, environment pruning with smart recompilation would have saved only 80 KLOC over a three-year period.[5]

The bar chart of Figure 3 illustrates the cost of compilation in lines processed. The lower part of the bars represents the touched units; the upper part, the directly and indirectly dependent untouched units. Smart recompilation compiles only slightly more than the set of changed units, without sacrificing static type safety.

Table II shows the same costs as Table I, but measured in number of units rather than in number of lines. For instance, the first line provides the actual number of compilation runs. Note that the ratio of inhaled versus compiled units is much higher than for lines, because specification units are smaller than implementation units. The savings achievable with environment pruning also appear more pronounced, for the same reason.

[5]Environment pruning figures are calculated as if *all* external symbols were qualified and, hence, represent a best case for Ada.

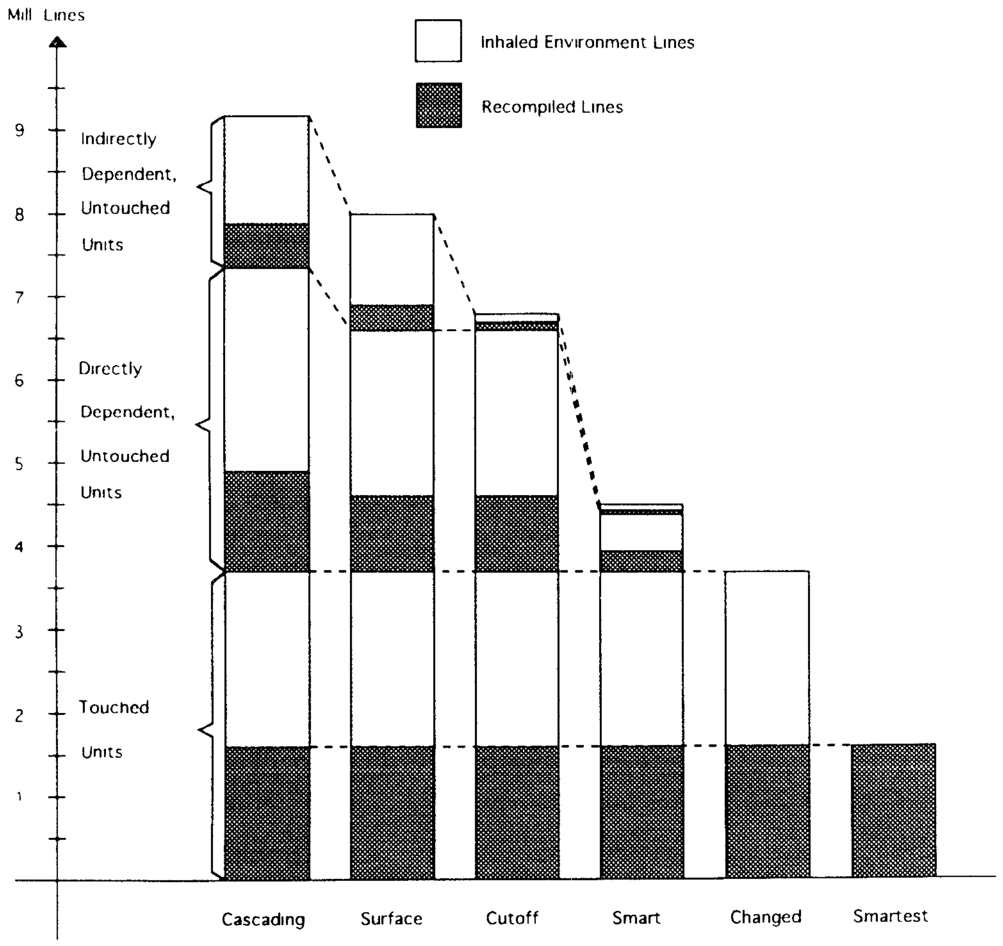


Fig. 3. Compilation costs in lines processed.

