An Empirical Analysis of C Preprocessor Use

Michael Ernst Greg J. Badros* David Notkin

Technical Report UW-CSE-97-04-06
Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350 USA
{mernst,gjb,notkin}@cs.washington.edu
22 April 1997

Abstract

The C programming language is intimately connected to its macro preprocessor. This relationship affects, indeed generally hinders, both the tools (compilers, debuggers, call graph extractors, etc.) built to engineer C programs and also the ease of translating to other languages such as C++. This paper analyzes 27 packages comprising 1.2 million lines of publicly available C code, determining how the preprocessor is used in practice. We developed a framework for analyzing preprocessor usage and used it to extract information about the incidence of preprocessor directives, the frequency of macro use and redefinition, the purposes of macros (in terms of both definitions and uses), and expressibility of macros in terms of other C or C++ language features. We particularly note data that are material to the development of tools for C or C++, including translating from C to C++ to reduce preprocessor usage. The results are of interest to language designers, tool writers, programmers, and software engineers.

1 Introduction

The C programming language [KR88] is intimately connected to its macro preprocessor, Cpp [HS95, Ch. 3]. C is incomplete without the preprocessor, which supplies essential facilities such as file inclusion, definition of constants and macros, and conditional compilation. While disciplined use of the preprocessor can reduce programmer effort and improve portability, performance, or readability, Cpp also lends itself to arbitrary source code manipulations that complicate understanding of the program by both software engineers and tools. The designer of C++, which shares C's preprocessor, also noted these problems: "Occasionally, even the most extreme uses of Cpp are useful, but its facilities are so unstructured and intrusive that they are a constant problem to programmers, maintainers, people porting code, and tool builders." [Str94, p. 424]

*Supported by a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author, and do not necessarily reflect the views of the National Science Foundation.

Package	Version	Physical lines	NCNB lines	Description
bash	1.14.7	68119	46598	Command shell
bc	1.03	7438	5177	Desktop calculator
bison	1.25	11542	7765	Parser generator
cvs	1.9	79440	53618	Revision control system
dejagnu	1.3	65885	40270	Testing framework
flex	1.3	18943	13284	Scanner generator
fvwm	19.34	55811	42745	Window manager
g77	2.5.3	134046	99690	Fortran compiler
gawk	2.0.43	27501	18674	GAWK interpreter
genscript	0.5.18	12049	8166	Text-to-PostScript converter
ghostview	2.15.6	11348	8711	PostScript previewer
glibc	2.7.2.1	128337	71453	C library
gnuchess	1.3.2a	17774	14574	Chess player
gnuplot	1.5	38209	29582	Graph Plotter
groff	1.09.1	69334	60502	Text formatter
gs	2.6.2	77787	56378	PostScript interpreter
gzip	4.0.pl77	9076	5787	File compressor
m4	3.50.1.17	16767	10402	Macro expander
perl	5.003	65210	57152	Perl interpreter
plan	1.10	23838	18885	Schedule planner
python	1.4	82397	62340	Python interpreter
rcs	5.7	18045	11909	Revision control system
remind	1.2.4	18222	13130	Schedule reminder
workman	1.4	13419	9653	Audio CD player
xfig	1.5.3	53244	42020	Drawing program
zephyr	5.7	42315	29218	Notification system
zsh	03.00.15	46223	35403	Command shell
Total		1212319	873086	

Figure 1: Analyzed packages and their sizes

1.1 The analyses

To build a better understanding of how the preprocessor is used, we wrote tools to analyze preprocessor usage and ran them on 27 C packages comprising 1.2 million lines of code. Figure 1 describes the packages and lists their sizes in terms of physical lines (or newline characters) and non-comment, non-blank (NCNB) lines, which disregards lines consisting of only comments or whitespace. The remainder of the analysis uses only the NCNB length, which more accurately reflects the amount of source code.

Overall, our analysis confirms that the C preprocessor is used in exceptionally broad and diverse ways, complicating the development of C programming support tools. On the other hand, the analysis also convinces us that, by extending our analysis framework with some class type inferencing techniques (similar to those used by Siff and Reps for C to C++ translation [SR96], O'Callahan

and Jackson for program understanding [OJ97], and others), we can take significant steps towards a tool that usefully converts a high percentage of Cpp code into C++ language features.[1] We are interested not in translations that merely allow a C program to be compiled by a C++ compiler (which is usually easy, by intentional design of C++) but those that take advantage of the added richness and benefits of C++ constructs.

In terms of the complexity of preprocessor usage, the results reported here contain both good news and bad. By far the largest number of macro definitions and uses are relatively simple, of the variety that a programmer could understand without undue effort (although perhaps requiring tedious work) or that a relatively unsophisticated tool could understand (although in practice very few even try). Despite the preponderance of innocuous macros, the preprocessor is so heavily used that the remaining ones are numerically significant. It is precisely these macros that are mostly likely to cause difficulties, and there are enough of them to be problematic in practice and to make the effort of understanding, annotating, or eliminating them worthwhile.

1.2 Coping with Cpp

Tools — and, to a lesser degree, software engineers — have three options for coping with Cpp. They may ignore preprocessor directives (including macro definitions) altogether, accept only post-processed code (usually by running Cpp on their input), or attempt to emulate the preprocessor.

Ignoring preprocessor directives is an option for approximate tools (such as those based on lexical or approximate parsing techniques), but accurate information about function extents, scope nesting, declared variables and functions, and other aspects of a program requires addressing the preprocessor.

Operating on post-processed code, the most common strategy, is simple to implement, but then the tool's input differs from what the programmer sees. Even when line number mappings are maintained, other information is lost in the mapping back to the original source code. For instance, source-level debuggers have no symbolic names or types for constants and functions introduced via `#define`, nor can tools trace or set breakpoints in function macros, as they can for ordinary functions (even those that have been inlined [Zel83]). As another example, Siff and Reps describe a technique that uses type inferencing to produce C++ function templates from C; however, the input is "a C program component that ... has been preprocessed so that all include files are incorporated and all macros expanded [SR96, p. 145]." Such preprocessing may limit the readability and reusability of the resulting C++ templates. As yet another related example, call graph extractors generally work in terms of the post-processed code, even when a human is the intended consumer of the call graph [MNL96]. Some tools even leave the software engineer responsible for inferring the mapping between the original and the post-processed source, which is an undesirable and error-prone situation.

A tool that first preprocesses code, or takes already-preprocessed code as input, cannot be run on a non-syntactic program or one that will not preprocess on the platform on which the tool is being run. These constraints complicate porting and maintenance, two of the situations in which program understanding and transformation tools are most likely to be needed. Additionally, a tool supplied with only one post-processed instantiation of the source code cannot reason about the

[1] Preliminary results indicate that many C++ packages rely heavily on Cpp, even when C++ supports a nearly identical language construct, probably due to a combination of trivial translations from C to C++ and of C programmers becoming C++ programmers without changing their habits.

program as a whole, only about that version that results from one particular set of preprocessor variables. For instance, a bug in one configuration may not be discovered despite exhaustive testing of other configurations that do not incorporate particular code or do not admit particular execution paths.

