Analyzing Source Code Across Static Conditionals

by

Paul Gazzillo

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

January 2016

Thomas Wies

Dedication

Dedicated to my mother.

Acknowledgments

I would like to thank Thomas Wies for taking me under his wing when I needed a new advisor even though he was incredibly busy with his own research and students. His advice on this thesis, the defense, paper-writing, and computer science were invaluable. He happily gave time and earnest consideration to my work, and I could not have finished this thesis without his help. I also would like to thank my first advisor Robert Grimm for providing a intense entree into academia. He inculcated in me good practices in research, writing, programming. He showed me how far I could push myself, that I am capable of meeting challenges I never before thought possible. I would like to thank Ben Goldberg. He is the best teacher I have ever had. The first graduate course I took was his Honors Programming Languages class. I had no academic computer science experience and was not the best student as an undergrad. Ben took time to sit down with me before the course to talk and took a chance letting a first-year Masters student take this PhD-level course. For the first-time in my life, I actually *enjoyed* going to class. And finally, I would like to thank Eric Koskinen for taking a chance on me.

Abstract

We need better tools for C, such as source browsers, bug finders, and automated refactorings. The problem is that large C systems such as Linux are software product lines, containing thousands of configuration variables controlling every aspect of the software from architecture features to file systems and drivers. The challenge of such configurability is how do software tools accurately analyze all configurations of the source without the exponential explosion of trying them all separately. To this end, we focus on two key subproblems, parsing and the build system. The contributions of this thesis are the following: (1) a configuration-preserving preprocessor and parser called SuperC that preserves configurations in its output syntax tree; (2) a configuration-preserving Makefile evaluator called Kmax that collects Linux's compilation units and their configurations; and (3) a framework for configuration-aware analyses of source code using these tools.

C tools need to process two languages: C itself and the preprocessor. The latter improves expressivity through file includes, macros, and static conditionals. But it operates only on tokens, making it hard to even parse both languages. SuperC is a complete, performant solution to parsing all of C. First, a configuration-preserving preprocessor resolves includes and macros yet leaves static conditionals intact, thus preserving a program's variability. To ensure completeness, we analyze all interactions between preprocessor features and identify techniques for correctly handling them. Second, a configuration-preserving parser generates a well-formed AST with static choice nodes for conditionals. It forks new subparsers when encountering static conditionals and merges them again after the conditionals. To ensure performance, we present a simple algorithm for table-driven Fork-Merge LR parsing and four novel optimizations. We demonstrate SuperC's effectiveness on the x86 Linux kernel.

Large-scale C codebases like Linux are software product families, with complex build systems that

tailor the software with myriad features. Such variability management is a challenge for tools, because they need awareness of variability to process all software product lines within the family. With over 14,000 features, processing all of Linux's product lines is infeasible by brute force, and current solutions employ incomplete heuristics. But having the complete set of compilation units with precise variability information is key to static tools such a bug-finders, which could miss critical bugs, and refactoring tools, since behavior-preservation requires a complete view of the software project. Kmax is a new tool for the Linux build system that extracts all compilation units with precise variability information. It processes build system files with a variability-aware `make` evaluator that stores variables in a conditional symbol table and hoists conditionals around complete statements, while tracking variability information as presence conditions. Kmax is evaluated empirically for correctness and completeness on the Linux kernel. Kmax is compared to previous work for correctness and running time, demonstrating that a complete solution's added complexity incurs only minor latency compared to the incomplete heuristic solutions.

SuperC's configuration-preserving parsing of compilation units and Kmax's project-wide capabilities are in a unique position to process source code across all configurations. Bug-finding is one area where such capability is useful. Bugs may appear in untested combinations of configurations and testing each configuration one-at-a-time is infeasible. For example, one compilation unit that defines a global function called by other compilation units may not be linked into the final program due to configuration variable selection. Such a bug can be found with Kmax and SuperC's cross-configuration capability. Cilantro is a framework for creating variability-aware bug-checkers. Kmax is used to determine the complete set of compilation units and the combinations of features that activate them, while SuperC's parsing framework is extended with semantic actions in order to implement the checkers. A checker for detecting linker errors across all compilation units in the Linux kernel demonstrates each part of the Cilantro framework and is evaluated on the Linux source code.

Table of Contents

List of Tables

List of Figures

Introduction

As software systems become larger, automated software engineering tools such as source code browsers, bug finders, and automated refactorings, become more important. Larger systems are more vulnerable to bugs, and modifications to the codebase are more difficult to verify by hand due to the larger number of interactions between features of the system. C is the language of choice for many common large-scale software systems, including the Linux kernel, the Apache web server, and the GNU compiler collection, all of which are used in critical computing systems. One facet of large-scale software development is variability management, with which software systems are tailored to a specific use by enabling features at build-time. For example, the Linux kernel can be configured and compiled for embedded devices, PCs, and server farms alike from the same codebase. With variability, a codebase encompasses a software product line (SPL) of customized software products that share portions of the source code. This variability amplifies the difficulty of creating and using automated software tools, because such tools need to work on all product variations in the software family as a whole. Worse still, variability introduces new classes of bugs resulting from the interactions between variations. Abal et al. found such bugs in the Linux kernel, but lacking automated tools, searched by hand for previously patched bugs sent to the Linux kernel mailing list [4].

New techniques for describing and implementing variability promise safety and easier development of software tools. For instance, McCloskey and Brewer describe ASTEC, a new C preprocessor language that avoids the difficulties caused by C's unstructured preprocessor [48]. But translating existing C code to ASTEC runs into the same challenges that all variability-aware software tools encounter. Aspect-oriented programming better organizes variability by restricting changes to specific cutpoints in the program source, but aspects only handle limited preprocessor usage [5]. Formal module systems

define variability by decomposing features into modules [22], but are not realistic for variability in C projects, due to the exponential explosion of module combinations [43]. While such techniques provide hope for the future of software engineering, they are not widespread. An abundance of critical C software remains that uses ad-hoc techniques for variability. And it is not clear whether these new techniques are sufficiently powerful to express the variability constructs that C developers need and use. The preprocessor is not a programming language in the traditionally sense; it is a text processing tool separate from the C language, oblivious to C's syntax. The preprocessor is free to replace macros with virtually any other text. This freedom is used well by developers to implement modularity, portable code, and even to add higher-level language features like poor-man's generics and iterators to C programs. Such uses occur frequently in real-world code, as work by Ernst et al. shows [28]. The preprocessor's freedom also allows nonsensical uses. In fact, The International Obfuscated C Code Contest[1] is a yearly contest that invites submissions of the most obfuscated, yet fully-working, C code, much of which uses the preprocessor for purposely unreadable code. This freedom gives software analysis tools the difficult job of balancing their power with support for legitimate uses of the preprocessor. And because of the preprocessor's many uses, replacements for it need to do more than variability management; they need to replace the powerful extensions to the C language enabled for decades by the preprocessor.

This thesis provides the foundation for variability-aware software tools. It focuses on two core components, parsing and the build system, that are integral to all software engineering tools, and shows how they work together to perform project-wide analysis on C software that scales to large codebases like Linux. In creating these techniques, we expose of the challenges that need to be overcome and abstract them away, enabling future implementation of variability-aware software tools. Three typical software engineering tasks illustrate the primacy of these two components. Code browsers help developers wade through massive code bases, bug finders and static analyzers improve code quality and speed up testing, and automated refactorings reduce human error when restructuring and improving code. Common to all three tools is the need to first parse the source code. But C programs' ad-hoc implementation of variability with the preprocessor impedes parsing all configurations simultaneously. Additionally, all three tools need to see the entire codebase, but C supports separate compilation. Projects are broken into thousands of individual source files, and the build system implements ad-hoc variability with languages such as `make`. Simple for software tools without variability, these two tasks, parsing and building, are made more

[1] `http://www.ioccc.org/`

difficult when supporting variability. The contributions of this thesis are as follows:

1. a complete analysis of the challenges to parsing C with the preprocessor, new algorithms for configuration-preserving preprocessor and parsing, and a empirical evaluation of their implementation in a new tool, SuperC;

2. a new algorithm to extract compilation units and their presence conditions from Linux Makefiles and its embodiment in the Kmax tool, which is evaluated for correctness and compared to previous work; and

3. a demonstration of SuperC and Kmax working together to scale simple bug finders across the entire Linux kernel by extending SuperC with cross-configuration semantic actions and modeling varibility bugs as SAT problems.

Parsing is difficult because C source code mixes two languages: the C language proper and the preprocessor. The preprocessor adds facilities lacking from C itself. File includes ("#include") provide rudimentary modularity, macros ("#define") enable code transformation with a function-like syntax, and static conditionals ("#if", "#ifdef", and so on) capture variability. Worse, the preprocessor is oblivious to C constructs and operates only on individual tokens. Real-world C code reflects both points: preprocessor usage is widespread and often violates C syntax [28]. Chapter 1 describes SuperC, a configuration-preserving preprocessor and parser. Its preprocessor resolves includes and macros yet leaves static conditionals intact, thus preserving a program's variability. To ensure completeness, we analyze all interactions between preprocessor features and identify techniques for correctly handling them. SuperC's parser generates a well-formed AST with static choice nodes that preserve mutually-exclusive variations of the source code. It forks new subparsers when encountering static conditionals and merges them again after the conditionals. To ensure performance, we present a simple algorithm for table-driven Fork-Merge LR parsing and four novel optimizations.

While SuperC provides the foundation for variability-aware tools to handle individual source files, C programs are comprised of potentially thousands of compilation units, the separately compiled C files linked to form the final program. The Linux kernel v3.19, for instance, contains over 20,000 compilation units, while only a subset of these compilation units are ever used for a single variation of the kernel. The build system implements variability by conditionally compiling only those C files needed for the user's

selected features. Extracting all compilation units and their variability information is crucial for software engineering tools. For instance, C function calls can cross compilation unit boundaries, referenced only by an `extern` declaration. Without knowing the complete set of compilation units that can be linked, static analyses may not find all callees. Software product lines harbor untestable bugs, since it is not feasible to check every possible combination of features separately. Additionally, translating Linux's extensive C preprocessor use to a safer alternative, such as aspects [5] or to a new preprocessor language [48], depends on a complete view of the kernel source, as do code browsers, bug-finders, and automated refactorings. Chapter 2 describes collecting compilation unit information from Linux's `make`-based build system with the Kmax tool. Kmax first collects a feature model from Linux's Kconfig specification files. It then extracts all compilation units and their precise variability information from the Kbuild Makefiles. This is possible using a variability-aware `make` evaluator that stores variables in a conditional symbol table and hoists conditionals around complete statements, while tracking variability information as boolean expressions of features.

Combining Kmax's ability to deduce the project-wide collection of C files with SuperC's configuration-preserving preprocessor and parser enable bug checkers with added support for semantic actions in SuperC's parser. Semantic actions enable checkers for many compile-time bugs, such as undefined symbols, unused variables, mismatched types, and linker errors. Variability complicates this analysis, because two configurations may declare the same symbol with different types or omit different compilation units. SuperC already does the difficult job of tracking configurations, making semantic actions more convenient than processing the resulting AST. Two key techniques support variability-aware semantic analyses: a parsing context that forks and merges along with subparsers and a conditional symbol table that maps identifiers to all configurations' possible values. Chapter 3 details a framework for combining Kmax and SuperC together, describes checkers implemented with SuperC, and evaluates the framework by checking the entire Linux kernel for linker errors across all configurations.

This thesis provides the foundation for efficient software engineering tools that work across all variations of C systems. It solves the problem of parsing all of C with preprocessor usage and collecting all compilation units with their presence conditions from the build system. Simple bug checkers that scale across a large C system show that these components work together efficiently to enable variability-aware software tools.

Chapter 1

Parsing

1.1 Introduction

Large-scale software development requires effective tool support, such as source code browsers, bug finders, and automated refactorings. This need is especially pressing for C, since it is the language of choice for critical software infrastructure, including the Linux kernel and Apache web server. However, building tools for C presents a special challenge. C is not only low-level and unsafe, but source code mixes *two* languages: the C language proper and the preprocessor. First, the preprocessor adds facilities lacking from C itself. Notably, file includes ("#include") provide rudimentary modularity, macros ("#define") enable code transformation with a function-like syntax, and static conditionals ("#if", "#ifdef", and so on) capture variability. Second, the preprocessor is oblivious to C constructs and operates only on individual tokens. Real-world C code reflects both points: preprocessor usage is widespread and often violates C syntax [28].