Table II. Cost of Compilation and Environment Handling in Units (all configurations)

Method	Cascade	Surface	Cutoff	Smart	Changed	Smartest
Σ units recompiled	6,471	5,737	4,859	3,449	2,955	2,955
Σ units inhaled	69,515	59,435	47,364	29,679	23,277	0
Σ units total	75,986	65,172	52,228	33,128	26,232	2,955
Σ units analyzed	0	2,955	1,800	1,200	0	0
Inhaled/recompiled	10.7	10.4	9.7	8.6	7.9	0
Saved units relative to cascade	0 %	14 %	31 %	56 %	65 %	96 %
Σ units inhaled with environment pruning	48,700	43,000	37,620	25,753	20,989	0
Units saved relative to Σ units total with environment pruning	27 %	25 %	19 %	12 %	9 %	0 %

Table III. Ratio of Use to Visibility (RUV)

Declaration class	All	Constants	Types	Variables	Exceptions	Subprograms	Inner packages
RUV of changed specifications	0.16	0.08	0.30	0.00	0.10	0.14	0 01
RUV of specifications in final configuration	0.20	0.09	0.45	—	0 16	0.20	—

We shall base further comparisons on lines of code, since savings expressed in this measure are more conservative than in compilation units. In case the cost of opening and closing units actually dominates the input cost, then the potential savings will be higher.

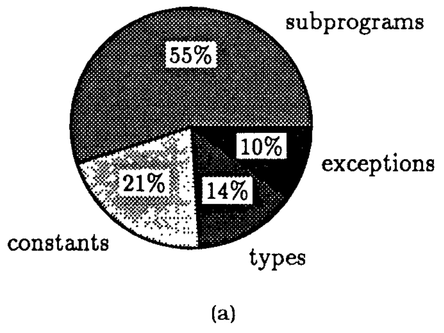
Table III shows the RUV for the declarations in all changed specifications and in the specifications of the last configuration. In both cases the RUV is also computed for the various declaration classes separately. The results show that RUV in the last configuration is slightly larger. This may be explained by the observation that the use of stable declarations tends to increase over time. Compared to the overall RUV, the RUV values for types are considerably larger, while the RUV values for constants are considerably smaller. Hence, the importing units use types relatively more often and constants less. The RUV fur subprograms corresponds closely to the total RUV, since subprograms strongly dominate the declarations (see Section 3.3). Since variables and inner packages occur rarely, the computed RUV values are unreliable. Tasks do not occur at all.

The total RUV value of 0.2 in the last configuration is rather small and means that a single symbol defined in a package specification is used in only 20 percent of those units in which it is visible. Thus, selective embedding can save up to 80 percent in symbol table space for the environment, and the same percentage in environment processing time, provided symbols can be looked up directly. Recall that the environment is between 1.4 and 1.9 times the size of the compilation unit. Thus, the 80 percent savings in environment handling translates to 46 percent total input savings for smart recompilation and 52 percent for cascading recompilation. However, these input savings may be difficult to translate into equivalent time savings. First, random access to symbols may carry some overhead, and second, the compiler may process lines in the environment faster than lines in implementation units. These issues are taken up in the next section.

3.3 Distribution of Changes and Program Structure

Concerning the distribution of changes and the program structure, we observe the following:

—Fifteen percent of the touched specifications are not semantically modified (164 out of 1,097). In the touched specifications, about 18 percent of the contained declarations were actually changed. Figure 4a shows the distri-



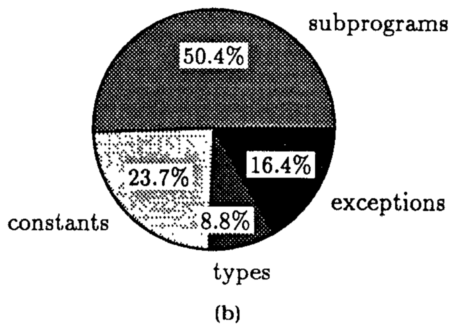


Fig. 4. (a) Distribution of changes. (b) Distribution of declarations in specification units.

bution of changed declarations. In relation to their total occurrence in all changed specifications, subprograms, constants, and types were changed slightly more often, whereas exceptions were changed considerably less often.

—In the last configuration of March 1989, the average specification unit exported 27.4 declarations. See Figure 4b for their distribution. Inner packages and variables were exported extremely rarely, and tasks never.

—In the last configuration of March 1989, a specification is used by about 20–21 other units (6 directly, 14–15 indirectly). The average dependency depth of the uses hierarchy is 2.3. This depth is the average length of a used-by path, starting from an arbitrary specification in the dependency hierarchy.

For comparing the distribution of declarations and the depth of the dependency structure, we analyzed another Ada system of approximately two-thirds the size of CM and VM together. The system was developed by the "Gesellschaft für Mathematik and Datenverarbeitung" (GMD) as part of an Ada compiler. CM and VM consist of nearly the same number of units as the GMD system. A specification unit of CM and VM contains twice as many lines of code as a specification of the GMD system; an implementation contains a third more. The GMD system has no revision history at all; however, for the present comparison, a history is not needed.