The third option, emulating the preprocessor, is fraught with difficulty. Macro definitions consist of complete tokens but need not be complete expressions or statements. Conditional compilation and alternative macro definitions lead to very different results from a single original program text. Preprocessing adds complexity to an implementation, which must trade off performing preprocessing against maintaining the code in close to its original form. Extracting structure from macro-obfuscated source is not a task for the faint-hearted. Despite these problems, in many situations only some sort of preprocessing or Cpp analysis can produce useful answers.

All three approaches would be unnecessary if programs did not use preprocessor directives. This is exactly what Stroustrup suggests:

> I'd like to see Cpp abolished. However, the only realistic and responsible way of doing that is first to make it redundant, then encourage people to use the better alternatives, and *then* — years later — banish Cpp into the program development environment with the other extra-linguistic tools where it belongs [Str94, p. 426].

C++ contains features — such as constant variables, inline functions, templates, and reference parameters — that obviate many uses of Cpp. Thus, translation to C++ is a path for partial elimination of Cpp. This study indicates the feasibility — and our framework for analyzing preprocessor usage provides a basis for the development — of an automatic translator with two attractive properties. It would take as input C programs complete with preprocessor directives, and it would map many — preferably most — uses of directives into C++ language features. (It is not practical to eliminate all uses of Cpp. For example, C++ currently provides no replacement for the `#include` directive, or for stringization or pasting. Macros that cannot be eliminated might be annotated with their types or effects on parser or program state, so that even tools that do no Cpp analysis can operate correctly on such programs.)

Another niche already filled by our tool is that of a "macro lint" program which warns of potentially dangerous (or non-standard) uses of Cpp.

1.3 Cpp: not all bad

Despite its evident shortcomings, Cpp is a useful and often necessary adjunct to C, for it provides capabilities unavailable in the language or its implementations. Cpp permits definition of portable language extensions that can define new syntax, abbreviate repetitive or complicated constructs, or eliminate reliance on a compiler implementation to open-code (inline) functions, propagate symbolic constants, eliminate dead code, and short-circuit constant tests. The latter guarantees are especially valuable for compilers that do a poor job optimizing or when the programmer wishes to override the compiler's heuristics. Cpp also permits system dependences to be made explicit and tested, resulting in a clearer separation of concerns. Finally, Cpp permits a single source to contain multiple different dialects of C; a frequent use is to support both K&R-style and ANSI-style declarations.

A limited number of tools do exist to assist software engineers to understand code with containing Cpp directives, such as debuggers that can call `#defined` functions and editors that support viewing one particular configuration of the code.

Our long-term goal is not to take these useful features away from programmers, but to reduce Cpp use, making programs easier for humans to understand and tools to analyze.

1.4 Outline

The remainder of this paper is organized as follows.

Section 2 reports the percentage of original C source code lines that are preprocessor directives, including a breakdown of the frequency of specific directives such as `#define`. C programs commonly have preprocessor directives as over 10% of their total lines, and over 20% of the lines were directives in 3 of the 27 packages.

Section 3 reports how often each macro is defined and expanded. In general identifiers are `#define`d relatively few times (96% of macro identifiers had three or fewer definitions). Many packages also have a significant number of macros that are never expanded, even disregarding system and library header files.

Section 4 categorizes macro definitions according to their expansions; for example, macros may simply define a preprocessor symbol, define a literal, expand to a statement, etc. We were particularly interested in determining the frequency of use of macros that are difficult to convert to other language features, such as those that string together characters as opposed to manipulating lexemes or syntactic units (less than one third of one percent of all macro definitions), those that expand to partial syntactic units such as unbalanced braces or partial declarations (half of one percent), and others not directly expressible in the programming language (about four percent).

Section 5 discusses the relevance of the research, suggests more techniques for mitigating the negative impact of Cpp on program understanding, and discusses avenues for future work, while section 6 discusses related work.

2 Occurrence of preprocessor directives

Figure 2 shows how often preprocessor directives appear in the programs we analyzed. Each group of bars in the figure represents the percentage of NCNB lines attributed to the specified category of directives, with each individual bar showing the percentage for a specific package. Conditional compilation directives are grouped together, as are "other" directives (such as `#error` and `#pragma`). These numbers do not include Cpp directives discovered in system header files, only in files included in the package.

Overall, more than 10% of NCNB program lines are preprocessor directives; the percentage varies by a factor of five across packages. Half of the packages have directives for over 9% of their lines, and one in nine exceed 21%, indicating quite heavy use of the preprocessor.

Conditional compilation directives account for just under half (46%) of the total directives in all packages, macro definitions comprise another 31%, file inclusion is 19%, macro undefinition makes up 2%, and the other directives are in the noise. The directive breakdown varies by quite a bit across packages: the percentage of `#define` varies from 14% to 51%, the percentage of `#include`s varies from 4% to 60%, and the percentage of conditional directives varies from 16% to 74%.