Existing C tools punt on the full complexity of processing both languages. They either process one configuration at a time (e.g., the Cxref source browser [15], the Astrée static analyzer [16], and Xcode refactorings [17]), rely on a single, maximal configuration (e.g., the Coverity bug finder [13]), or build on incomplete heuristics (e.g., the LXR source browser [36] and Eclipse refactorings [37]). Processing one configuration at a time is infeasible for large programs such as Linux, which has over 10,000 configuration variables [61]. Maximal configurations cover only part of the source code, mainly due to

static conditionals with more than one branch. For example, Linux' "allyesconfig" enables less than 80% of the code blocks contained in conditionals [60]. And heuristic algorithms prevent programmers from utilizing the full expressivity of C and its preprocessor. Most research focused on parsing the two languages does not fare better, again processing only some configurations at a time or relying on incomplete algorithms [5, 7, 9, 10, 29, 34, 48, 51, 59, 66].

Only MAPR [53] and TypeChef [41, 42] come close to solving the problem by using a two-stage approach. First, a configuration-preserving *preprocessor* resolves file includes and macros yet leaves static conditionals intact. Second, a configuration-preserving *parser* forks its state into subparsers when encountering static conditionals and then merges them again after conditionals. The parser also normalizes the conditionals so that they bracket only complete C constructs and produces a well-formed AST with embedded static choice nodes. Critically, both stages preserve a C program's full variability and thus facilitate analysis and transformation of all source code. But MAPR and TypeChef still fall short. First, the MAPR preprocessor is not documented at all, making it impossible to repeat that result, and the TypeChef preprocessor misses several interactions between preprocessor features. Second, both systems' parsers are limited. TypeChef's LL parser combinator library automates forking but has seven combinators to merge subparsers again. This means that developers not only need to reengineer their grammars with TypeChef's combinators but also have to correctly employ the various join combinators. In contrast, MAPR's table-driven LR parser engine automates both forking and merging. But its naive forking strategy results in subparsers exponential to the number of conditional branches when a constant number of subparsers suffices.

This paper significantly improves on both systems and presents a rigorous treatment of both configuration-preserving preprocessing and parsing. In exploring configuration-preserving preprocessing, we focus on completeness. We present a careful analysis of *all* interactions between preprocessor features and identify techniques for correctly handling them. Notably, we show that a configuration-preserving preprocessor needs to hoist conditionals around other preprocessor operations, since preprocessor operations cannot be composed with conditionals. In exploring configuration-preserving parsing, we focus on performance. We present a simple algorithm for *Fork-Merge LR* (FMLR) parsing, which not only subsumes MAPR's algorithm but also has well-defined hooks for optimization. We then introduce four such optimizations, which decrease the number of forked subparsers (the *token follow set* and *lazy shifts*), eliminate duplicate work done by subparsers (*shared reduces*), and let subparsers merge as soon as possible (*early reduces*).

Our optimizations are not only applied automatically, they also subsume TypeChef's specialized join combinators. The result is compelling. SuperC, our open-source tool[1] implementing these techniques, can fully parse programs with high variability, notably the entire x86 Linux kernel. In contrast, TypeChef can only parse a constrained version and MAPR fails for most source files.

Like MAPR, our work is inspired by GLR parsing [63], which also forks and merges subparsers. But whereas GLR parsers match different productions to the same input fragment, FMLR matches the same production to different input fragments. Furthermore, unlike GLR and TypeChef, FMLR parsers can reuse existing LR grammars and parser table generators; only the parser engine is new. This markedly decreases the engineering effort necessary for adapting our work. Compared to previous work, this paper makes the following contributions:

- An analysis of the challenges involved in parsing C with arbitrary preprocessor usage and an empirical quantification for the x86 version of the Linux kernel.

- A comprehensive treatment of techniques for configuration-preserving preprocessing and parsing, including novel performance optimizations.

- SuperC, an open-source tool for parsing all of C, and its demonstration on the x86 Linux kernel.

Overall, our work solves the problem of how to completely and efficiently parse all of C, 40 years after invention of the language, and thus lays the foundation for building more powerful C analysis and transformation tools.

[1] `http://cs.nyu.edu/xtc/`.

1.2 The Problem and Solution Approach

(a) The unpreprocessed source.

```
1 #include "major.h" // Defines
      MISC_MAJOR to be 10
2
3 #define MOUSEDEV_MIX 31
4 #define MOUSEDEV_MINOR_BASE 32
5
6 static int mousedev_open(struct
      inode *inode, struct file *file
      )
7 {
8   int i;
9
10 #ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
11   if (imajor(inode) == MISC_MAJOR)
12     i = MOUSEDEV_MIX;
13   else
14 #endif
15     i = iminor(inode) -
           MOUSEDEV_MINOR_BASE;
16
17   return 0;
18 }
```

C compilers such as gcc process only one variant of the source code at a time. They pick the one branch of each static conditional that matches the *configuration variables* passed to the preprocessor, e.g., through the "-D" command line option. Different configuration variable settings, or *configurations*, result in different executables, all from the same C sources. In contrast, other C tools, such as source browsers, bug finders, and automated refactorings, need to be *configuration-preserving*. They need to process all

(b) The preprocessed source preserving all configurations.

```
1  static int mousedev_open(struct
       inode *inode, struct file *file
       )
2  {
3    int i;
4
5  #ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
6    if (imajor(inode) == 10)
7      i = 31;
8    else
9  #endif
10     i = iminor(inode) - 32;
11
12   return 0;
13 }
```

(c) Sketch of the AST containing all configurations.

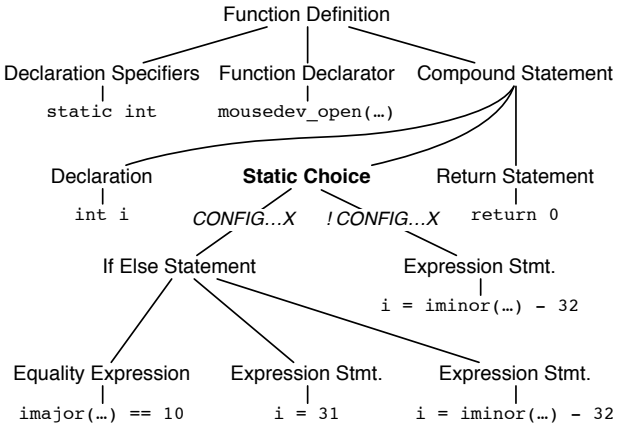


Figure 1.1: From source code to preprocessed code to AST. The example is edited down for simplicity from drivers/input/mousedev.c.

Language Construct	Implementation	Surrounded by Conditionals	Contain Conditionals	Contain Multiply-Defined Macros	Other
Lexer					
Layout	Annotate tokens	n/a	n/a	n/a	n/a

(a) Lexer interactions

Table 1.1: Interactions between C preprocessor and language features. Gray entries in the last three columns are newly supported by SuperC.

branches of static conditionals and, for each branch, track the configurations enabling the branch, i.e., its *presence condition*. This considerably complicates C tools except compilers, starting with preprocessing and parsing.

Figure 1.1 illustrates SuperC's configuration-preserving preprocessing and parsing on a simple example from the x86 Linux kernel (version 2.6.33.3, which is used throughout this paper). Figure 1.1a shows the original source code, which utilizes the three main preprocessor facilities: an include directive on line 1, macro definitions on lines 3 and 4, and conditional directives on lines 10 and 14. The code has two configurations, one when "CONFIG_INPUT_MOUSEDEV_PSAUX" is defined and one when it is not defined. After preprocessing, shown in Figure 1.1b, the header file has been included (not shown) and the macros have been expanded on lines 6, 7, and 10, but the conditional directives remain on lines 5 and 9. Finally, in Figure 1.1c, the parser has generated an AST containing both configurations with a static choice node corresponding to the static conditional on lines 5–9 in Figure 1.1b.

1.2.1 Interactions Between C and the Preprocessor

The complexity of configuration-preserving C processing stems from the interaction of preprocessor features with each other and with the C language. Table 1.1 summarizes these interactions. Rows denote language features and are grouped by the three steps of C processing: lexing, preprocessing, and parsing. The first column names the feature and the second column describes the implementation strategy. The remaining columns capture complications arising from the interaction of features, and the corresponding table entries indicate how to overcome the complications. Blank entries indicate impossible interactions.

Language Construct	Implementation	Surrounded by Conditionals	Contain Conditionals	Contain Multiply-Defined Macros	Other
Preprocessor					
Macro (Un)Definition	Use conditional macro table	Add multiple entries to macro table	n/a	Do not expand until invocation	Trim infeasible entries on redefinition
Object-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions	n/a	Expand nested macros	Get ground truth for built-ins from compiler
Function-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions	Hoist conditionals around invocations	Expand nested macros	Support differing argument numbers and variadics
Token Pasting & Stringification	Apply pasting & stringification		Hoist conditionals around token pasting & stringification		n/a
File Includes	Include and preprocess files	Preprocess under presence conditions	n/a	Hoist conditionals around includes	Reinclude when guard macro is not false
Static Conditionals	Preprocess all branches	Conjoin presence conditions		Ignore infeasible definitions	n/a
Conditional Expressions	Evaluate presence conditions	n/a	n/a	Hoist conditionals around expressions	Preserve order for non-boolean expressions
Error Directives	Ignore erroneous branches		n/a	n/a	n/a
Line, Warning, & Pragma Directives	Treat as layout	n/a	n/a	n/a	n/a

(b) Preprocessor interactions

Table 1.1: Interactions between C preprocessor and language features. Gray entries in the last three columns are newly supported by SuperC.

Language Construct	Implementation	Surrounded by Conditionals	Contain Conditionals	Contain Multiply-Defined Macros	Other
Parser					
C Constructs	Use FMLR Parser	Fork and merge subparsers		n/a	n/a
Typedef Names	Use conditional symbol table	Add multiple entries to symbol table	n/a	n/a	Fork subparsers on ambiguous names

(c) Parser interactions

Table 1.1: Interactions between C preprocessor and language features. Gray entries in the last three columns are newly supported by SuperC.

Gray entries highlight interactions not yet supported by TypeChef. In contrast, SuperC does address all interactions—besides annotating tokens with layout and with line, warning, and pragma directives. (We have removed a buggy implementation of these annotations from SuperC for now.)

Layout. The first step is lexing. The lexer converts raw program text into tokens, stripping layout such as whitespace and comments. Since lexing is performed before preprocessing and parsing, it does not interact with the other two steps. However, automated refactorings, by definition, restructure source code and need to output program text *as originally written*, modulo any intended changes. Consequently, they need to annotate tokens with surrounding layout—plus, keep sufficient information about preprocessor operations to restore them as well.

```
1 #ifdef CONFIG_64BIT
2 #define BITS_PER_LONG 64
3 #else
4 #define BITS_PER_LONG 32
5 #endif
```

Figure 1.2: A multiply-defined macro from include/asm-generic/bitsperlong.h.

Macro (un)definitions. The second step is preprocessing. It collects macro definitions (#define) and undefinitions (#undef) in a macro table—with definitions being either object-like

"#define" *name body*

or function-like

"#define" *name*"("*parameters*")" *body*

Definitions and undefinitions for the same macro may appear in different branches of static conditionals, creating a *multiply-defined macro* that depends on the configuration. Figure 1.2 shows such a macro, "BITS_PER_LONG", whose definition depends on the "CONFIG_64BIT" configuration variable. A configuration-preserving preprocessor records *all* definitions in its macro table, tagging each entry with the presence condition of the "#define" directive while also removing infeasible entries on each update. The preprocessor also records undefinitions, so that it can determine which macros are neither defined nor undefined and thus *free*, i.e., configuration variables. Wherever multiply-defined macros are used, they propagate an *implicit conditional*. It is as if the programmer had written an explicit conditional in the first place—an observation first made by Garrido and Johnson [34].

Macro invocations. Since macros may be nested within each other, a configuration-preserving preprocessor, just like an ordinary preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such as "__STDC_VERSION__" to indicate the version of the C standard, the preprocessor needs to be configured with the ground truth of the targeted compiler.

Beyond these straightforward issues, a configuration-preserving preprocessor needs to handle two more subtle interactions. First, a macro invocation may be surrounded by static conditionals. Consequently, the preprocessor needs to ignore macro definitions that are infeasible for the presence condition of the invocation site. Second, function-like macro invocations may contain conditionals, either explicitly in source code or implicitly through multiply-defined macros. These conditionals can alter the function-like macro invocation by changing its name or arguments, including their number and values. To preserve the function-like invocation while also allowing for differing argument numbers and variadics (a gcc extension) in different conditional branches, the preprocessor needs to *hoist* the conditionals around the invocation.

```
1  // In include/linux/byteorder/little_endian.h:
2  #define __cpu_to_le32(x) ((__force __le32)(__u32)(x))
3
4  #ifdef __KERNEL__
5  // Included from include/linux/byteorder/generic.h:
6  #define cpu_to_le32 __cpu_to_le32
7  #endif
8
9  // In drivers/pci/proc.c:
10 _put_user(cpu_to_le32(val), (__le32 __user *) buf);
```

Figure 1.3: A macro conditionally expanding to another macro.