Despite the difference in size, the average number of declarations per specification is about the same in both systems. As for the distribution of declarations, 26 percent of declarations in the GMD specifications are variables, while there are hardly any variables exported by CM and VM specifications. Constants and exceptions are used more often in CM and VM (23.7 percent versus 11.8 percent and 16.4 percent versus 1.8 percent). The percentages for procedures, types, and packages are nearly the same.

The dependency structure among the compilation units of the GMD system is considerably deeper; its average depth is nearly five. However, this is to be expected, since the analysis of the GMD system is complete, while the analysis of SMS omitted the subsystem SUPPORT and the operating-system interface. Inclusion of these systems would result in a deeper dependency structure for SMS, but would also more than double total size.

4. INTERPRETATION OF RESULTS

Saving one-half to three-quarters of compilation work is significant, especially for slow compilers and large systems. However, the savings are offset by the overhead of additional analysis. This section relates savings to overhead and discusses other factors influencing total compilation turnaround time, such as parallel compilation and linking.

So far, all measures for compilation work were in terms of input size. The following formula relates input to compilation time:

$$t_M = IL_M \times c_I + EL_M \times c_E + O_M, \tag{1}$$

where

t_M = compile time for selective recompilation method M,
IL_M = kilo lines compiled by method M,
EL_M = kilo environment (specification) lines processed by method M,
c_I = time for compiling 1 KLOC of an implementation unit,
c_E = time for processing 1 KLOC of a specification unit,
O_M = overhead caused by method M.

Eq. (1) will be used to compute the cumulative compilation time (all runs) under the various methods using data from Table I. Compiler speed is modeled with two constants, c_I and c_E, because environment processing is often faster than compiling. We assume that

$$c_E \leq \frac{c_I}{2} \tag{2}$$

for the following discussion, but the reader may substitute a different ratio.

4.1 Overhead of Selective Recompilation

Cascading recompilation is the baseline. The overhead of surface change is

$$\begin{aligned} O_{Surf} &= A_{Surf} \times c_{Surf} \\ &\leq \frac{1621 c_E}{5} \\ &\leq 324 c_E \\ &\leq 162 c_I, \end{aligned} \tag{3}$$

where A_{Surf} is the amount of input analyzed by surface change, measured in KLOC (see the fourth row of Table I), and c_{Surf} is the average time for comparing a pair of units of 1 KLOC each for identity, while ignoring whitespace and comments. Because the comparison is only lexical, we can assume that $c_{Surf} \leq c_E/5$. With Eq. (2), surface change's overhead is less than 3 percent of $t_{Cascade}$.

The cutoff recompilation method requires only the comparison of compiler output:

$$\begin{aligned} O_{Cut} &= A_{Cut} \times c_{Cut} \\ &\leq \frac{180 c_E}{5} \\ &\leq 36 c_E \\ &\leq 18 c_I. \end{aligned} \qquad (4)$$

A_{Cut} is the number of lines (in KLOC) in recompiled specifications, c_{Cut} is the time for a bytewise comparison of the compiler outputs produced from a pair of specification units of 1 KLOC each. Note that the first three rows of Table I already account for all compilations that occurred, so we only need to consider file comparisons. Compiler output may be more voluminous than input, but bytewise comparison is extremely fast, so we can assume that $c_{Cut} \leq c_E/5$. With (2), O_{Cut} is 0.3 percent of $t_{Cascade}$.

The overhead of smart recompilation consists of computing deltas and intersecting symbol lists:

$$\begin{aligned} O_{Smart} &= A_{Smart} \times c_{s1} + I_{Smart} \times c_{s2} \\ &\leq 135 c_E + \frac{938 c_E}{5} \\ &\leq 323 c_E \\ &\leq 161 c_I. \end{aligned} \qquad (5)$$

A_{Smart} is the size of recompiled specifications in KLOC, and c_{s1} is the time of computing a delta between a pair of specification units, each 1 KLOC long. A delta consists of the identifiers of added, deleted, and changed declarations. Table I provides the value for A_{Smart}. Computing deltas is fast, since it involves merely building syntax trees and comparing them structurally. It is safe to assume that $c_{s1} \leq c_E$.