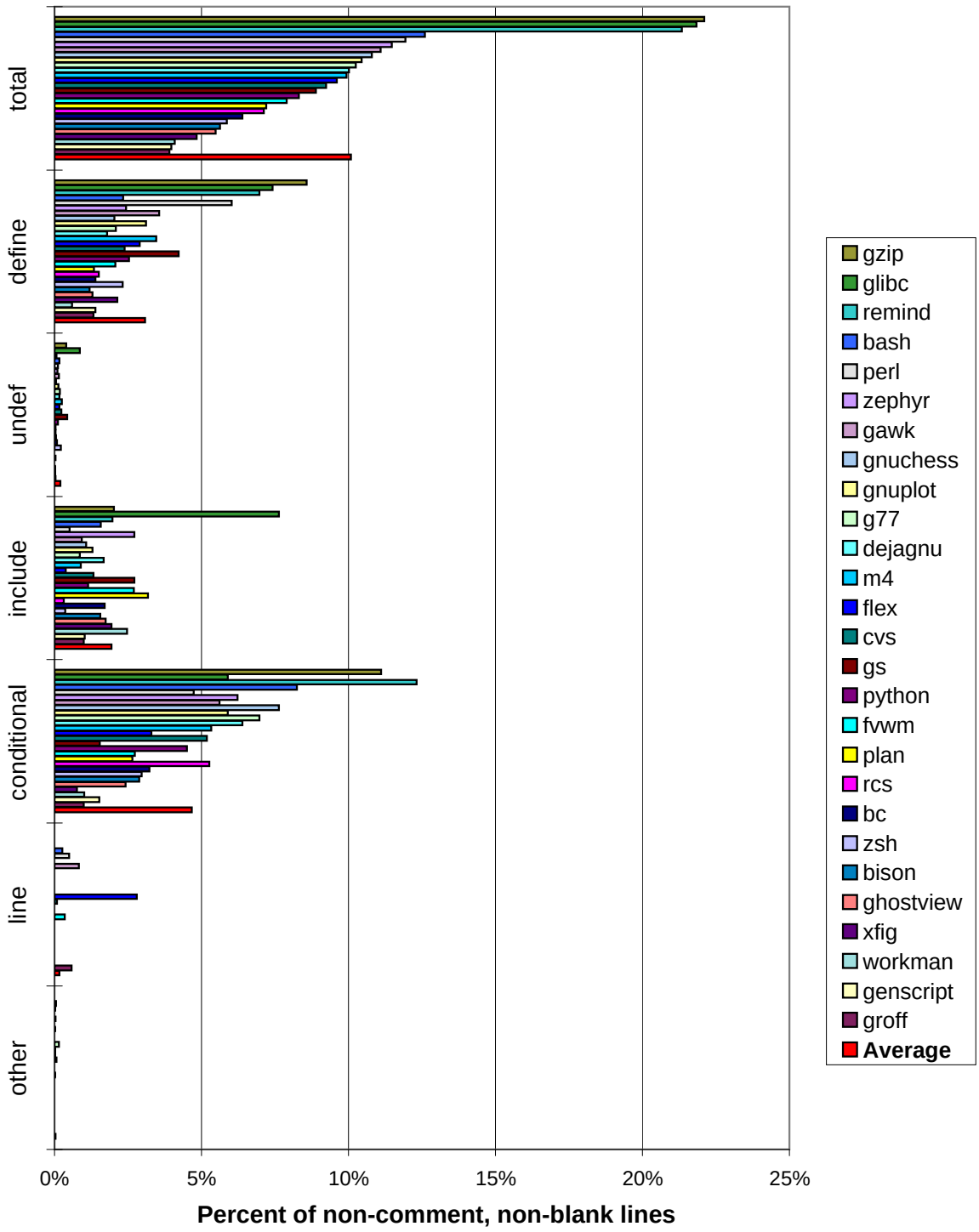

Figure 2: Preprocessor directives as a fraction of non-comment, non-blank (NCNB) lines.

2.1 `#line`, `#undef`, and "other" directives

The definedness of a macro is often used as a boolean value. However, our study shows that `#undef` is rarely used to set such macros to "false". Most uses of `#undef` immediately precede a definition of the just-undefined macro, to avoid preprocessor warnings about incompatible macro redefinitions. (About 90% of glibc's `#undef`s are used this way, and 216 of the 614 `#undef`s appear in a single file which consists of a long series of `#undef`s followed by a single `#include`.)

Every use of `#line` (in bash, cvs, flex, fvwm, gawk, groff, and perl) appears in lex or yacc output that enables packages build on systems lacking lex, yacc, or their equivalents. For instance, flex uses itself to parse its input, but also includes an already-processed version of its input specification (that is, C code corresponding to a `.l` file) for bootstrapping.

The only significant user of "other" directives is the g77 package, which contains 154 uses of `#error` (representing 1.5% of all preprocessor directives and 0.16% of all lines) to check for incompatible preprocessor flags.

2.2 Packages with heavy preprocessor use

Four packages — gzip, glibc, remind, and bash — deserve special attention for their heavy preprocessor usage. The first three have preprocessor directives as 21–23% of their lines.

gzip `#define`s disproportionately many macros as literals used as arguments to system calls, enumerated values, directory components, and more. These macros act like `const` variables and are evidence of good programming style. gzip also contains many conditional compilation directives, since low-level file operations (such as setting creation time and access control bits, accessing directories, and so forth) are done differently on different systems; gzip is a highly portable program.

glibc's heavy preprocessor use is largely accounted for by `#include` directives. Its files' average length is just 42 NCNB lines, and most contain several `#include` directives. Of the 1684 files, 182 are header files consisting of a single `#include` line, relieving glibc users of the need to know in which directory a header file actually really resides.

remind supports speakers of many different languages by using `#define`d constants for basically all user output. It also contains disproportionately many conditional compilation directives; over half of these test the definedness of `HAVE_PROTO`, in order to provide both K&R and ANSI prototypes.

Like gzip, bash is portable across a large variety of systems, but bash uses even more operating system services. Ninety-seven percent of bash's conditional compilation directives test the definedness of a macro whose presence or absence is a boolean flag indicating whether the current system supports a specific feature. The presence or absence of a feature requires different (or sometimes additional) system calls or other code.

3 Frequency of macro definition and usage

Figure 3 graphs the number of times each identifier is `define`d in each of the packages. No distinction is made between sequential redefinitions of a macro and multiple definitions that cannot take effect in a single configuration (say, because they appear in different branches of a Cpp conditional).

This graph shows a great deal of variation. For example, all macros defined by bc have only one or two different expansions, but more than 10% of macros defined by remind expand to more