(a) After expansion of `cpu_to_le32`.

```
1 #ifdef __KERNEL__
2 __cpu_to_le32
3 #else
4 cpu_to_le32
5 #endif
6 (val)
```

(b) After hoisting the conditional.

```
1 #ifdef __KERNEL__
2 __cpu_to_le32(val)
3 #else
4 cpu_to_le32(val)
5 #endif
```

(c) After expansion of `__cpu_to_le32`.

```
1 #ifdef __KERNEL
2 ((__force __le32)(__u32)(val))
3 #else
4 cpu_to_le32(val)
5 #endif
```

Figure 1.4: Preprocessing `cpu_to_le32(val)` in Fig. 1.3:10.

Figures 1.3 and 1.4 illustrate the hoisting of conditionals. Figure 1.3 contains a sequence of tokens on line 10, "cpu_to_le32(val)", which either expands to an invocation of the function-like macro "__cpu_to_le32", if "__KERNEL__" is defined, or denotes the invocation of the C function "cpu_to_le32", if "__KERNEL__" is not defined. Figure 1.4 shows the three stages of preprocessing the sequence. First, in 1.4a, the preprocessor expands "cpu_to_le32", which makes the conditional explicit but also breaks the nested macro invocation on line 2. Second, in 1.4b, the preprocessor hoists the conditional around the entire sequence of tokens, which duplicates "(val)" in each branch and thus restores the invocation on

line 2. Third, in 1.4c, the preprocessor recursively expands "__cpu_to_le32" on line 2, which completes preprocessing for the sequence.

(a) The macro definitions and invocation.

```
1 #define uintBPL_t uint(
      BITS_PER_LONG)
2 #define uint(x) xuint(x)
3 #define xuint(x) __le ## x
4
5 uintBPL_t *p = $\ldots\,\,$;
```

(b) After expanding the macros.

```
1 __le ##
2 #ifdef CONFIG_64BIT
3 64
4 #else
5 32
6 #endif
7 *p = $\ldots\,\,$;
```

(c) After hoisting the conditional.

```
1 #ifdef CONFIG_64BIT
2 __le ## 64
3 #else
4 __le ## 32
5 #endif
6 *p = $\ldots\,\,$;
```

Figure 1.5: A token-pasting example from fs/udf/balloc.c.

Token-pasting and stringification. Macros may contain two operators that modify tokens: The infix token-pasting operator "##" concatenates two tokens, and the prefix stringification operator "#" converts a

sequence of tokens into a string literal. The preprocessor simply applies these operators, with one complication: the operators' arguments may contain conditionals, either explicitly in source code or implicitly via multiply-defined macros. As for function-like macros, a configuration-preserving preprocessor needs to hoist conditionals around these operators. Figure 1.5 illustrates this for token-pasting: 1.5a shows the source code; 1.5b shows the result of expanding all macros, including "BITS_PER_LONG" from Figure 1.2; and 1.5c shows the result of hoisting the conditional out of the token-pasting.

File includes. To produce complete compilation units, a configuration-preserving preprocessor recursively resolves file includes ("#include"). If the directive is nested in a static conditional, the preprocessor needs to process the header file under the corresponding presence condition. Furthermore, if a guard macro, which is traditionally named "FILENAME_H" and protects against multiple inclusion, has been undefined, the preprocessor needs to process the same header file again. More interestingly, include directives may contain macros that provide part of the file name. If the macro in such a *computed include* is multiply-defined, the preprocessor needs to hoist the implicit conditional out of the directive, just as for macro invocations, token-pasting, and stringification.

Conditionals. Static conditionals enable multiple configurations, so both configuration-preserving preprocessor and parser need to process all branches. The preprocessor converts static conditionals' expressions into presence conditions, and when conditionals are nested within each other, conjoins nested conditionals' presence conditions. As described for macro invocations above, this lets the preprocessor ignore infeasible definitions during expansion of multiply-defined macros.

However, two issues complicate the conversion of conditional expressions into presence conditions. First, a conditional expression may contain arbitrary macros, not just configuration variables. So the preprocessor needs to expand the macros, which may be multiply-defined. When expanding a multiply-defined macro, the preprocessor needs to convert the macro's implicit conditional into logical form and hoist it around the conditional expression. For example, when converting the conditional expression

"BITS_PER_LONG == 32"

from kernel/sched.c into a presence condition, the preprocessor expands the definition of "BITS_PER_LONG" from Figure 1.2 and hoists it around the conditional expression, to arrive at

```
        "defined(CONFIG_64BIT) && 64 == 32"    \
    "|| !defined(CONFIG_64BIT) && 32 == 32"
```

which makes testing for "CONFIG_64BIT" explicit with the "defined" operator and simplifies to

```
    "!defined(CONFIG_64BIT)"
```

after constant folding.

Second, configuration variables may be non-boolean and conditional expressions may contain arbitrary arithmetic subexpressions, such as "NR_CPUS" "<" "256" (from arch/x86/include/asm/spinlock.h). Since there is no known efficient algorithm for comparing arbitrary polynomials [40], such subexpressions prevent the preprocessor from trimming infeasible configurations. Instead, it needs to treat non-boolean subexpressions as opaque text and preserve their branches' source code ordering, i.e., never omit or combine them and never move other branches across them.

Other preprocessor directives. The C preprocessor supports four additional directives, to issue errors ("#error") and warnings ("#warning"), to instruct compilers ("#pragma"), and to overwrite line numbers ("#line"). A configuration-preserving preprocessor simply reports errors and warnings, and also terminates for errors appearing outside static conditionals. More importantly, it treats conditional branches containing error directives as infeasible and disables their parsing. Otherwise, it preserves such directives as token annotations to support automated refactorings.

```
1 static int (*check_part[])(struct parsed_partitions *) = {
2 #ifdef CONFIG_ACORN_PARTITION_ICS
3   adfspart_check_ICS,
4 #endif
5 #ifdef CONFIG_ACORN_PARTITION_POWERTEC
6   adfspart_check_POWERTEC,
7 #endif
8 #ifdef CONFIG_ACORN_PARTITION_EESOX
9   adfspart_check_EESOX,
10 #endif
11   // 15 more, similar initializers
12   NULL
13 };
```

Figure 1.6: An example of a C construct containing an exponential number of unique configurations from fs/partitions/check.c.

C constructs. The third and final step is parsing. The preprocessor produces entire compilation units, which may contain static conditionals but no other preprocessor operations. The configuration-preserving parser processes all branches of each conditional by *forking* its internal state into subparsers and *merging* the subparsers again after the conditional. This way, it produces an AST containing all configurations, with static choice nodes for conditionals.

One significant complication is that static conditionals may still appear between arbitrary tokens, thus violating C syntax. However, the AST may only contain nodes representing complete C constructs. To recognize C constructs with embedded configurations, the parser may require a subparser per configuration. For example, the statement on lines 5–10 in Figure 1.1b has two configurations and requires two subparsers. The parser may also parse tokens shared between configurations several times. In the example, line 10 is parsed twice, once as part of the if-then-else statement and once as a stand-alone expression statement. This way, the parser hoists conditionals out of C constructs, much like the preprocessor hoists them out of preprocessor operations.

Using a subparser per embedded configuration is acceptable for most declarations, statements, and

expressions. They have a small number of terminals and nonterminals and thus can contain only a limited number of configurations. However, if a C construct contains repeated nonterminals, this can lead to an exponential blow-up of configurations and therefore subparsers. For example, the array initializer in Figure 1.6 has 2^{18} unique configurations. Using a subparser for each configuration is clearly infeasible and avoiding it requires careful optimization of the parsing algorithm (Section 1.4).

Typedef names. A final complication results from the fact that C syntax is context-sensitive [55]. Depending on context, names can either be typedef names, i.e., type aliases, or they can be object, function, and "enum" constant names. Furthermore, the same code snippet can have fundamentally different semantics, depending on names. For example, "T" "*" "p;" is either a *declaration* of "p" as a pointer to type "T" or an *expression statement* that multiplies the variables "T" and "p", depending on whether "T" is a typedef name. C parsers usually employ a symbol table to disambiguate names [38, 55]. In the presence of conditionals, however, a name may be both. Consequently, a configuration-preserving parser needs to maintain configuration-dependent symbol table entries and fork subparsers when encountering an implicit conditional due to an ambiguously defined name.

1.3 The Configuration-Preserving Preprocessor

SuperC's configuration-preserving preprocessor accepts C files, performs all operations while preserving static conditionals, and produces compilation units. While tedious to engineer, its functionality mostly follows from the discussion in the previous section. Two features, however, require further elaboration: the hoisting of conditionals around preprocessor operations and the conversion of conditional expressions into presence conditions.

1.3.1 Hoisting Static Conditionals

Preprocessor directives as well as function-like macro invocations, token-pasting, and stringification may only contain ordinary language tokens. Consequently, they are ill-defined in the presence of implicit or explicit embedded static conditionals. To perform these preprocessor operations, SuperC's configuration-preserving preprocessor needs to hoist conditionals, so that only ordinary tokens appear in the branches of the innermost conditionals.

Algorithm 1 Hoisting Conditionals. Hoist takes a presence condition c and a list τ of tokens and entire conditionals.

```
 1: procedure Hoist(c, τ)
 2:     ▷ Initialize a new conditional with an empty branch.
 3:     C ← [ (c, •) ]
 4:     for all a ∈ τ do
 5:         if a is a language token then
 6:             ▷ Append a to all branches in C.
 7:             C ← [ (c_i, τ_i a) | (c_i, τ_i) ∈ C ]
 8:         else ▷ a is a conditional.
 9:             ▷ Recursively hoist conditionals in each branch.
10:             B ← [ b | b ∈ Hoist(c_i, τ_i) and (c_i, τ_i) ∈ a ]
11:             ▷ Combine with already hoisted conditionals.
12:             C ← C × B
13:         end if
14:     end for
15:     return C
16: end procedure
```

Algorithm 1 formally defines Hoist. It takes a presence condition c and a list of ordinary tokens and entire conditionals τ under the presence condition. Each static conditional C, in turn, is treated as a list of branches

$$C := [\,(c_1, \tau_1), \ldots, (c_n, \tau_n)\,]$$

with each branch having a presence condition c_i and a list of tokens and nested conditionals τ_i. Line 3 initializes the result C with an empty conditional branch. Lines 4–14 iterate over the tokens and conditionals in τ, updating C as necessary. And line 15 returns the result C. Lines 5–7 of the loop handle ordinary tokens, which are present in all embedded configurations and are appended to all branches in C, as illustrated for "(val)" in Figure 1.4b and for "__le" "##" in Figure 1.5c. Lines 8–13 of the loop handle conditionals by recursively hoisting any nested conditionals in line 10 and then combining the result B with C in line 12. The cross product for conditionals in line 12 is defined as

$$C \times B := [\,(c_i \wedge c_j, \tau_i \tau_j) \mid (c_i, \tau_i) \in C \text{ and } (c_j, \tau_j) \in B\,]$$

and generalizes line 7 by combining every branch in C with every branch in B.

For example, in Figure 1.5b, `__le ##` interacts with a static conditional. Both tokens `__le` and `##` are appended to each branch in C, which starts with just one empty branch. After appending,

C = [(**true** , `__le ##`)]

Then, when Hoist encounters the static conditional, each branch just contains a single token, so B is

B = [(`CONFIG_64BIT`, `64`), (¬`CONFIG_64BIT`, `32`)]

The cross produce combines every branch from C with every branch of B, computing the conjoined condition for the resulting branches. $C \times B$ is then

[(**true** ∧ `CONFIG_64BIT`, `__le ## 64`), (**true** ∧¬`CONFIG_64BIT`, `__le ## 32`)]

SuperC uses Hoist for all preprocessor operations that may contain conditionals except for function-like macro invocations. The problem with the latter is that, to call Hoist, the preprocessor needs to know which tokens and conditionals belong to an operation. But different conditional branches of a function-like macro invocation may contain different macro names and numbers of arguments, and even additional, unrelated tokens. Consequently, SuperC uses a version of Hoist for function-like macro invocations that interleaves parsing with hoisting. For each conditional branch, it tracks parentheses and commas, which change the parsing state of the invocation. Once all variations of the invocation have been recognized across all conditional branches, each invocation is separately expanded. If a variation contains an object-like or undefined macro, the argument list is left in place, as illustrated in Fig. 1.4c:4.