The second component of the overhead formula is more difficult to quantify. c_{s2} is the time for processing 1,000 symbols in an intersection, while I_{Smart} indicates how many symbols (in thousands) are processed in this manner. Table II indicates that 1,200 deltas are computed, while Section 3.3 states that the average number of dependent units is 21. Thus, each delta must be loaded once and then intersected with the use lists of 21 other units. The size of the delta is minimal: According to Section 3.3, the average specification unit exports 27 symbols, of which only 18 percent are changed. So a delta is on average only 5 symbols long. Use lists are also short. Table II indicates that each compilation has an environment of 8.6 units. Each unit provides 27 symbols, of which 16 percent are actually used (see Table III). Thus, we have 37 symbols per use list. The total number of symbols processed is thus $1{,}200 \times (5 + 21 \times 37) \approx 938 \times 10^3$. It is reasonable to assume that processing a symbol (basically, hashing it against 5 preloaded ones) takes much less time than inhaling a line. Hence, we assume that $c_{s2} \leq c_E/5$. With Eq. (2), smart recompilation's overhead is 2.6 percent of cascade recompilation's compile time.

Smartest recompilation incurs no compile-time overhead, if we assume that the time for performing type inference is negligible. Link-time overhead is unavailable.

4.2 Overhead of Environment Processing

Environment pruning built into the compiler carries no overhead, because it simply skips wholly unused environment units. A separate pruning program such as Incl takes time that is some fraction of big bang, but it is run infrequently, so its cost is amortized over many compilation runs.

Selective embedding might cut environment input by 80 percent,[6] but its run time is difficult to estimate. The most obvious approach of implementing the method is to use a database, but it is unclear how update and access time in the database compare to c_E.

The size of the environment indicates that maintaining a resolved environment per compilation unit carries a substantial space penalty, even if environments are compressed. However, the potential sharing of resolved environments among compilation units was not investigated. It is also unclear how much time saving compression can yield, because we did not determine how often an environment is read before its source changes.

4.3 Compile-Time Estimates

Table IV provides the estimated total compilation time over all configurations analyzed. It is computed from Eqs. (1)–(5) and the data from Table I. Overhead is included, as is the assumption that environment reading is twice as fast as compiling. The first row provides cumulative times for the entire development history in multiples of c_I with full environment reading, whereas the second row provides percentages relative to cascading recompilation. The third and fourth rows are similar, but with environment pruning applied (row 8 instead of row 2 of Table I). The bottom two rows of Table IV assume that the environment input has been reduced by selective embedding, but not with pruning (20 percent of Table I, row 2). For lack of an estimate, we include no overhead for selective embedding. This may be an unrealistic assumption. However, the reader may recompute the last two rows by using a less favorable reduction rate in order to account for the overhead. The reader may also use a different c_I/c_E ratio and different cost factors for computing O_M.

Several observations can be made. Smartest recompilation is the fastest method among all combinations, potentially compiling at four times the speed of cascading recompilation. However, the costs for type inference, linking, and programmer time spent in resolving delayed error messages are excluded.

[6] We are assuming that the number of declarations in environment units is proportional to input lines.

Table IV. Compile Time Measured in Multiples of c_I (all configurations)

Method	Cascade	Surface	Cutoff	Smart	Changed	Smartest
Full environment (c_I)	6,178	5,543	4,672	3,367	2,683	1,621
Relative to cascade (full environment)	100 %	90 %	76 %	54 %	43 %	26 %
Environment pruning (c_I)	5,592	5,098	4,452	3,327	2,671	1,621
Environment pruning relative to cascade (full environment)	91 %	83 %	72 %	54 %	43 %	26 %
Selective embedding (c_I)	3,752	3,465	2,973	2,305	1,833	1,621
Selective embedding relative to cascade (full environment)	61 %	56 %	48 %	37 %	30 %	26 %

Smart recompilation is not much slower than processing the changed units and has none of the problems of smartest recompilation.

On our data, environment pruning has a small to negligible effect: The more selective the compilation method, the smaller is the gain. However, Vo and Chen [1992] reported that Incl's environment pruning removed 41–71 percent of the environment units of three systems, saving between 12 and 36 percent of compilation time. A plausible explanation is that SMS has not undergone any maintenance and, thus, has a clean architecture.