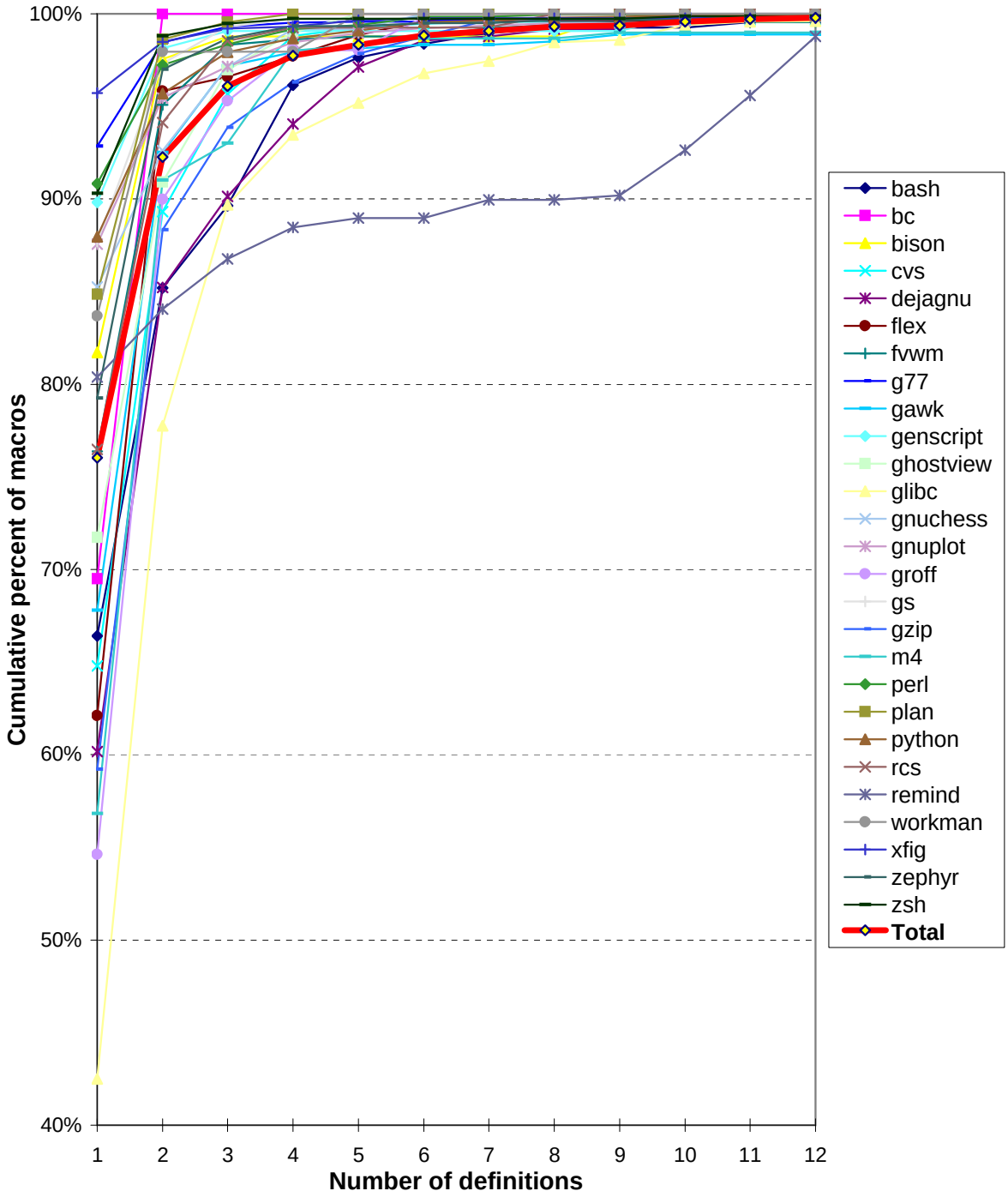

Figure 3: Number of definitions per Cpp identifier, graphed as the percentage of identifiers that are defined a given number of times or fewer. Overall, 96% of macros were defined three or fewer times; the other 4% of macros had four or more distinct definitions (`#define` directives). The outlier, for which 10% of macros are defined more than eight times, is the `remind` package, which uses macros for all user output.

than eight different texts.

In all but four packages, at least 93% of all macros are defined three or fewer times. For bash, glibc, and dejagnu, such macros account for 90%, largely because these packages are highly portable and also quite dependent on system libraries. The remind program uses macro definitions to provide localization support for ten different natural languages (and multiple character sets for some of them), accounting for its surprisingly large number of macros with many definitions. All of remind's macros are defined 14 or fewer times, but 16 macros in the 27 packages are defined more than 16 times, including three with more than 30 distinct definitions.

These data demonstrate that multiple definitions of symbols is not numerically frequent; even more importantly, the definitions of a symbol tend to be compatible, as shown in section 4.

Figure 4 is structured as the previous figure, but it represents the number of times that a defined name is expanded in either the package (not in system headers). About 82% of all macros were expanded eight or fewer times.

It is notable that most packages contain a significant number of defined macros that are never expanded — on average, over 13%. (Figure 4 reports only on macros defined in a package, not those defined in system or library header files, inclusion of which would push the unused percentage well above 50%.) Most packages are in the 4-12% range, while gnuplot exceeds 40%. Although it's difficult to fully account for the larger numbers, contributing factors include a lot of cut-and-paste and a lack of attention to implementations for specific platforms in some packages. Macros with 10 or fewer uses cover approximately 85% of the cases.

The tail of this distribution is quite long, indicating that some macros are used very heavily. Ninety-nine percent of macros are expanded 147 or fewer times, 99.5% of macros are expanded 273 or fewer times, 99.9% are expanded 882 or fewer times, and python uses `NULL` (which python itself defines) 4233 times. Figure 4 weights each macro equally rather than weighting each macro use equally, which would weight python's `NULL` 4233 times more heavily than a macro used only once (and infinitely more than a macro never used at all). Only macros defined in a package, and uses in that package, are counted; system macros and uses are excluded.

Figure 5 breaks down macro usage according to whether the macro invocation occurs in Cpp directives (which is further broken down into conditional tests and definition bodies), in other C code, in both, or in neither (i.e., no uses).

No package expanded all of its defined macros; two expanded fewer than 70% of the defined macros. The dominant usage was in C code only; these uses do not, therefore, have any affect on conditional compilation (for example).

In general, packages use macros either to direct conditional compilation or to produce code, but not for both purposes; this separation of concerns makes the source code easier to understand. Only 3.1% of macros expand in both code and conditional contexts (the fourth and fifth categories in the figure; the sixth, macro and code, accounts for only another 0.2% of macros). Conditional usage is rare in general; conditional compilation accounts for half of Cpp directives but only 5.4% of macros (plus the categories just listed above).

4 Categorization

This section examines the purposes of macros and how they are intended to be used, which requires heuristic categorization of macro definition bodies. A straightforward refinement that we are pursuing examines macro uses to aid this categorization. (For example, a macro used where a