1.3.2 Converting Conditional Expressions

To reason about presence conditions, SuperC converts conditional expressions into Binary Decision Diagrams (BDDs) [20, 67], which are an efficient, symbolic representation of boolean functions. BDDs include support for boolean constants, boolean variables, as well as negation, conjunction, and disjunction. On top of that, BDDs are *canonical*: Two boolean functions are the same if and only if their BDD representations are the same [20]. This makes it not only possible to directly combine BDDs, e.g., when tracking the presence conditions of nested or hoisted conditionals, but also to easily compare two BDDs for equality, e.g., when testing for an infeasible configuration by evaluating $c_1 \wedge c_2 = \mathit{false}$.

Before converting a conditional expression into a BDD, SuperC expands any macros outside invocations of the "defined" operator, hoists multiply-defined macros around the expression, and performs constant folding. The resulting conditional expression uses negations, conjunctions, and disjunctions to

combine four types of subexpressions: constants, free macros, arithmetic expressions, and "defined" invocations. SuperC converts each of these subexpressions into a BDD as follows and then combines the resulting BDDs with the necessary logical operations:

1. A constant translates to *false* if zero and to *true* otherwise.
2. A free macro translates to a BDD variable.
3. An arithmetic subexpression also translates to a BDD variable.
4. "defined("*M*")" translates into the disjunction of presence conditions under which *M* is defined. However, if *M* is free:
   (a) If *M* is a guard macro, "defined("*M*")" translates to *false*.
   (b) Otherwise, "defined("*M*")" translates to a BDD variable.

Just like gcc, Case 4a treats *M* as a guard macro, if a header file starts with a conditional directive that tests "!defined("*M*")" and is followed by "#define" *M*, and the matching "#endif" ends the file. To ensure that repeated occurrences of the same free macro, arithmetic expression, or "defined("*M*")" for free *M* translate to the same BDD variable, SuperC maintains a mapping between these expressions and their BDD variables. In the case of arithmetic expressions, it normalizes the text by removing whitespace and comments.

1.4 The Configuration-Preserving Parser

SuperC's configuration-preserving FMLR parser builds on LR parsing [6, 45], a bottom-up parsing technique. To recognize the input, LR parsers maintain an explicit parser stack, which contains terminals, i.e., tokens, and nonterminals. On each step, LR parsers perform one of four actions: (1) *shift* to copy a token from the input onto the stack and increment the parser's position in the input, (2) *reduce* to replace one or more top-most stack elements with a nonterminal, (3) *accept* to successfully complete parsing, and (4) *reject* to terminate parsing with an error. The choice of action depends on both the next token in the input and the parser stack. To ensure efficient operation, LR parsers use a deterministic finite control and store the state of the control with each stack element.

Algorithm 2 Fork-Merge LR Parsing. Parse takes a_0, the first token or static conditional of the program. Q is a priority queue of subparser triples (c, a, s), where c is the presence condition, a is the head, and s is the parsing state stack. T represents a token follow set for a given subparser.

```
1:  procedure Parse(a_0)
2:      Q.init((true, a_0, s_0)) ▷ The initial subparser for a_0.
3:      while Q ≠ ∅ do
4:          p ← Q.pull() ▷ Step the next subparser p.
5:          T ← Follow(p.c, p.a)
6:          if |T| = 1 then
7:              ▷ Do an LR action and reschedule the subparser.
8:              Q.insert(LR(T(1), p))
9:          else ▷ The follow-set contains several tokens.
10:             ▷ Fork subparsers and reschedule them.
11:             Q.insertAll(∀(c_i, a_i) ∈ Fork(T, p))
12:         end if
13:         Q ← Merge(Q)
14:     end while
15: end procedure
```

Compared to top-down parsing techniques, such as LL [54] and PEG [14, 30], LR parsers are an attractive foundation for configuration-preserving parsing for three reasons. First, LR parsers make the parsing state explicit, in form of the parser stack. Consequently, it is easy to fork the parser state on a static conditional, e.g., by representing the stack as a singly-linked list and by creating new stack elements that point to the shared remainder. Second, LR parsers are relatively straight-forward to build, since most of the complexity lies in generating the parsing tables, which determine control transitions and actions. In fact, SuperC uses LALR parsing tables [24] produced by an existing parser generator. Third, LR parsers support left-recursion in addition to right-recursion, which is helpful for writing programming language grammars.

1.4.1 Fork-Merge LR Parsing

Algorithm 2 formalizes FMLR parsing. It uses a queue Q of LR subparsers p. Each subparser $p := (c, a, s)$ has a presence condition c, a token or static conditional a that comes next in the input (also called the *head*) and an LR parsing state stack s. Each subparser recognizes a distinct configuration, i.e., the different presence conditions $p.c$ are mutually exclusive, and all subparsers together recognize all configurations, i.e., the disjunction of all their presence conditions is *true*. Q is a priority queue, ordered by the position

of the head $p.a$ in the input. This ensures that subparsers merge at the earliest opportunity, as no subparser can outrun the other subparsers.

Line 2 initializes the queue Q with the subparser for the initial token or conditional a_0, and lines 3–14 step individual subparsers until the queue is empty, i.e., all subparsers have accepted or rejected. On each iteration, line 4 pulls the earliest subparser p from the queue. Line 5 computes the *token follow-set* for $p.c$ and $p.a$, which contains pairs (c_i, a_i) of ordinary language tokens a_i and their presence conditions c_i. The follow-set computation is detailed in Section 1.4.2. Intuitively, it captures the actual variability of source code and includes the first language token on each path through static conditionals from the current input position. If the follow-set contains a single element, e.g., $p.a$ is an ordinary token and $T = \{(p.c, p.a)\}$, lines 6–8 perform an LR action on the only element $T(1)$ and the subparser p. Unless the LR action is *accept* or *reject*, line 8 also reschedules the subparser. Otherwise, the follow-set contains more than one element, e.g., $p.a$ is a conditional. Since each subparser can only perform LR actions one after another, lines 9–12 fork a subparser for each presence condition and token $(c_i, a_i) \in T$ and then reschedule the subparsers. Finally, line 13 tries to merge subparsers again. Subparsers may merge if they have the same head and LR stack, which ensures that conditionals are hoisted out of C constructs.

1.4.2 The Token Follow-Set

A critical challenge for configuration-preserving parsing is *which* subparsers to create. The naive strategy, employed by MAPR, forks a subparser for every branch of every static conditional. But conditionals may have empty branches and even omit branches, like the implicit else branch in Figure 1.1. Furthermore, they may be directly nested within conditional branches, and they may directly follow other conditionals. Consequently, the naive strategy forks a great many unnecessary subparsers and is intractable for complex C programs such as Linux. Instead, FMLR relies on the token follow-set to capture the source code's actual variability, thus limiting the number of forked subparsers.

Algorithm 3 The Token Follow-Set

```
1:  procedure Follow(c, a)
2:      T ← ∅   ▷ Initialize the follow-set.
3:      procedure First(c, a)
4:          loop
5:              if a is a language token then
6:                  T ← T ∪ {(c, a)}
7:                  return false
8:              else ▷ a is a conditional.
9:                  c_r ← false   ▷ Initialize remaining condition.
10:                 for all (c_i, τ_i) ∈ a do
11:                     if τ_i = • then
12:                         c_r ← c_r ∨ c ∧ c_i
13:                     else
14:                         c_r ← c_r ∨ First(c ∧ c_i, τ_i(1))
15:                     end if
16:                 end for
17:                 if c_r = false or a is last element in branch then
18:                     return c_r
19:                 end if
20:                 c ← c_r
21:                 a ← next token or conditional after a
22:             end if
23:         end loop
24:     end procedure
25:     loop
26:         c ← First(c, a)
27:         if c = false then return T end if ▷ Done.
28:         a ← next token or conditional after a
29:     end loop
30: end procedure
```

Algorithm 3 formally defines Follow. It takes a presence condition c and a token or conditional a, and it returns the follow-set T for a, which contains pairs (c_i, a_i) of ordinary tokens a_i and their presence conditions c_i. By construction, each token a_i appears exactly once in T; consequently, the follow-set is *ordered* by the tokens' positions in the input. Line 2 initializes T to the empty set. Lines 3–24 define the nested procedure First. It scans well-nested conditionals and adds the first ordinary token and presence condition for each configuration to T. It then returns the presence condition of any remaining configuration, i.e., conditional branches that are empty or implicit and thus do not contain ordinary tokens. Lines 25–29 repeatedly call First until all configurations have been covered, i.e., the remaining configu-

ration is *false*. Line 28 moves on to the next token or conditional, while also stepping out of conditionals. In other words, if the token or conditional a is the last element in the branch of a conditional, which, in turn, may be the last element in the branch of another conditional (and so on), line 28 updates a with the first element after the conditionals.

First does the brunt of the work. It takes a token or conditional a and presence condition c. Lines 4–23 then iterate over the elements of a conditional branch or at a compilation unit's top-level, starting with a. Lines 5–7 handle ordinary language tokens. Line 6 adds the token and presence condition to the follow-set T. Line 7 terminates the loop by returning *false*, indicating no remaining configuration. Lines 8–22 handle conditionals. Line 9 initializes the remaining configuration c_r to *false*. Lines 10–16 then iterate over the branches of the conditional a, including any implicit branch. If a branch is empty, line 12 adds the conjunction of its presence condition c_i and the overall presence condition c to the remaining configuration c_r. Otherwise, line 14 recurses over the branch, starting with the first token or conditional $\tau_i(1)$, and adds the result to the remaining configuration c_r. If, after iterating over the branches of the conditional, the remaining configuration is *false* or there are no more tokens or conditionals to process, lines 17–19 terminate First's main loop by returning c_r. Finally, lines 20–21 set up the next iteration of the loop by updating c with the remaining configuration and a with the next token or conditional.

1.4.3 Forking and Merging

Figure 1.7a shows the definitions of Fork and Merge. Fork creates new subparsers from a token follow-set T to replace a subparser p. Each new subparser has a different presence condition c and token a from the follow-set T but the same LR stack $p.s$. Consequently, it recognizes a more specific configuration than the original subparser p. Merge has the opposite effect. It takes the priority queue Q and combines any subparsers $p \in Q$ that have the same head and LR stack. Such subparsers are redundant: they will necessarily perform the same parsing actions for the rest of the input, since FMLR, like LR, is deterministic. Each merged subparser replaces the original subparsers; its presence condition is the disjunction of the original subparsers' presence conditions. Consequently, it recognizes a more general configuration than any of the original subparsers. Merge is similar to GLR's *local ambiguity packing* [63], which also combines equivalent subparsers, except that FMLR subparsers have presence conditions.

(a) Basic forking and merging.

$$\text{Fork}(T, p) := \{\, (c, a, p.s) \mid (c, a) \in T \,\}$$

$$\text{Merge}(Q) := \{\, (\bigvee p.c, a, s) \mid a = p.a \text{ and } s = p.s \;\forall p \in Q \,\}$$

(b) Optimized forking.

$$\text{Fork}(T, p) := \{\, (H, p.s) \mid H \in \text{Lazy}(T, p) \cup \text{Shared}(T, p) \,\}$$

$$\text{Lazy}(T, p) := \{\, \bigcup \{(c, a)\} \mid \text{Action}(a, p.s) = \text{`}\texttt{shift}\text{'} \;\forall (c, a) \in T \,\}$$

$$\text{Shared}(T, p) :=$$
$$\{\, \bigcup \{(c, a)\} \mid \text{Action}(a, p.s) = \text{`}\texttt{reduce}\; n\text{'} \;\forall (c, a) \in T \,\}$$

Figure 1.7: The definitions of fork and merge.

1.4.4 Optimizations

In addition to the token follow-set, FMLR relies on three more optimizations to contain the state explosion caused by static conditionals: early reduces, lazy shifts, and shared reduces. *Early reduces* are a tie-breaker for the priority queue. When subparsers have the same head a, they favor subparsers that will reduce over subparsers that will shift. Since reduces, unlike shifts, do not change a subparser's head, early reduces prevent subparsers from outrunning each other and create more opportunities for merging subparsers.

While early reduces seek to increase merge opportunities, lazy shifts and shared reduces seek to decrease the number and work of forked subparsers, respectively. First, *lazy shifts* delay the forking of subparsers that will shift. They are based on the observation that a sequence of static conditionals with empty or implicit branches, such as the array initializer in Figure 1.6, often results in a follow-set, whose tokens all require a shift as the next LR action. However, since FMLR steps subparsers by position of the head, the subparser for the first such token performs its shift (plus other LR actions) and can merge again *before* the subparser for the second such token can even perform its shift. Consequently, it is wasteful

to eagerly fork the subparsers. Second, *shared reduces* reduce a single stack for several heads at the same time. They are based on the observation that conditionals often result in a follow-set, whose tokens all require a reduce to the same nonterminal; e.g., both tokens in the follow-set of the conditional in Figure 1.1b reduce the declaration on line 3. Consequently, it is wasteful to first fork the subparsers and then reduce their stacks in the same way.