A properly implemented version of selective embedding might cut the environment input by 80 percent. Note that this figure applies to a system with a clean dependency architecture, where unused environment units are rare. So this reduction may be achievable even after environment pruning has occurred. However, random access to declarations in the environment may be time consuming, wiping out much of the gain of selective embedding.

From this discussion we conclude that further substantial reductions in compilation work can only come from partially compiling the touched units, as implemented in the Rational environment and some language-oriented editors.

4.4 Total Turnaround Time

Selective recompilation and efficient environment processing are but two ingredients for achieving the goal of fast turnaround after changes. Other important ingredients are the programming language itself, the basic compiler speed, parallelization of the compilation process, and the speed of the linker.

Parallelization can shorten the time to rebuild a system considerably. All of the selective recompilation methods discussed here can be combined with parallel compilation. The only drawback is that selective recompilations might reduce the amount of parallelism available. However, Leblang and Chase [1987] reported that on a network of workstations the number of compilations that a single scheduler can spawn concurrently peaks around 10. Even if this peak improves by a factor of three, lack of parallelism is not going to be an issue when hundreds of units need to be compiled.

Linking can also consume a substantial amount of time, especially if large configurations must be completely relinked after every change. Incremental linkers reduce linkage time by patching new object modules into already linked code [Linton and Quong 1989]. With such linkers, savings produced by selective recompilation translate into nearly the same savings in linkage time.

Industrial-strength compiling systems should therefore strive for the best available techniques in the areas of compiler speed, selective recompilation, environment processing, and incremental linking.

4.5 Other Observations

Attribute recompilation has not been discussed yet. Its performance lies between that of smart recompilation and changed. It seems highly unlikely that the latter case can be attained, because not all changes will be limited to attributes that are unused; otherwise, there would hardly be any point in making the change or exporting the changed declarations. Furthermore, only types and variables have compile-time attributes that may remain unchanged when the corresponding declaration changes. Recall that types make up a mere 14 percent of both declarations and changes, and there were practically no exported variables. Thus, we conjecture that only a small fraction of the additional savings of attribute recompilation can be realized. The bookkeeping involved in attribute analysis seems quite expensive and might easily wipe out the small potential gain.

One of the surprises in this study is that specifications change relatively frequently, especially during the early development. The high frequency makes selective recompilation strategies all the more important, because changes of specifications often cause far-reaching recompilations. It also suggests that a separation between specification and implementation units is perhaps not worth the bother, provided the language offers an adequate encapsulation mechanism in some other way.

Another surprise is that the dependency graph was quite shallow, with a wide fanout. This observation might not generalize to other systems. First, the flatness of SMS might have been a conscious design decision. Second, we did not analyze the dependencies into the underlying software layer. Obviously, a larger system would exhibit a deeper dependency graph. Note, however, that not all changes occur at the roots of the dependency graph. For bottom-up development, levels closer to the leaves are likely to change more frequently. Thus, we expect that the average change propagates only a few levels in the dependency graph before it reaches leaves. In other words, the average propagation depth might be relatively small, even for large systems.

Our study was carried out for batch compilations. We believe that similar results would hold for interactive compilations. The number of total compilations would probably rise, while the number of changed units per configuration would drop. However, there is no reason to assume that the dependency graph and the relative frequency of changes to specifications and implementations would differ greatly.

There exist few other studies in this area. Besides our earlier work [Adams et al. 1989], the only systematic approach known to us is Borison's [1989] Ph.D. thesis. Borison analyzed the evolution of a relatively small C program that consisted of 28 specifications (header files), 26 implementations, and a total of 12,000 lines of code. The history contains 190 revisions over a period of about half a year. With respect to selective recompilation, she reports the following savings, measured in lines relative to cascading recompilation; surface change—6.5 percent; smart recompilation—51.5 percent; and attribute recompilation—58.3 percent. These are quite close to our results. The difference in surface change could be caused by less documentation in Borison's data. In SMS, about one-third of the source lines are comments.

Eidnes et al. [1989] reported that change detection in CHILL consumes about 3–4 percent of compilation time, which matches our estimate of the overhead of smart recompilation.

Conradi and Wanvik [1985] estimated that environment reading may consume 30–50 percent of compilation time. They also confirmed RUV as below 0.2. Borison's thesis reports the RUV for three C programs (0.29, 0.19, and 0.14) and three Ada programs (0.42, 0.14, and 0.07). The values vary considerably, but are all rather small. The only larger value of 0.42 is for the smallest of the six programs, consisting of merely 4,300 lines of code. It is easy to see that for small programs the RUV value tends to be large, since the interface of a module is used by a small number of units and should therefore contain little redundancy.