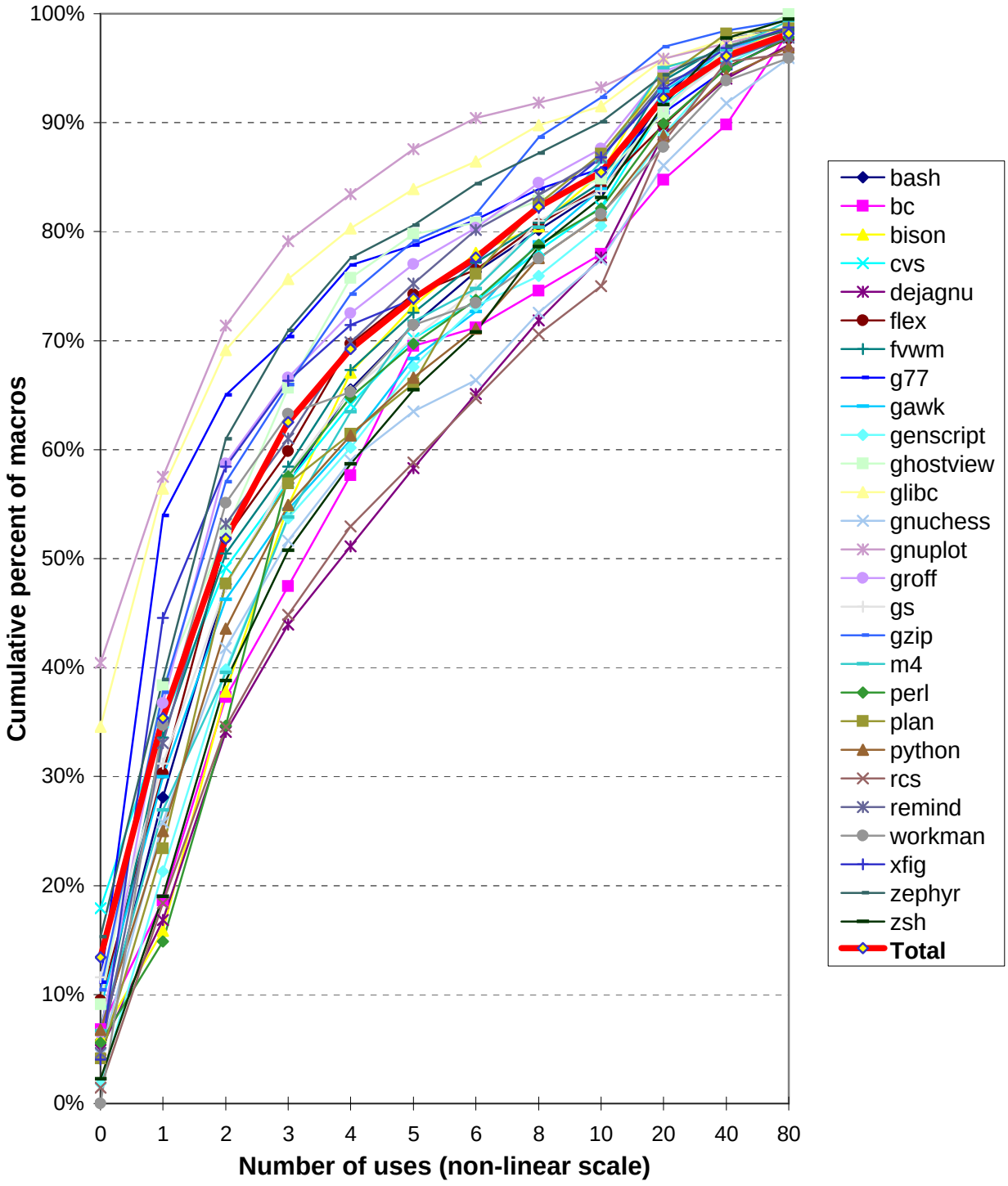

Figure 4: Number of expansions per Cpp macro. The numbers in the table represent the percentage of identifiers which are expanded a given number of times or fewer. For example, g77 expands 65% of its macros two or fewer times.

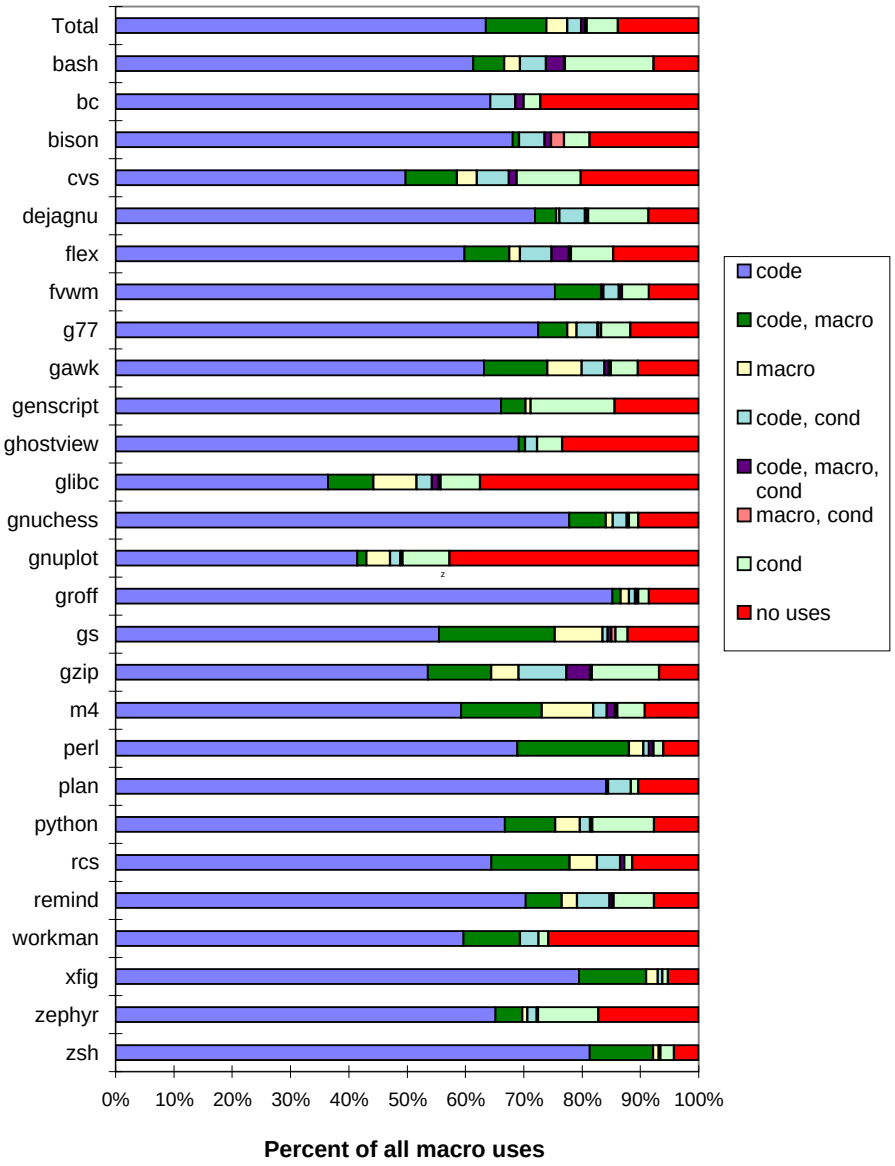

Figure 5: Where macros are used: in C code, in macro definition bodies, in conditional tests, or in some combination thereof. The figure reflects only package macros and uses, not system files.

type should appear can be inferred to expand to a type; a macro used before a function body is probably expanding to a declarator.)

In addition to classifying each macro as taking arguments or not, our tool identifies the following specific categories (and a number of more rarely-used ones omitted for brevity; figure 7 contains a more complete list). The examples are chosen for clarity and brevity from the packages studied.

Null define The `#define` gives only an identifier name but no macro body, as in `#define HAVE_PROTO`. Such macros appear most frequently in Cpp directives (such as `#ifdef`), where they are used as boolean variables by the preprocessor, but may also appear in code. For

instance, macro `private` may expand either to `static` or to nothing, depending on whether a debugging mode is set. The definition which causes it to expand to nothing is categorized as a null define (the other is categorized as "other syntactic macro"; see below).

Constant The macro is defined to be either a literal or an operator applied to constant values. For instance, `#define NULL 0`, `#define ARG_MAX 131072`, and `#define ETCHOSTS "/etc/hosts"` define literals, while `#define RE_DUP_MAX ((1<<15)-1)` and `#define RED_COLS (1 << RED_BITS)` (where `RED_BITS` is a constant, possibly a literal) define constants. Such macros act like `const` variables.