Figure 1.7b formally defines both lazy shifts and shared reduces. Both optimizations result in *multi-headed* subparsers $p := (H, s)$, which have more than one head and presence condition

$$H := \{(c_1, a_1), \ldots, (c_n, a_n)\}$$

Just as for the follow-set, each token a_i appears exactly once in H and the set is ordered by the tokens' positions in the input. Algorithm 2 generalizes to multi-headed subparsers as follows. It prioritizes a multi-headed subparser by its earliest head a_1. Next, by definition of optimized forking, the follow-set of a multi-headed subparser (H, s) is H. However, the optimized version of the FMLR algorithm always performs an LR operation on a multi-headed subparser, i.e., treats it as if the follow-set contains a single ordinary token. If the multi-headed subparser will shift, it forks off a single-headed subparser p' for the earliest head, shifts p', and then reschedules both subparsers. If the multi-headed subparser will reduce, it reduces p and immediately recalculates $\mathrm{Fork}(H, p)$, since the next LR action may not be the same reduce for all heads anymore. Finally, it merges multi-headed subparsers p if they have the same head $\{(\_\_, a_1), \ldots, (\_\_, a_n)\} = p.H$ and the same LR stack $s = p.s$; it computes the merged parser's presence conditions as the disjunction of the original subparser's corresponding presence conditions $c_i = \bigvee p.H(i).c$.

1.4.5 Putting It All Together

We are now ready to illustrate FMLR on the array initializer in Figure 1.6. For simplicity, we treat "NULL" as a token and ignore that the macro usually expands to "((void" "*)0)". For concision, we subscript each subparser and set symbol with its current line number in Figure 1.6. We also use b_n to denote the boolean variable representing the conditional expression on line n, e.g.,

$$b_2 \sim \text{"defined(CONFIG\_ACORN\_PARTITION\_ICS)"}$$

Finally, we refer to one iteration through FMLR's main loop in Algorithm 2 as a *step*.

Since line 1 in Figure 1.6 contains only ordinary tokens, FMLR behaves like an LR parser, stepping through the tokens with a single subparser p_1. Upon reaching line 2, FMLR computes Follow for the conditional on lines 2–4. To this end, First iterates over the conditionals and "NULL" token in the initializer list by updating a in Alg. 3:21. On each iteration besides the last, First also recurses over the branches of a conditional, including the implicit else branch. As a result, it updates the remaining configuration in Alg. 3:12 with a conjunction of negated conditional expressions, yielding the follow-set

$$T_2 = \{\,(b_2,\ \text{"adfspart\_check\_ICS"}),\\ (\neg b_2 \wedge b_5,\ \text{"adfspart\_check\_POWERTEC"}),\\ \ldots,\ (\neg b_2 \wedge \neg b_5 \wedge \neg b_8 \wedge \ldots,\ \text{"NULL"})\,\}$$

Since all tokens in T_2 reduce the empty input to the *InitializerList* nonterminal, *shared reduces* turns p_2 into a multi-headed subparser with $H_2 = T_2$. FMLR then steps p_3. It reduces the subparser, which does not change the heads, i.e., $H_3 = H_2$, but modifies the stack to

$$p_3.s = \ldots\ \text{"\{"}\ \mathit{InitializerList}$$

It then calculates $\text{Fork}(H_3, p_3)$; since all tokens in H_3 now shift, *lazy shifts* produces the same multi-headed subparser. FMLR steps p_3 again. It forks off a single-headed subparser p'_3 and shifts the identifier token on line 3 onto its stack. Next, FMLR steps p'_3. It shifts the comma token onto the stack, which yields

$$p'_3.s = \ldots\ \text{"\{"}\ \mathit{InitializerList}\ \text{"}\mathit{adfspart\_check\_ICS}\text{"}\ \text{","}$$

and updates the head $p'_3.a$ to the conditional on lines 5–7. FMLR steps p'_5 again, computing the subparser's follow-set as

$$T'_5 = \{\,(b_2 \wedge b_5,\ \text{"adfspart\_check\_POWERTEC"}),\\ \ldots,\ (b_2 \wedge \neg b_5 \wedge \neg b_8 \wedge \ldots,\ \text{"NULL"})\,\}$$

Since all tokens in T'_5 reduce the top three stack elements to an *InitializerList*, *shared reduces* turns p'_5 into a multi-headed subparser with $H'_5 = T'_5$. At this point, both p_6 and p'_6 are multi-headed subparsers with the same heads, though their stacks differ. Due to *early reduces*, FMLR steps p'_6. It reduces the stack, which yields the same stack as that of p_6, and calculates Fork, which does not change p'_6 due to *lazy shifts*. It then merges the two multi-headed subparsers, which disjoins b_2 with $\neg b_2$ for all presence

conditions and thus eliminates b_2 from H_6. FMLR then repeats the process of forking, shifting, reducing, and merging for the remaining 17 conditionals until a single-headed subparser p completes the array initializer on lines 12–13. That way, FMLR parses 2^{18} distinct configurations with only 2 subparsers!

1.5 Pragmatics

Having covered the overall approach and algorithms, we now turn to the pragmatics of building a real-world tool. SuperC implements the three steps of parsing all of C—lexing, preprocessing, and parsing—in Java. We engineered both preprocessor and parser from scratch, but rely on JFlex [44] to generate the lexer and on Bison [31] to generate the LALR parser tables. Since Bison generates C headers, we wrote a small C program that converts them to Java. As inputs to JFlex and Bison, we reuse Roskind's tokenization rules and grammar for C [55], respectively; we added support for common gcc extensions. To parse conditional expressions, the preprocessor also reuses a C expression grammar distributed with the *Rats!* parser generator [38]. To facilitate future retargeting to other languages, SuperC's preprocessor accesses tokens through an interface that hides source language aspects not relevant to preprocessing. Furthermore, the preprocessor does not pass conditional directives to the parser but rather replaces each directive's tokens with a single special token that encodes the conditional operation and references the conditional expression as a BDD. Finally, the parser is not only configured with the parser tables but also with plug-ins that control AST construction (Section 1.5.1) and context management (Section 1.5.2). To support these plug-ins, each subparser stack element has a field for the current semantic value and each subparser has a field for the current context.

1.5.1 Building Abstract Syntax Trees

To simplify AST construction, SuperC includes an annotation facility that eliminates explicit semantic actions in most cases. Developers simply add special comments next to productions. Our AST tool then extracts these comments and generates the corresponding Java plug-in code, which is invoked when reducing a subparser's stack. By default, SuperC creates an AST node that is an instance of a generic node class, is named after the production, and has the semantic values of all terminals and nonterminals as children. Four annotations override this default. (1) "layout" omits the production's value from the AST. It is used for punctuation. (2) "passthrough" reuses a child's semantic value, if it is the only child

in an alternative. It is particularly useful for expressions, whose productions tend to be deeply nested for precedence (17 levels for C). (3) "list" encodes the semantic values of a recursive production as a linear list. It is necessary because LR grammars typically represent repetitions as left-recursive productions. (4) "action" executes arbitrary Java code instead of automatically generating an AST node.

A fifth annotation, "complete", determines which productions are complete syntactic units. SuperC merges only subparsers with the same, complete nonterminal on top of their stacks; while merging, it combines the subparsers' semantic values with a static choice node. The selection of complete syntactic units requires care. Treating too many productions as complete forces downstream tools to handle static choice nodes in too many different language constructs. Treating too few productions as complete may result in an exponential subparser number in the presence of embedded configurations, e.g., the array initializer in Figure 1.6. SuperC's C grammar tries to strike a balance by treating not only declarations, definitions, statements, and expressions as complete syntactic units, but also members in commonly configured lists, including function parameters, "struct" and "union" members, as well as "struct", "union", and array initializers.

1.5.2 Managing Parser Context

SuperC's context management plug-in enables the recognition of context-sensitive languages, including C, without modifying the FMLR parser. The plug-in has four callbacks: (1) "reclassify" modifies the token follow-set by changing or adding tokens. It is called after computing the follow-set, i.e., line 5 in Algorithm 2. (2) "forkContext" creates a new context and is called during forking. (3) "mayMerge" determines whether two contexts allow merging and is called while merging subparsers. (4) "mergeContexts" actually combines two contexts and is also called while merging.

SuperC's C plug-in works as follows. Its context is a symbol table that tracks which names denote values or types under which presence conditions and in which C language scopes. Productions that declare names and enter/exit C scopes update the symbol table through helper productions that are empty but have semantic actions. "reclassify" checks the name of each identifier, which is the only token generated for names by SuperC's lexer. If the name denotes a type in the current scope, "reclassify" replaces the identifier with a typedef name. If the name is ambiguously defined under the current presence condition, it instead adds the typedef name to the follow-set. This causes the FMLR parser to fork an extra subparser

on such names, even though there is no explicit conditional. "forkContext" duplicates the current symbol table scope. "mayMerge" allows merging only at the same scope nesting level. Finally, "mergeContexts" combines any symbol table scopes not already shared between the two contexts.

1.5.3 Engineering Effort

The SuperC preprocessor and parsing framework consists of about 10k lines of Java code and about 1,000 lines of yacc grammar specification for the C parser, including its semantic actions written in Java. The Java language was chosen to exploit existing parsing libraries, e.g., for tree traversal, AST nodes, etc, provided by the xtc extensible parser generator [38]. We also considered extending the Gnu Compiler Collection, but found it unamenable to convenient modification. Since most of the parser generation work for LR parsers consists of creating parse tables, we decided to use the Bison parser generator for this tasks, and create the fork-merge parser from scratch.

1.6 Evaluation

To evaluate our work, we explore three questions. Section 1.6.1 examines how prevalent preprocessor usage is in real-world code. It measures preprocessor directives and feature interactions in the Linux kernel. Section 1.6.2 examines how effective FMLR is at containing the state explosion caused by static conditionals. It measures the number of subparsers necessary for parsing Linux and also compares to our reimplementation of MAPR. Section 1.6.3 examines how well SuperC performs. It measures the latency for parsing Linux and also compares to TypeChef. We focus on Linux for three reasons: (a) it is large and complex, (b) it has many developers with differing coding styles and skills, and (c) it is subject to staggering performance and variability requirements. However, since the Linux build system does not use the preprocessor for setting architecture-specific header files, we evaluate only the x86 version of the kernel. In summary, our evaluation demonstrates that Linux provides a cornucopia of preprocessor usage, that FMLR requires less than 40 subparsers for Linux whereas MAPR fails on most source files, and that SuperC performs well enough, out-running TypeChef by more than a factor of four and out-scaling it for complex compilation units.

1.6.1 Preprocessor Usage and Interactions

	Total	C Files	Headers
LoC	5,600,227	85%	15%
All Directives	532,713	34%	66%
"#define"	366,424	16%	84%
"#if", "#ifdef", "#ifndef"	38,198	58%	42%
"#include"	86,604	85%	15%

(a) Number of directives compared to lines of code (LoC).

Header Name	C Files That Include Header
include/linux/module.h	3,741 (49%)
include/linux/init.h	2,841 (37%)
include/linux/kernel.h	2,567 (33%)
include/linux/slab.h	1,800 (23%)
include/linux/delay.h	1,505 (20%)

(b) The top five most frequently included headers.

Table 1.2: A developer's view of x86 Linux preprocessor usage.

Table 1.2 provides a *developer's view* of preprocessor usage in the x86 Linux kernel. The data was collected by running "cloc", "grep", and "wc" on individual C and header *files*. Table 1.2a compares the number of preprocessor directives to lines of code (LoC), excluding comments and empty lines. Even this simple analysis demonstrates extensive preprocessor usage: almost 10% of all LoC are preprocessor directives. Yet, when looking at C files, preprocessor usage is not nearly as evident for two reasons. First, macro invocations look like C identifiers and C function calls; they may also be nested in other macros. Consequently, they are not captured by this analysis. Second, C programs usually rely on headers for common definitions, i.e., as a poor man's module system. The data corroborates this. 66% of all directives and 84% of macro definitions are in header files. Furthermore, 15% of include directives are in header

files, resulting in long chains of dependencies. Finally, some headers are directly included in thousands of C files (and preprocessed for each one). Table 1.2b shows the top five most frequently included headers; "module.h" alone is included in nearly half of all C files.