5. CONCLUSION

This paper has compared several techniques of selective recompilation and environment processing. By analyzing the three-year history of a realistic, industrial software system, a conclusive comparison was possible. Smart recompilation and selective embedding can each cut input nearly in half, potentially doubling to tripling compilation speed and doubling linking speed.

Environment pruning could not be assessed. Its effectiveness depends strongly on the dependency architecture of a given system.

Further studies should quantify the gains possible when recompiling at the declaration and statement level. The fanout and depth of the dependency structure and the distribution of declarations and changes would be of interest. Corroborating these results with different languages would be helpful. Finally, systems formulated in object-oriented or functional languages may have different properties from those observed here.

APPENDIX

Table V gives an overview of the number of units, revisions, and configurations in the subsystems CM and VM. In the surveyed time period of about three years, CM was changed on 372 days and VM on 334 days. On average, CM was modified on 10 days each month, and VM on 9 days. Altogether, 3,076 revisions were deposited (counting multiple deposits of the same unit on the same day as one). This means that 4.4 units were touched on average for each change of one of the two subsystems. The 3,076 revisions consist of 1,137 specification units and 1,939 implementation units. Surprisingly, speci-

Table V. Overview of Subsystems CM and VM

	Units in last configuration			Unit revisions			Configurations		
System	All	Specification	Implementation	All	Specification	Implementation	All	Analyzed Successfully	Period
CM	85	45	40	1,642	566	1,086	372	368	3/86–3/89
VM	76	39	37	1,424	571	853	334	319	9/85–3/89
CM + VM	161	84	77	3,076	1,137	1,939	706	687	9/85–3/89

fication units were revised rather frequently: 37 percent of all changed units were specifications.

The analyzer successfully processed 687 out of the 706 configurations (368 for system CM and 319 for system VM). The remaining configurations (4 in system CM and 15 in system VM) contain syntax errors. Altogether, 79 (or about 2 percent) of the individual units (50 in system CM and 29 in system VM) contain syntax errors.

ACKNOWLEDGMENTS

Professor Hans-Helmut Nagel encouraged this project. Many thanks go to the staff at Siemens Corporate Research, who made the data available and also generously supplied software, hardware, and travel grants. Ellen Borison's Ph.D. thesis and discussions with her had a deep influence. Reidar Conradi and an anonymous referee provided detailed comments. Valuable suggestions and generous support by Rita Altucher, Helmut Faasch, William Hasling, Pat Vroom, and Michael Wagner are greatly appreciated.

REFERENCES

ADAMS, R., WEINERT, A., AND TICHY, W. 1989. Software change dynamics or half of all Ada compilations are redundant. In *Proceedings of the 2nd European Software Engineering Conference* (Warwick, U.K., Sept. 1989). Springer-Verlag, New York, 203–221.

BORISON, E. 1989. Program changes and the cost of selective recompilation. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., July.

BURKE, M., AND TORCZON, L. 1993. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Trans. Program. Lang. Syst. 15*, 3 (July), 367–399.

CASHIN, P. M., JOLIAT, M. L., KAMEL, R. F., AND LASKER, D. M. 1981. Experience with a modular typed language. In *Proceedings of the 5th International Conference on Software Engineering* (March 9–12, 1981). IEEE Computer Society Press, New York, 136–143.

CCITT 1988. CCITT high level language (CHILL) language definition, 1988. CCITT Recomm. Z.200, ITU, Geneva, Switzerland.

CONRADI, R., AND WANVIK, D. H., 1985. Mechanisms and tools for separate compilation. Tech. Rep. 25/85, The Norwegian Institute of Technology, Univ. of Trondheim, Trondheim, Norway.

DAUSMANN, M. 1985. Informationsstrukturen und Verfahren für die getrennte Übersetzung von Programmteilen. GMD-Bericht 155, R. Oldenburg Verlag, Munich, Germany.

EIDNES, H., HALLSTEINSEN, S. O., AND WANVIK, D. H. 1989. Separate compilation in CHIPSY. *ACM SIGSOFT*, *Softw. Eng. Notes 16*, 7 (Nov.), 42–45.

FEILER, P. H., DART, S. A., AND DOWNEY, G. 1988. Evaluation of the Rational environment. Tech. Rep. CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie-Mellon Univ., Pittsburgh, Pa.