Expression The macro body is an expression, as in `#define sigmask(x) (1 << ((x)-1))`. This expression might have a single constant value everywhere (the usual case for expression macros without arguments, most of which are classified as constants, above) or might have a different value on each use (the usual case for expression macros with arguments).

One tenth of expression macros in our study use assignment operators, which have potentially unexpected results. A macro argument that is assigned to is similar to a pass-by-reference function argument and need only be noted in the macro's documentation. A macro that assigns a global variable also presents no difficulties in understanding or translation into a C++ inline function. Assignment to a local variable that is free in the macro body, however, demands that such a variable exist wherever the macro is invoked, and assigns to different variables at different invocations.[2] Such a macro implements a restricted form of dynamic scoping by capturing the instance of a variable visible at the point of macro invocation.

Statement The macro body is a complete statement such as "`x = 3;`", "`if (s) free(s);`" or "`{ int x = y*y; printf("%d", x); }`". Such a macro is like a function returning `void`, except that uses should not be followed by a semicolon.[3]

Stringization and pasting The macro body contains `#` or `##`, which treat the macro argument not as a token but as a string. Examples include `#define spam1(OP,DOC) {#OP, OP, 1, DOC},`, `#define REG(xx) register long int xx asm (#xx)`, and `#define __CONCAT(x,y) x ## y`. No C or C++ language mechanism can replace such macros.

Other syntactic macros Like stringization and pasting, these macros make essential use of the unique features of the preprocessor. Our framework separately categorizes a number of such macros, including those that expand to a reserved word (such as `#define private static`, mentioned above), those that expand to a delimiter (such as `#define AND ;`), and those with mismatched parentheses, brackets, or braces. The latter are often used to create a block and perform actions that must occur at its beginning and end, as for `BEGIN_GC_PROTECT` and `END_GC_PROTECT`.

[2] By contrast, LCLint considers assignment to a macro argument dangerous but does not appear to check for assignments to local variables. [Eva96a]

[3] Since the body is already a complete statement, the extra semicolon can cause problems such as mis-parsing of nested `if` statements. Such macros can be confusing to use, because programmers are inclined to add a semicolon after invocations that look like functions; wrapping the body in `do {...} while (0)`, a partial statement which requires a trailing semicolon, solves this problem. To our surprise, we found few uses of that construct, but many error-prone instances of a macro that expanded to a statement like {...} in which a call to the macro was immediately followed by a semicolon.

Type-related macros These macros either take a type as an argument, pass a type to another macro, expand to a type or partial type, or use such a macro. Examples include `#define __ptr_t void *`, `#define __INLINE extern inline`, `#define ALIGN_SIZE sizeof(double)`, and `#define PTRBITS __BITS(char*)`. Since types are not first-class in C, they may not be passed to functions or returned as results; additionally, these macros may produce or use only part of a type (such as a storage class). As a result, these macros may be tricky to understand, and cannot be eliminated via straightforward translation (though C++ templates may provide some hope).

Recursive The ISO C standard permits macros to be recursively defined (the preprocessor performs only one level of expansion), as in `#define LBIT vcat(LBIT)`. This mechanism permits already-defined or to-be-defined macros to be extended or modified.

Classification failure Multiple adjacent identifiers — as in `#define EXFUN(name, proto) name proto` and `#define DO_OP(OP,a,b) (a OP b)` — caused many failures of our classification heuristics. Of the 1025 classification failures in the 27 packages, 496 were caused by a single definition in gnuplot, `#define CUR cur_term->type.` (period is part of expansion), and uses of that macro, as in `#define acs_plus CUR Strings[408]`. Four packages — bison, gnuchess, remind, and workman — had no macro classification failures. These packages contain 93, 297, 932, and 58 macro definitions, respectively.

Figure 6 shows the percentage of macros that fit into these categories for each package. Overall, 83% of macros are expressions — mostly constants; further analysis of the conditional compilation structure (in the style of Krone and Snelting [KS94]) and of the macros with free variables (essentially achieving dynamic scoping) is needed to see which of the roughly 33% of expression macros should be easy to convert to C++ language features such as constants or enumerated values. The 7% that are null defines, should also be easy to understand and/or translate. Another 5% are statements, most of which are straightforward (complications include scoping and semicolon swallowing). That only 0.2% of macros exploit stringization or pasting, the only truly extra-linguistic capabilities in the C preprocessor, is encouraging.

Our tool failed to categorize less than 2% of the 26701 definitions; performing even a single level of macro expansion in bodies would make most of those failures categorizable. Other straightforward improvements include making a second pass after an initial categorization and using dependence information to determine which definitions can be active at an invocation site. We have not pursued these enhancements, primarily because our tool is already accurate enough for our purposes.

Figure 7 indicates that multiple definitions of a particular macro tend to be compatible. It classifies macros rather than macro definitions, and uses a finer breakdown of categories. Each macro is given a set of categorizations (corresponding to the categorizations of its definitions) and the incidence of each such is displayed. For over 97% of macros, all of the macro's definitions are given the same categorization — even when categories such as literal, constant, expression, and expression with assignment are considered unrelated. The most common "conflict", statement and null define, is also harmless in most contexts. We expect that closer examination of most of the other conflicts will demonstrate that they present no real obstacles to understanding (even if they do complicate some details).