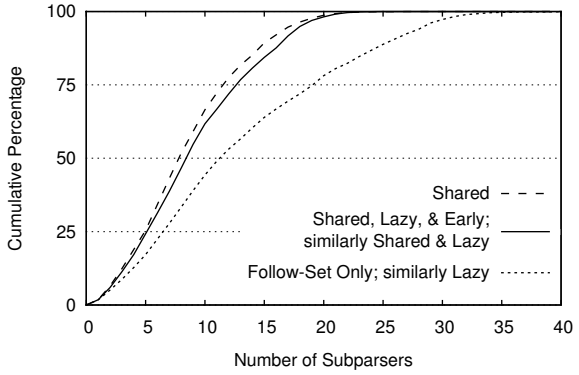
Table 1.3 provides a *tool's view* of preprocessor usage in the x86 Linux kernel. The data was collected by instrumenting SuperC and applying our tool on *compilation units*, i.e., C files plus the closure of included headers. It captures information not available in the simple counts of Table 1.2, including macro invocations. Table 1.3 loosely follows the organization of Table 1.1. Each row shows a preprocessor or C language construct. The first column names the construct, the second column shows its usage, and the third and fourth columns show its interactions. Each entry is the distribution in three percentiles, "50th · 90th · 100th," across compilation units. Table 1.3 confirms that preprocessor usage is extensive. It also confirms that most interactions identified in Section 1.2 occur in real-world C code.

The vast majority of measured preprocessor interactions involve macros. First, almost all macro definitions are contained in static conditionals, i.e., any difference is hidden by rounding to the nearest thousand. This is due to most definitions occurring in header files and most header files, in turn, containing a single static conditional that protects against multiple inclusion. Second, over 60% of macro invocations appear from within other macros; e.g., the median for total macro invocations is 98k, while the median for nested invocations is 64k. This makes it especially difficult to fully analyze macro invocations without running the preprocessor, e.g., by inspecting source code. While not nearly as frequent as interactions involving macros, static conditionals do appear within function-like macro invocations, token-pasting and stringification operators, file includes, as well as conditional expressions. Consequently, a configuration-preserving preprocessor must hoist such conditionals. Similarly, non-boolean expressions do appear in conditionals and the preprocessor must preserve them. However, two exceptions are notable. Computed includes are very rare and ambiguously-defined names do not occur at all, likely because both make it very hard to reason about source code.

1.6.2 Subparser Counts

	Subparsers	
Optimization Level	**99th %**	**Max.**
Shared, Lazy, & Early	21	39
Shared & Lazy	22	39
Shared	21	77
Lazy	32	468
Follow-Set Only	33	468
MAPR & Largest First	>16,000 on 98% of comp. units	
MAPR	>16,000 on 98% of comp. units	

(a) The maximum number across optimizations.



(b) The cumulative distribution across optimizations.

Figure 1.8: Subparser counts per main FMLR loop iteration.

According to Table 1.3, most compilation units contain thousands of static conditionals. This raises the question of whether recognizing C code across conditionals is even feasible. Two factors determine feasibility: (1) the breadth of conditionals, which forces the forking of subparsers, and (2) the incidence of partial C constructs in conditionals, which prevents the merging of subparsers. The number of subparsers per iteration of FMLR's main loop in Alg. 2:3–14 precisely captures the combined effect of these two factors.

Figure 1.8 shows the cumulative distribution of subparser counts per FMLR iteration for the x86 Linux kernel under different optimization levels: 1.8a identifies the maxima and 1.8b characterizes the overall shape. For comparison, the former also includes MAPR. We reimplemented MAPR by modifying SuperC to optionally fork a subparser for every conditional branch instead of using the token follow-set. We also reimplemented MAPR's tie-breaker for the priority queue, which favors the subparser with the larger stack [53]. Figure 1.8 demonstrates that MAPR is intractable for Linux, triggering a kill-switch at 16,000 subparsers for 98% of all compilation units. In contrast, the token follow-set alone makes FMLR feasible for the entire x86 Linux kernel. The lazy shifts, shared reduces, and early reduces optimizations further decrease subparser counts, by up to a factor of 12. They also help keep the AST smaller: fewer forked subparsers means fewer static choice nodes in the tree, and earlier merging means more tree fragments outside static choice nodes, i.e., shared between configurations.

1.6.3 Performance

Both SuperC and TypeChef run on the Java virtual machine, which enables a direct performance comparison. All of SuperC and TypeChef's preprocessor are written in Java, whereas TypeChef's parser is written in Scala. Running either tool on x86 Linux requires some preparation. (1) As discussed in Section 1.2, both tools need to be configured with gcc's built-in macros. SuperC automates this through its build system; TypeChef's distribution includes manually generated files for different compilers and versions. (2) Both tools require a list of C files identifying the kernel's compilation units. We reuse the list of 7,665 C files distributed with TypeChef. Kästner et al. assembled it by analyzing Linux' configuration database [42]. (3) SuperC needs to be configured with four definitions of non-boolean macros. We discovered the four macros by comparing the result of running gcc's preprocessor, i.e., "gcc" "-E", under the "allyesconfig" configuration on the 7,665 C files with the result of running it on the output of SuperC's

configuration-preserving preprocessor for the same files. With those four definitions in place, the results are identical modulo whitespace. This comparison also provides us with high assurance that SuperC's preprocessor is correct. (SuperC's parser is less rigorously validated with hand-written regression tests.) (4) TypeChef needs to be configured with over 300 additional macro definitions. It also treats macros that are not explicitly marked as configuration variables, i.e., have the "CONFIG_" prefix, as undefined instead of free.

We refer to the experimental setup including only the first three steps as the *unconstrained* kernel and the setup including all four steps as the *constrained* kernel. As of 2/18/12, TypeChef runs only on the constrained kernel, and only on version 2.6.33.3. To ensure that results are comparable, the examples and experiments in this paper also draw on version 2.6.33.3 of Linux. At the same time, SuperC runs on both constrained and unconstrained kernels. In fact, the data presented in Table 1.3 for preprocessor usage and in Figure 1.8 for subparser counts was collected by running SuperC on the unconstrained kernel. By comparison, the constrained kernel has less variability: its 99th and 100th percentile subparser counts are 12 and 32, as opposed to 21 and 39 for the unconstrained kernel. SuperC also runs on other versions of Linux; we validated our tool on the latest stable version, 3.2.9.

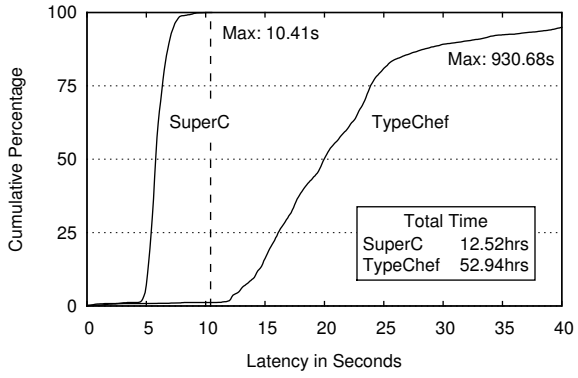


Figure 1.9: SuperC and TypeChef latency per compilation unit.

Figure 1.9 shows the cumulative latency distribution across compilation units of the constrained kernel when running SuperC or TypeChef on an off-the-shelf PC. For each tool, it also identifies the maximum latency for a compilation unit and the total latency for the kernel. The latter number should be treated as a convenient summary, but no more: workload and tools easily parallelize across cores and machines.

When considering the 50th and 80th percentiles, both tools perform reasonably well. While SuperC is between 3.4 to 3.8 times faster than TypeChef, both curves show a mostly linear increase, which is consistent with a normal distribution. However, the "knee" in TypeChef's curve at about 25 seconds and the subsequent long tail, reaching over 15 minutes, indicates a serious scalability bottleneck. The likely cause is the conversion of complex presence conditions into conjunctive normal form [41]; this representation is required by TypeChef's SAT solver, which TypeChef uses instead of BDDs.

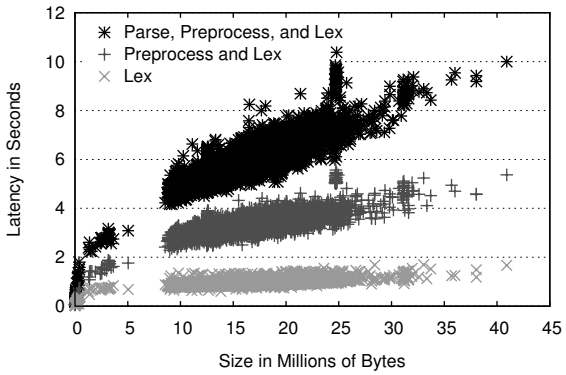


Figure 1.10: SuperC latency by compilation unit size.

Figure 1.10 plots a breakdown of SuperC latency. It demonstrates that SuperC's performance scales roughly linearly with compilation unit size. Lexing, preprocessing, and parsing each scale roughly linearly as well, with most of the total latency split between preprocessing and parsing. The spike at about 25 MB is due to fs/xfs/ containing code with a high density of macro invocations. To provide a performance baseline, we measured the cumulative latency distribution for gcc lexing, preprocessing, and parsing the 7,665 compilation units under "allyesconfig". We rely on gcc's "-ftime-report" command line option for the timing data. The 50th, 90th, and 100th percentiles are 0.18, 0.24, and 0.87 seconds, i.e., a factor of 12 to 32 speedup compared to SuperC. It reflects that gcc does not have to preserve static conditionals and that gcc's C implementation has been carefully tuned for many years.

1.7 Related Work

Our work joins a good many attempts at solving the problem of parsing C with arbitrary preprocessor usage [5, 7, 9, 10, 29, 34, 41, 42, 48, 51, 53, 59, 66]. Out of these efforts, only MAPR [53] and Type-

Chef [41, 42] come close to solving the problem. Since we already provided a detailed comparison to MAPR and TypeChef in Sections 1.1, 1.2 and 1.6, we only discuss the other efforts here.

Previous, and incomplete, work on recognizing all of C can be classified into three categories. First are tools, such as Xrefactory [66], that process source code one configuration at a time, after full preprocessing. This approach is also taken by Apple's Xcode IDE [17]. However, due to the exponential explosion of the configuration space, this is only practical for small source files with little variability. Second are tools, such as CRefactory [34], that employ a fixed but incomplete algorithm. This approach is also taken by the Eclipse CDT IDE [37]. It is good enough—as long as source code does not contain idioms that break the algorithm, which is a big if for complex programs such as Linux. Third are tools, such as Yacfe [51], that provide a plug-in architecture for heuristically recognizing additional idioms. However, this approach creates an arms race between tool builders and program developers, who need to push both preprocessor and C itself to wring the last bit of flexibility and performance out of their code—as amply demonstrated by Ernst et al. [28], Tartler et al. [61], and this paper's Section 1.6.

Considering parsing more generally, our work is comparable to efforts that build on the basic parsing formalisms, i.e., LR [45], LL [54], and PEG [14, 30], and seek to improve expressiveness and/or performance. Notably, Elkhound [49] explores how to improve the performance of generalized LR (GLR) parsers by falling back on LALR for unambiguous productions. Both SDF2 [19, 65] and *Rats!* [38] explore how to make grammars modular by building on formalisms that are closed under composition, GLR and PEG, respectively. *Rats!* also explores how to speed up PEG implementations, which, by default, memoize intermediate results to support arbitrary back-tracking with linear performance. Finally, ANTLR [52] explores how to provide most of the expressivity of GLR and PEG, but with better performance by supporting variable look-ahead for LL parsing.

At a finer level of detail, Fork-Merge LR parsing relies on a DAG of parser stacks, just like Elkhound, but for a substantially different reason. Elkhound forks its internal state to accept ambiguous *grammars*, while SuperC forks its internal state to accept ambiguous *inputs*. Next, like several other parser generators, SuperC relies on annotations in the grammar to control AST building. For instance, ANTLR, JavaCC/JJTree [39], *Rats!*, SableCC [32], and SDF2 provide comparable facilities. Finally, many parsers for C employ an ad-hoc technique for disambiguating typedef names from other names, termed the "lexer hack" by Roskind [55]. Instead, SuperC relies on a more general plug-in facility for context management. *Rats!* has a comparable facility, though details differ significantly due to the underlying parsing

formalisms, i.e., LR for SuperC and PEG for *Rats!*.

1.8 Conclusion

This chapter explores how to perform syntactic analysis of C code while preserving its variability, i.e., static conditionals. First, we identify all challenges posed by interactions between C preprocessor and language proper. Our anecdotal and empirical evidence from the x86 Linux kernel demonstrates that meeting these challenges is critical for processing real-world C programs. Second, we present novel algorithms for configuration-preserving preprocessing and parsing. Hoisting makes it possible to preprocess source code while preserving static conditionals. The token follow-set as well as early reduces, lazy shifts, and shared reduces make it possible to parse the result with very few LR subparsers and to generate a well-formed AST. Third, we discuss the pragmatics of building a real-world tool, SuperC, and demonstrate its effectiveness on Linux. For future work, we will extend SuperC with support for automated refactorings and explore configuration-preserving semantic analysis. We expect that the latter, much like our configuration-preserving syntactic analysis, will require incorporating presence conditions into all functionality, including by maintaining multiply-defined symbols. In summary, forty years after C's invention, we finally lay the foundation for efficiently processing *all of C*.