FELDMAN, S. I. 1979. Make—A program for maintaining computer programs. *Softw. Pract. Exper. 9*, 3 (Mar.), 255–265.

FOSTER, D. G. 1986. Separate compilation in a Modula-2 compiler. *Softw. Pract. Exper. 16*, 2 (Feb.), 101–106.

GUTKNECHT, J. 1986. Separate compilation in Modula-2: An approach to efficient symbol files. *IEEE Softw. 3*, 6 (Nov.), 29–38.

KAMEL, R. F. 1987. Effect of modularity on system evolution. *IEEE Softw. 4*, 1 (Jan.), 48–54.

LEBLANG, D. B., AND CHASE, R. P., JR. 1987. Parallel software configuration management in a network environment. *IEEE Softw. 4*, 6 (Nov.), 28–35.

LEHMANN, M. M., AND BELADY, L. A., EDS. 1985. *Program Evolution: Processes of Software Change*. APIC Studies in Data Processing, vol. 27. Academic Press, New York.

LEVY, M. R. 1984. Type checking, separate compilation and reusability. *SIGPLAN Not.* (ACM) *19*, 6 (June), 285–289.

LINTON, M. A., AND QUONG, R. W. 1989. A macroscopic profile of program compilation and linking. *IEEE Trans. Softw. Eng. 15*, 4 (April), 427–436.

MITCHELL, J. G., MAYBURY, W. AND SWEET, R. 1978. Mesa language manual. Tech. Rep., Xerox Palo Alto Research Center, Palo Alto, Calif., Feb.

OLSSON, R. A., AND WHITEHEAD, G. R. 1989. A sımple technique for automatic recompilation in modular programming languages. *Softw. Pract. Exp 19*, 8 (Aug.), 753–773.

RAIN, M. 1984. Avoiding trickle-down recompilation in the Mary2 implementatıon. *Softw Pract. Exper. 14*, 12 (Dec.), 1149–1157.

ROSS, G. 1987. Integral C—A practical environment for C programming. *SIGPLAN Not.* (ACM) *22*, 1 (Jan.), 42–48.

SCHWANKE, R. W., AND KAISER, G. E. 1988. Technical correspondence: Smarter recompilatıon. *ACM Trans. Program. Lang. Syst. 10*, 4 (Oct.), 627–632.

SCHWANKE, R. W., COHEN, E. S., GLUECKER, R , HASLING, W. M., SONI, D. A., AND WAGNER, M. E. 1989. Configuration management in BiiN SMS. In *Proceedıngs of the 11th Internatıonal Conference on Software Engıneering* (Pittsburgh, Pa., May 1989), pp. 383–393.

SHAO, Z , AND APPEL, A. W. 1993. Smartest recompilation. In *20th Annual ACM SIGPLAN–SIGACT Symposium on Prınciples of Programmıng Languages* (Charleston, S.C., Jan. 1993). ACM, New York, 439–450.

TICHY, W. F. 1981 Software development control based on module interconnection. In *Software Development Envıronments*. A. I. Wasserman, Ed. IEEE Computer Society Press, New York, 272–284. (Originally published in *Proceedings of the 4th Internatıonal Conference on Software Engıneering*. IEEE, New York, Sept. 1979.)

TICHY, W. F 1985. RCS—A system for version control *Softw Pract. Exper. 15*, 7 (July), 637–654.

TICHY, W. F. 1986. Smart recompilation. *ACM Trans. Program. Lang Syst. 8*, 3 (July), 273–291.

U.S. DEPARTMENT OF DEFENSE. 1983 Reference manual for the Ada programming language. ANSI/MIL-STD 1815 A-1983, 2, U.S. Dept. of Defense, Washington, D C.

VO, K.-P., AND CHEN, Y.-F. 1992. Incl: A tool to analyze include files. In *Summer 1992 USENIX Conference* (San Antonio, Tex., June), pp. 199–208.

WIRTH, N. 1985. *Programmıng ın Modula-2*. Springer-Verlag, New York.

WIRTH, N , GEISSMANN, L., HOPPE, J., JACOBI, C., KNUDSEN, S. E., AND WINIGER, W. 1982. Lilith handbook: A guıde for Lilith users and programmers Tech. Rep , Institut für Informatik der ETH Zürich, Switzerland.

Received February 1993; revised and accepted September 1993