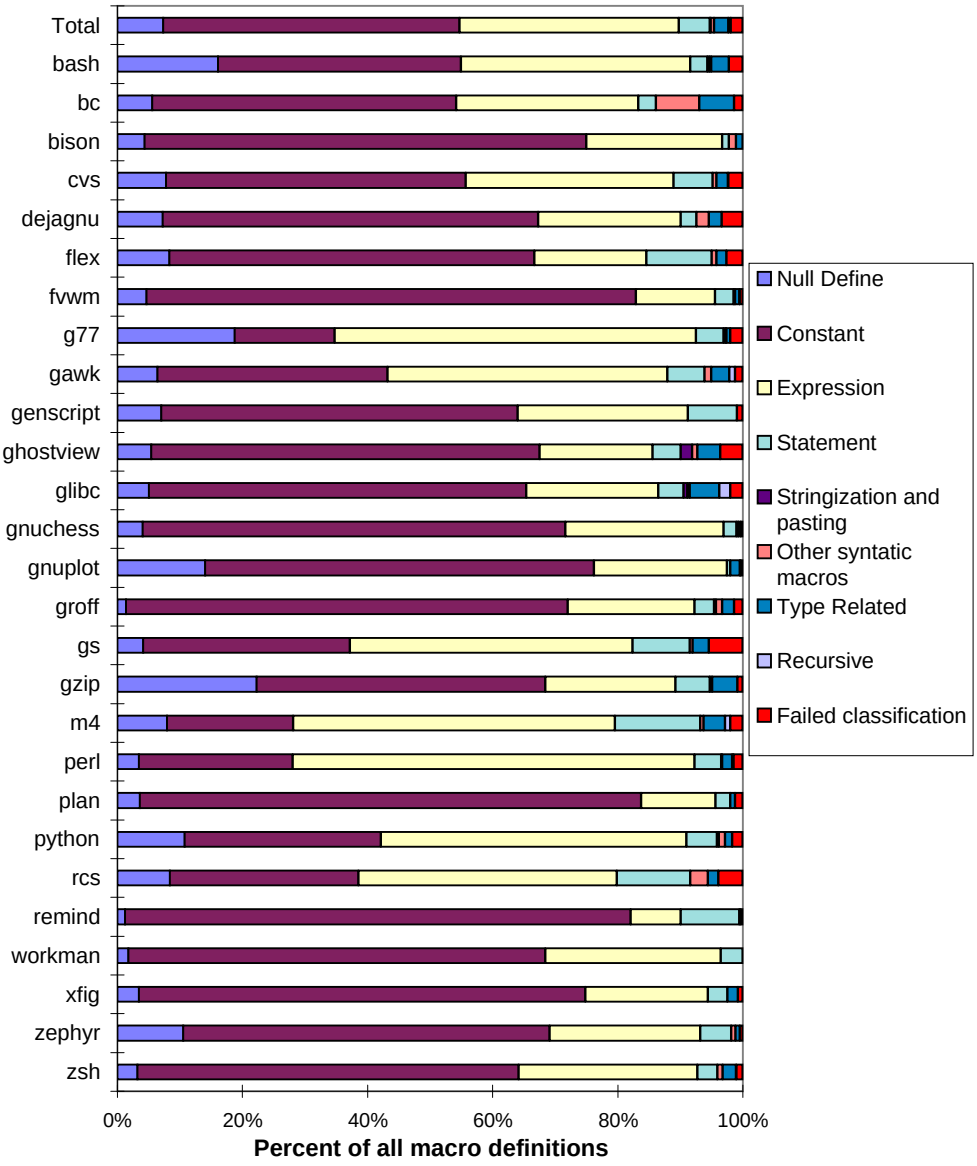

Figure 6: Categorization of macro definition bodies.

5 Conclusions

5.1 Relevance of the results

The results of this research are of interest to language designers, tool writers, programmers, and software engineers.

Language designers can examine uses of the macro system's extra-linguistic capabilities to determine what programmers consider missing from the language. Future language specifications can support (or prevent!) such practices in a more disciplined, structured way.

Programming tool writers, too, need to understand how Cpp is used, for that sheds insight on

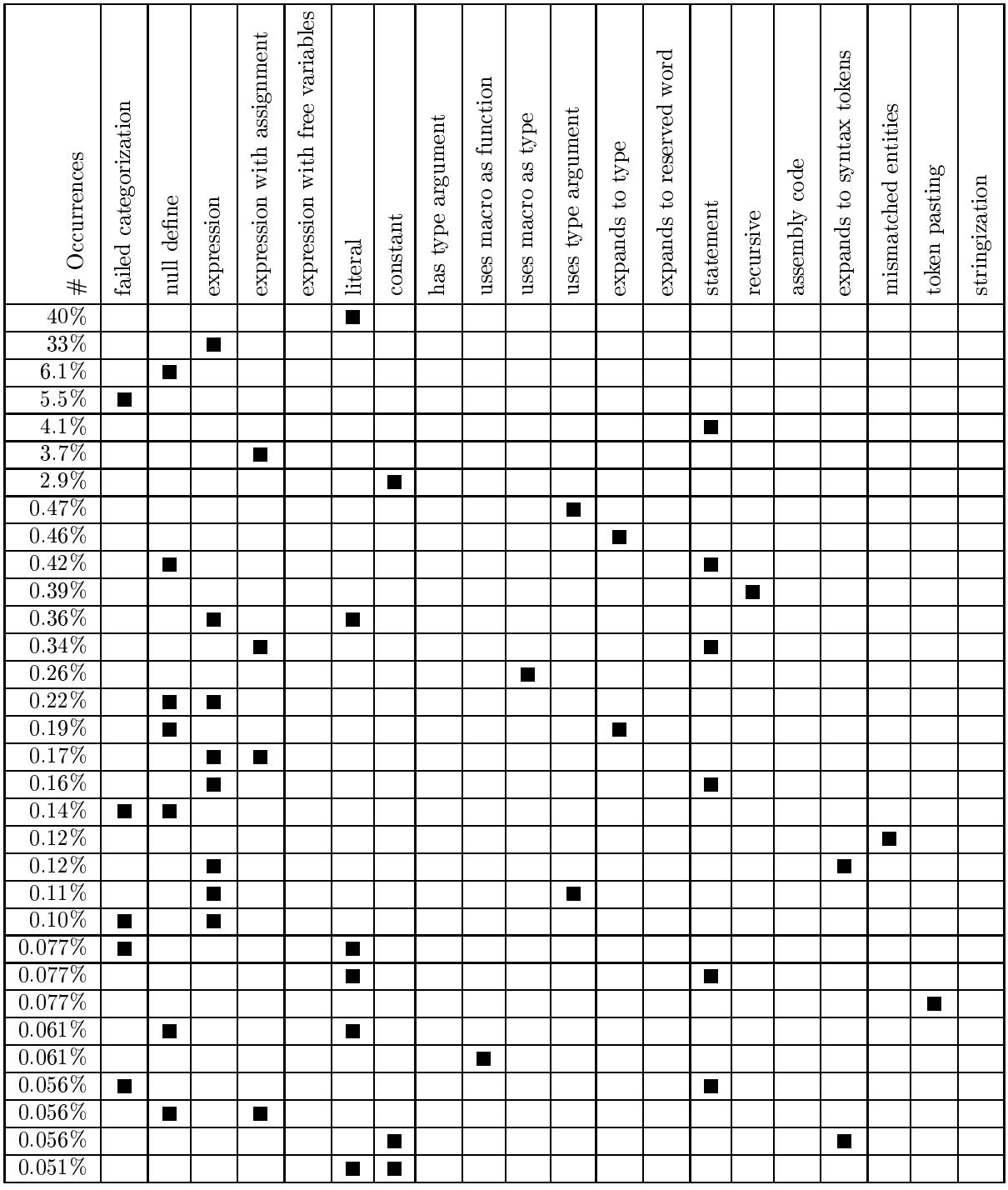

Figure 7: Subset categorization of macros (not macro definitions). Items less than one twentieth of a percent are omitted; such items appear fewer than ten times in the codebase.

the sorts of inputs that will be provided to the tool. By coping with the most common constructs, the tool can provide relatively good coverage for low effort. By identifying problematic uses, much better feedback can be given to the programmer, who can be more effective as a result. The analysis results also indicate the difficulty of processing preprocessor directives; before these analyses, we did not know whether the task was so trivial as to be uninteresting, so difficult as to be not worth attempting, or somewhere in between.