Language Construct	Total	Interaction with Conditionals	
Macro Definitions	34k · 45k · 122k	Contained in	34k · 45k · 122k
Macro Invocations	98k · 140k · 381k	Trimmed	16k · 21k · 70k
		Hoisted	154 · 292 · 876
Token-Pasting	4k · 6k · 22k	Hoisted	0 · 0 · 180
Stringification	6k · 8k · 23k	Hoisted	361 · 589 · 6,082
File Includes	1,608 · 2,160 · 5,939	Hoisted	33 · 55 · 165
Static Conditionals	8k · 10k · 29k	Hoisted	331 · 437 · 1,258
		Max. depth	28 · 33 · 40
Error Directives	42 · 57 · 168		
C Declarations & Statements	34k · 49k · 127k	Containing	722 · 896 · 2,746
Typedef Names	748 · 1,028 · 2,554	Ambiguously defined names	0 · 0 · 0

(a) Total interactions and interactions with conditionals

Table 1.3: A tool's view of x86 Linux preprocessor usage. Entries show percentiles across compilation units: 50th · 90th · 100th.

Language Construct	Total	Other Interactions	
Macro Definitions	34k · 45k · 122k	Redefinitions	23k · 33k · 111k
Macro Invocations	98k · 140k · 381k	Nested invocations	64k · 97k · 258k
		Built-in macros	135
Token-Pasting	4k · 6k · 22k		
Stringification	6k · 8k · 23k		
File Includes	1,608 · 2,160 · 5,939	Computed includes	34 · 56 · 168
		Reincluded headers	1,185 · 1,743 · 5,488
Static Conditionals	8k · 10k · 29k	With non-boolean expressions	509 · 713 · 1,975
Error Directives	42 · 57 · 168		
C Declarations & Statements	34k · 49k · 127k		
Typedef Names	748 · 1,028 · 2,554		

(b) Other interactions

Table 1.3: A tool's view of x86 Linux preprocessor usage. Entries show percentiles across compilation units: 50th · 90th · 100th.

Chapter 2

Building with Variability

2.1 Introduction

As software systems become larger, automated software engineering tools such as source code browsers, bug finders, and automated refactorings, become more important. Larger systems are more vulnerable to bugs, and modifications to the codebase are more difficult to verify by hand due to the larger number of interactions between features of the system. C is the language of choice for many common large-scale software systems, including the Linux kernel, the Apache web server, and the GNU compiler collection, all of which are used in critical computing systems. One facet of large-scale software development is variability management, with which software systems are tailored to a specific use by enabling features at build-time. With such variability, a codebase encompasses a family of customized software product lines, which share portions of the source code and features. Variability amplifies the difficulty of creating and using automated software tools, because such tools need to be aware of the variability in order to operate on all product lines in the software family. Worse still, variability introduces new classes of bugs. Abal et al found bugs resulting from the interactions and dependencies between features of the Linux kernel, but lacking automated tools, found them by manually examining patches send to the Linux kernel mailing list [4].

New languages and formalisms for describing variability promise safety and the easier application of software engineering tools [48, 43, 62], but until such variability tools are widespread, an abundance of

critical C software remains that uses ad-hoc techniques for variability. In our previous work, SuperC, we preprocess and parse all variations of C source files in the Linux kernel, which uses the preprocessor to implement variability within source files [35]. While this provides the foundation for variability-aware tool implementation for individual source files, large C programs are comprised of potentially thousands of compilation units, i.e., C files compiled separately and linked to form the final program. The Linux kernel v3.19, for instance, contains over 20,000 compilation units, but only a subset of these compilation units are used for a single software product line, depending on the selected features.

Being able to extract all compilation units and their variability information is crucial for software engineering tools. For instance, C function calls can cross compilation unit boundaries, only being referenced by an `extern` declaration. Without knowing the complete set of compilation units that may be linked together, static analyses cannot find all callees. Bug-checking in particular is limited without variability-awareness. Chou et al shows that static checkers find bugs in the Linux kernel [23], but they only operate on one software product line. Families of product lines harbor untestable bugs, since it is not feasible to check every possible combination of features separately. Variability-awareness enables tools to operate across sofware product lines. For instance, knowing which compilation units are linked under which combinations of features can help root out linker errors without having to build and link every possible variation of the software. Additionally, previous work on translating Linux's extensive C preprocessor use to a safer alternative, such as aspects [5] or to the ASTEC preprocessor [48], depends on a complete view of the kernel source.

This chapter focuses on the Linux build system due to its size, complexity, and prevalence. Several software projects also use Linux's build system tools to manage variability (Table 2.1), including the BusyBox toolkit [21] and the uClibc library [64], and Kmax's approach applies generally to any build system language that uses conditionals to implement variability. The Linux build system is a relevant target for Kmax, because Linux is a frequent object of study for researchers, yet it is difficult to extract variability information from its build system. For instance, Liebig et al computes statistics on variability metrics in the Linux kernel [47]. But that study along with others, including including this author's previous work, use an incomplete set of compilation units to experiment on the 2.6.33.3 version Linux [42, 35]. All three report using 7,665 or fewer units while there are 9,344, which is off by more than 15%. At best, this leads to incomplete data. At worst, static analysis tools get a incomplete view of the kernel, missing critical problems in the source code. These incomplete studies can be traced to back to a single

Software	Description	Kconfig	Kbuild
coreboot	Open-source boot firmware	•	•
Busybox	complete embedded system including linux, system tools, shell, etc	•	•
uClibc	C library for embedded systems (complements Busybox)	•	•
uClibc++	C++ library for embedded systems	•	•
buildroot	automates embedded system build process (complements Busybox)	•	
crosstool-ng	builder for cross-compiling toolchains	•	
OpenWRT	specialized variant of Linux for installing into routers	•	
PTXdist	Builds a complete embedded system	•	
kconfig-frontends	standalone version of Linux's kconfig parser and front-ends	•	
OpenBricks	builds embedded systems: toolchain, board support, and installation	•	
NuttX	real-time OS	•	
EmbToolkit	configure and build toolchain for embedded systems	•	•

Table 2.1: Software that uses Kconfig, Kbuild, or both.

tool, KBuildMiner.

Using a fuzzy parser for Linux Makefiles, KBuildMiner collects compilation units by looking for usage patterns [11]. This approach misses the mark, because some compilation unit names are defined by concatenation and function calls, which requires evaluating the `make` language. Furthermore, some Kbuild files need to be hand-modified to fit the syntax recognized by the parser. Aware of the limitations of KBuildMiner, Dietrich et al sought to improve the state of build system analysis with GOLEM. GOLEM enables one or more features at a time and runs `make` to see which compilation units get activated [25]. While this semi-brute-force approach successfully avoids having to try all combinations of features, its heuristic approach falls short.

This paper introduces Kmax. Kmax extracts all compilation units and their variability information without using heuristics. At its core is its variability-preserving `make` evaluator, that records all possible compilation units that comprise any software product line, and the feature selections that lead to these compilation units. In addition to evaluating most of the `make` language, it employs three key techniques: (1) it maintains a conditional symbol table with all possible variable definitions, (2) it evaluates all

branches of conditionals blocks, and (3) it *hoists* conditionals around statements that contain conditionals. During processing, Kmax's `make` evaluator tracks the combinations of features as a boolean expression, called a *presence condition*. By tracking the presence condition during evaluation, it can discover the combination of features that enables each compilation unit.

The contributions of this paper are as follows:

1. Algorithms to find selectable features and evaluate a subset of the `make` language across software product lines,

2. A new tool, Kmax, that implements the algorithms to extract compilation units and their presence conditions from the Linux build system, and

3. Empirical evaluation of Kmax's correctness and performance with a comparison to previous work.

Kmax is available at `http://cs.nyu.edu/~gazzillo/kmax.html`.

2.2 Problem and Solution Approach

Kmax's challenges stem from both the difficulty of evaluating the `make` language in the presence of variability and the peculiarities of the Linux source tree's organization. Its build system uses two specification languages, Kconfig to define features and their constraints and Kbuild, a `make`-based language that describes how features control the build process. To build one product line from the Linux codebase, the user first selects the architecture. Then the user selects the features to include in the kernel, such as drivers and file systems. Once configured, the build process is typical of C programs, using `make` to run the Kbuild files, compiling and linking the compilation units according to the selected features. The process Kmax uses to extract the compilation units and their variability information from the build system mirrors the build process. Given an architecture, Kmax first processes the Kconfig files to find the domain of features. Then, using its variability-preserving `make` evaluator on the Kbuild Makefiles, it extracts the compilation units while recording their presence conditions. The challenges to this process include handling architecture-specific source code, finding selectable features, the particulars of Kbuild, and evaluating `make` across software product lines. These challenges and Kmax's solution approach are detailed below.

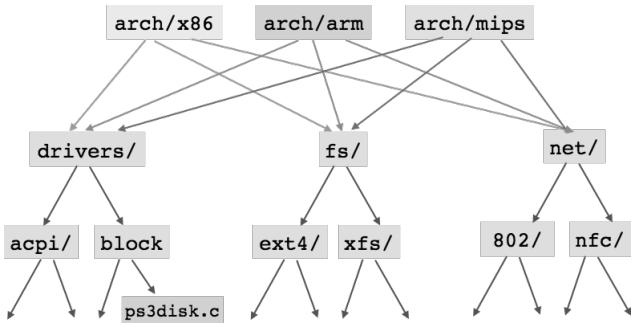


Figure 2.1: Hierarchy of source code in the Linux kernel codebase. Each architecture directory is a separate root of the source tree and includes the rest of the common codebase. Some compilation units appear in the common codebase, but can only be enabled when building for one architecture, e.g., `ps3disk.c` can only be enabled in `arch/arm`

2.2.1 Architecture-Specific Source Code

The Linux kernel source code is hierarchical. Top-level directories define major subsystems, such as `net/` for networking and `drivers/`, and nest related code in subdirectories, for example, `net/ethernet` and `drivers/video`. While the codebase contains source code that is mostly shared by all software product lines, each architecture serves as the root of its own hierarchy. Figure 2.1 illustrates this with a forest. At the roots of the trees are the architecture-specific source code directories. These directories roughly form the hardware abstract layer (HAL), defining macros, functions, types, and include paths that the rest of the codebase uses. Beneath the HAL are the top-level directories, to which each architecture points to form the rest of its hiearchy. There are two consequences to this structure. Firstly, static analyses only make sense only after a HAL is selected and should operate on one architecture at at time. Secondly, not all compilation units are accessible to each architecture's product lines. As Figure 2.1 also shows, finding these compilation units is not straightforward. The `drivers/block/ps3disk.c` compilation unit is part of every architecture's hierarchy. But because of the constraints on features, defined in Kconfig files, only software product lines built for the `arm` architecture can ever enable this compilation unit. Kmax first employs its Selectable algorithm to find architecture-specific features. Then Kmax only allows selectable features to be enabled when extracting compilation units from Kbuild. Table 2.2, generated by Kmax, illustrates how much architectures share with each other. While most architecture-specific compilation

Metric	Count
Total compilation units	21,158
Shared compilation units	13,881
Architecture-specific units in `arch/` directories	5,973
Architecture-specific units in common directories	1,304
Total configuration variables	14,636
Shared	9,658
Architecture-specific	4,978
Per-architecture compilation units	
Minimum	13,906
Maximum	15,976
Per-architecture configuration variables	
Minimum	9,684
Maximum	11,232

Table 2.2: Linux v3.19 build system metrics broken out by architecture-sharing.

units live in the `arch/` directory, more than a thousand are in the common source code directories. As for features, nearly a third are architecture specific.

2.2.2 Finding Selectable Features

Kconfig files use a domain-specific specification language to define features and their constraints. Figure 2.2 shows several representative examples from Linux. (All examples in this paper come from Linux v3.19). Example (a) defines the `USB` feature that enables Universal Serial Bus (USB) support. Line 1 is the variable declaration, while line 2 gives `USB` its type, `tristate`. Tristate variables can be set to one of three values, `y` for inclusion the compiled kernel, `m` for inclusion as a loadable kernel module, and an empty string for exclusion from the kernel. Other types include boolean, which is tristate without the `m`, string, and number. The latter two take constants of their respective types, and they can be used in boolean expressions with relational operators. The text after `tristate` on line 2 is displayed to the user during interactive feature selection.

```
1 config USB
2         tristate "Support for Host-side USB"
3         depends on USB_ARCH_HAS_HCD
4         select NLS # for UTF-8 strings
```

(a) A feature definition. From drivers/usb/Kconfig.

```
1 if USB
2 source "drivers/usb/storage/Kconfig"
3 endif
```

(b) Kconfig's if and source commands. From drivers/usb/Kconfig. Edited to show one out of nine includes.

```
1 config BLK_DEV_IDE_ICSIDE
2         tristate "ICS IDE interface support"
3         depends on ARM && ARCH_ACORN
```

(c) A feature unselectable in most architectures. From drivers/ide/Kconfig.