The analyses are of interest to programmers who wish to make their code cleaner and more portable, and can help them to avoid constructs that cause tools (such as test frameworks and program understanding tools) to give incomplete or incorrect results.

Finally, our results are of interest to software engineers for all of the above reasons and more. Since this is the first Cpp usage study of which we are aware, it is worth performing simply to determine whether the results were predictable a priori; we did in fact discover a number of interesting features of our suite of programs.

5.2 Making C programs easier to understand

The combination of C and Cpp makes a source text unnecessarily difficult to understand. A good first step is to eliminate Cpp uses where an equivalent C or C++ construct exists, and to apply tools to explicate the remaining uses. Here we discuss a few approaches to solving this problem by eliminating the source of confusion rather than applying tools. We do not seriously consider simply eliminating the preprocessor, for it provides conveniences and functionality not present in the language.

Since many of the most problematic uses of Cpp provide portability across different language dialects or different operating environments, standardization can obviate many such uses. Canonicalizing library function names and calling conventions makes conditional compilation less necessary and incidentally makes all programs more portable, even those which have not gone to special effort to achieve portability. This proposal moves the responsibility for portability (really, conformance to a specification) from the application program into the library or operating system, which is a reasonable design choice since many application programs rely on a much smaller number of libraries and run on relatively few operating systems.

Likewise, the most common single cause for Cpp directives would be eliminated if the C language and its dialects had only a single declaration syntax. Because most C compilers, and all C++ compilers, accept ANSI-style declarations, much support for multiple declaration style may have outlived its usefulness. We are investigating the effect on our statistics (and program understandability) of "partially evaluating" a program source by specifying the definedness and values of some Cpp identifiers.

Some Cpp directives, such as `#include`, can be moved into the language proper; this would also eliminate the need for Cpp constructs that prevent multiple inclusion of header files. Likewise, compilers that do a good job of constant-folding and dead code elimination can encourage programmers to use language constructs rather than relying on the guarantees of an extra-linguistic tool like Cpp.[4]

[4]Interestingly, the issue seems to not be whether compilers do the appropriate optimizations, but whether programmers have confidence that the optimizations will be performed; if unsure, programmers will continue to resort to Cpp, since certainly a compiler cannot generate code for source that it does not ever even see (because Cpp has already stripped it away).

Cpp constructs meant for specific purposes could be replaced by a special-purpose syntax for those frequently-occurring cases. For instance, declarations or partial declarations could be made explicit (perhaps first-class) objects. Manipulations of these objects would then be performed through a clearly-specified interface rather than via string concatenation, easing the understanding burden on the programmer. Such uses would also be visible to the compiler and could be checked and reasonable error messages provided. The downside of this approach is the introduction of a new syntax or new library functions which may not simplify the program text and which cannot cover all cases, only a few specified ones.

An alternative approach which avoids the clumsiness of a separate language of limited expressibility is to make the macro language more powerful — perhaps even using the language itself via constructs evaluated at compile time rather than run time. (The macro systems of Common Lisp and Scheme, and their descendants [WC93] take this approach.) An extreme example would be to provide a full-fledged reflection capability. Such an approach is highly general, powerful, and and theoretically clean; it circumvents many of the limitations of Cpp. However, this approach does *not* lead to more understandable programs and may result in just the opposite. As difficult as it may be to determine what output a macroless program produces, it can be just as difficult simply to determine the text of a program which uses such macros. (In practice, such systems are used in fairly restricted ways.) A dialog among users, compiler writers, tool writers, and language theorists is necessary when introducing a feature in order to prevent unforeseen consequences from turning it into a burden.

6 Related work

We could find no other empirical study of the use of the C preprocessor nor any other macro processor. However, we did find guidance on using C macros effectively and tools for checking macro usage.

Carroll and Ellis state that "almost all uses of macros can be eliminated from C++ libraries" [CE95, p. 146]. They list eight categories of macro usage and explain how to convert them into C++ mechanisms. They do not discuss automatic conversion, but focus on instructing the software engineer on better ways to do Cpp-like things.

Similarly, a number of organizations provide hints about effective ways to the use the C preprocessor. The GNU documentation, for example, discusses a set of techniques including simple macros, argument macros, predefined macros, stringization macros, concatenation macros, undefining and redefining macros. It also identifies a set of "pitfalls and subtleties of macros"; these are much like some of the problems our analysis tool identifies. We discovered that these categorizations sometimes focussed on constructs that don't happen very often or missed ones that are actually frequent. Our effort not only categorizes problems, but it also determines the frequency of appearance of those problems and discovers other idiosyncratic uses.

A number of tools check whether specific C programs satisfy particular constraints. The lint program checker, distributed with most Unix systems, checks for potentially problematic uses of C. The implementation of lint is complicated by the fact that it tries to replicate significant functions of both the C compiler and the preprocessor.

LCLint performs many of lint's checks and also allows the programmer to add annotations which enable additional checks [Eva96b, EGHT94]. LCLint optionally checks function-like macros — that is, those which take arguments — for macro arguments on the left hand side of assignments, for

statements playing the role of expressions, and for consistent return types. LCLint's approach is prescriptive: programmers are encouraged not to use constructs that might be dangerous, or to change code that contains such constructs. We are more interested in analyzing, describing, and automatically removing such uses so that tools can better process existing code without requiring human interaction or producing misleading results.

Krone and Snelting use mathematical concept analysis to determine the conditional compilation structure of code [KS94]. They determine, for each line, which preprocessor macros it depends upon, and display that information in a lattice. They do not determine how macros depend upon one another directly, only by their nesting in `#if`, and the information conveyed is about the program as a whole.

References

[CE95] Martin D. Carroll and Margaret A. Ellis. *Designing and Coding Reusable C++*. Addison-Wesley, Reading, Massachusetts, 1995.

[EGHT94] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of SIGSOFT '94 Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.

[Eva96a] David Evans. *LCLint User's Guide*, version 2.2 edition, August 1996. http://larch-www.lcs.mit.edu:8001/larch/lclint/guide/guide.html.

[Eva96b] David Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN '96: Programming Language Design and Implementation*, pages 44–53, May 1996.

[HS95] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, fourth edition, 1995.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering*, pages 49–57. IEEE Computer Society Press, May 1994.

[MNL96] Gail C. Murphy, David Notkin, and E. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*, pages 90–99, March 1996.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, May 1997.

[SR96] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Proceedings of SIGSOFT '96 Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 135–146, October 1996.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, Albuquerque, NM, June 1993.

[Zel83] Polle T. Zellweger. An interactive high-level debugger for control-flow optimized programs. Technical Report CSL-83-1, Xerox Palo Alto Research Center, Palo Alto, California, January 1983.