Figure 2.2: Examples of Kconfig from Linux v3.19.

Kconfig provides three ways to specify constraints between features. The `depends on` keyword on line 3 of example (a) creates a *direct dependency* on `USB_ARCH_HAS_HCD`. USB support can only be enabled if `USB_ARCH_HAS_HCD` is also enabled. The dependency can be any boolean expression of features. Another way to make a dependency is with the `select` keyword, as shown on line 4. This *reverse dependency* forces `NLS` to be enabled when `USB` is enabled, regardless of `NLS`'s other dependencies. Example (b) shows the last way to create a dependency, with an `if`/`endif` block. Every feature defined in the block on lines 1–3 gets a direct dependency on `USB`. The `source` statement on line 2 includes another Kconfig file, which is used to form the hierarchy of Kconfig files.

Example (c) shows an architecture-specific variable defined in the shared part of the Kconfig hierarchy. BLK_DEV_IDE_ICSIDE is defined for all architectures, but can only be enabled for ARM because of its direct dependence on the ARM feature. For example, x86's Kconfig files never define ARM, making it an *unreachable* feature. In contrast, BLK_DEV_IDE_ICSIDE is reachable, but its dependencies prevent it from ever being enabled when building for x86, making it *unselectable*. A feature is selectable only if two conditions hold: (1) it is reachable and (2) any dependencies are also selectable. Kmax uses Linux's own parser for the Kconfig files, which yields a in-memory representation of the features and their constraints as boolean expressions. The Selec-table algorithm finds the selectable features for an architecture. As Table 2.2 shows, out of 14,636 features in Linux v3.19, only between 9,684 and 11,232 are selectable for any given architecture.

```
1 obj-$(CONFIG_USB_UAS) += uas.o
2 obj-$(CONFIG_USB_STORAGE) += usb-storage.o
3 usb-storage-y := scsiglue.o protocol.o transport.o usb.o
4
5 obj-$(CONFIG_USB_STORAGE) += storage/
```

Figure 2.3: Snippets of Kbuild from Linux v3.19. Lines 1–3 are from drivers/usb/storage/Makefile, line 5 from drivers/usb/Makefile.

2.2.3 The Particulars of Kbuild

Compilation units are defined in Kbuild using Makefile syntax. Their names are added to Kbuild's reserved variables obj-y for built-ins and obj-m for dynamically-loadable modules. The build system later uses these lists to compile and link the kernel binaries. Since enabled tristate features are set to y or m, Kbuild files make use of a common pattern where the obj- prefix is concatenated with the value of the feature. Figure 2.3 is an example of this pattern. On line 1, uas.c is only compiled if the USB_UAS feature is enabled with y or m. In Makefile syntax, $(CONFIG_USB_UAS) expands to the value of the feature, which is given the CONFIG_ prefix as a de facto namespace. Adjacent strings get concatenated, requiring no special operator. When USB_UAS is set to y, expansion and concatenation yield the string obj-y, while the += operator appends uas.o to the existing definition of the obj-y variable, adding it

to the list of built-in compilation units. When disabled, `USB_UAS` expands to the empty string, adding the compilation unit to the variable `obj-`, which is ignored by Kbuild. This pattern makes clear which feature controls a compilation unit, only compiling it when the feature is enabled.

Line 2 is an example of a *composite* compilation unit. If a compilation unit, such as `usb-storage.o`, has no corresponding C file, the Kbuild evaluator looks for a variable with the name of the compilation unit plus a `-y` or `-objs` suffix. In this case, `usb-storage-y` on line 3 defines the constituent compilation units, which can themselves be composite. As with `obj-`, a composite's variable name may be concatenated with a feature to conditionally include compilation units.

Line 5 adds a subdirectory name to `obj-y` or `obj-m` instead of a compilation unit. The Kbuild evaluator enters these subdirectories to find more compilation units, which is how the Kbuild hierarchy is formed. Each subdirectory's compilation units are linked into `builtin.o` or `.ko` files for modules. Once finished with the subdirectory, Kbuild replaces `storage/` with `storage/builtin.o` for linking into the parent directory's own `builtin.o`. The `subdir-y` variable may also be used to explicitly add subdirectory names for Kbuild to traverse.

```
1 ifdef CONFIG_NO_BOOTMEM
2         obj-y += nobootmem.o
3 else
4         obj-y += bootmem.o
5 endif
```

(a) Makefile conditionals create mutually-exclusive compilation units. From mm/Makefile.

```
1 obj-$(CONFIG_SMP) += smp.o
2
3 // after hoisting
4 ifeq (CONFIG_SMP, y)
5 obj-y += smp.o
6 endif
7 ifndef CONFIG_SMP
8 obj- += smp.o
9 endif
```

(b) Using Kbuild's reserved obj-y variable with configuration variable expansion. From kernel/Makefile.

Figure 2.4: Examples from Linux v3.19 of the challenges of evaluating Kbuild.

2.2.4 Challenges to Evaluating make

Even though they frequently use common patterns, Kbuild files have the full power of the `make` language features available to use. Figure 2.4 contains examples that illustrate Kbuild usage. The first two examples show how Kbuild files make certains combinations of features mutually exclusive, the next two show variable expansion and functions used while defining compilation units, and the last is an example of an architecture-specific compilation unit.

Example (a) is a tests for the feature named `CONFIG_NO_BOOTMEM` and compiles one of `nobootmem.o` or `bootmem.o`, but never both. Kmax first evaluates the conditional expression on line 1 to find the

```
1 cacheops-$(CONFIG_CPU_SH2) := cache-sh2.o
2 cacheops-$(CONFIG_CPU_SH2A) := cache-sh2a.o
3 cacheops-$(CONFIG_CPU_SH3) := cache-sh3.o
4 // three more reassignments
5 obj-y += $(cacheops-y)
```

(c) Variable assignment creates mutually exclusive compilation units. From arch/sh/mm/Makefile.

```
1 // From arch/x86/Makefile
2 ifeq ($(CONFIG_X86_32),y)
3        BITS := 32
4 else
5        BITS := 64
6 endif
7
8 obj-$(CONFIG_X86_LOCAL_APIC) += probe_$(BITS).o
```

(d) Compilation unit names can be generated from Makefile variables. From arch/x86/kernel/apic/Makefile.

Figure 2.4: Examples from Linux v3.19 of the challenges of evaluating Kbuild.

conditions needed to enter the if-branch. It then evaluates the statements in both branches on lines 2 and 4, storing both definitions of `obj-y` in its conditional symbol table. Example (b) shows a feature `SMP` concatenated with the `obj-` to conditionally compile `smp.o` on line 1. When features or other Makefile variables that have multiple definitions are expanded, it is an implicit conditional, since each definition has a condition, called a presence condition, under which it is expanded. Kmax handles multiply-defined variables by expanding all their definitions to a conditional and hoisting it around the statement. Lines 3–9 show the conceptual result of hoisting, although Kmax does not explicitly create a conditional block.

Example (c) shows how variable reassignment can implicitly create mutually exclusive feature combinations. Because `cacheops-y` is reassigned repeatedly on lines 1–3, only one of the named compilation units can appear in any single software product line. Kmax creates an entry for each possible definition

```
1 // From net/ipv6/Makefile.
2 obj-$(subst m,y,$(CONFIG_IPV6)) += inet6_hashtables.o
3
4 // From arch/s390/Makefile.
5 head-y += arch/s390/kernel/$(if $(CONFIG_64BIT),head64.o,head31.o)
6
7 // From arch/arm/Makefile.
8 machdirs := $(patsubst %,arch/arm/mach-%/,$(machine-y))
```

(e) Makefiles can use functions when expanding configuration variables.

```
1 obj-$(CONFIG_BLK_DEV_IDE_ICSIDE) += icside.o
```

(f) Some compilation units depend on architecture-specific configuration variables. From drivers/ide/Makefile.

Figure 2.4: Examples from Linux v3.19 of the challenges of evaluating Kbuild.

of `cacheops-y` along with a boolean expression representing the conditions under which the definition is possible.

Example (d) shows a case where a compilation unit's name is constructed by concatenation with the value of a variable. `BITS` is a global variable, defined in a top-level Makefile, that expands to either 32 or 64 depending on a feature as shown on lines 2–6. On line 8, Kmax expands both definitions a conditional, hoists the implicit conditional around the entire assignment statement, and as with the the conditional in example (a) stores both compilation unit names.

Example (e) shows function calls used while defining compilation units. Line 2 uses the substitute function to force the compilation unit to be built-in, instead of a module. Line 5 uses the conditional function to decide between compilation units. And Line 8 uses pattern substitution to generate a list of directories from arm machine names. These cases in particular make it difficult to collect compilation units without doing some evaluation. As with variable expansion, any feautres used in function arguments are expanded to a conditional and hoisted around the function calls. After hoisting, the functions can be evaluated normally under each resulting presence condition.

Example (f) shows the architecture-specific feature from Figure 2.2c being used to control a compilation unit. Because this feature is only available when compiling for ARM, Kmax takes a set of selectable configuration variables for each architecture before evaluating Kbuild Makefiles.

2.3 Algorithms

The core of Kmax's solution comprises the Selectable algorithm that finds features available to a given architecture and a `make` language evaluator the collects all compilation units, recording their enabling features as boolean expressions. This section describes these algorithms in detail.

2.3.1 Selectable Features

Kmax finds selectable features by excluding those that depend only on other unreachable or unselectable features. Kmax employs Linux's own Kconfig parser, which produces a in-memory list of features and symbolic boolean expressions for their dependencies. It then uses the Selectable algorithm on each one, which returns true if selectable.

Algorithm 4 defines Selectable, which takes a feature name v and the list of features C produced by the Kconfig parser. Lines 13–15 check whether v is reachable by checking it against the list of parsed features. Unreachable features are never selectable. Lines 16–18 look at features with no dependencies. Such variables are always selectable, since there are no other features constraining them. Lines 19–21 check all other features, i.e., those that are reachable, but have dependencies. Evaluate determines whether such a variable is selectable by examining its dependencies. Since either a direct or reverse dependency can allow a variable to be enabled, their expressions are first ORed. Lines 2–12 define Evaluate, which computes the given boolean expression e, recursively evaluating its subexpressions and any other features used. Lines 3–4 handle an AND operator by ANDing the results from checking the subexpressions for selectability. Only when both subexpressions allow selection will the expression be selectable. Lines 5–6 handle an OR operator by ORing the subexpression. Only one subexpression needs to be true for selectability. Lines 7–8 recursively check features used in the expression by recursively calling Selectable. This call is optimized by memoizing the return value for features previously evaluated.

Evaluating selectability mirrors evaluating boolean expressions, except for negation. To see why this is, take a feature `VAR` that depends on ¬`DEP`. If `DEP` is unselectable, then `VAR` is selectable, because of

Algorithm 4 Find selectable configuration variables.

```
 1: procedure Selectable(v, C)
 2:     procedure Evaluate(e)
 3:         if e = l ∧ r then
 4:             return Evaluate(l) ∧ Evaluate(r)
 5:         else if e = l ∨ r then
 6:             return Evaluate(l) ∨ Evaluate(r)
 7:         else if e = w, for config variable w then
 8:             return Selectable(w, C)
 9:         else if e = ¬w, for config variable w then
10:             return true
11:         end if
12:     end procedure
13:     if v ∉ C then
14:         ▷ Unreachable variables never selectable
15:         return false
16:     else if v ∈ C and v.direct = v.reverse = ∅ then
17:         ▷ Non-dependent variables always selectable
18:         return true
19:     else ▷ v is reachable and has dependencies.
20:         ▷ Check v's dependencies.
21:         return Evaluate(v.direct ∨ v.reverse)
22:     end if
23: end procedure
```

the negation. If instead `DEP` is selectable, using boolean negation would force the `VAR` to be unselectable. This would be incorrect, because `VAR` can still be enabled when `DEP` is disabled. Thus negation gives no information about selectability, so lines 9–10 always return true so as not to incorrectly limit selectability. The Selectable algorithm has a complementary Unselectable algorithm that returns true when a feature cannot be selected. This algorithm differs only in the grayed sections of Algorithm 4, namely the boolean operators and the true and false constants. Swapping AND with OR and true with false yields the complementary algorithm. The resulting sets of selectable and unselectable features are also complementary.

2.3.2 Evaluating the make Language

The selectable features are fed to Kmax's `make` evaluator, which evaluates Kbuild files across all software product lines. To acheive this, Kmax uses a conditional symbol table that holds all definitions of the `make` variables it encounters, enters and evaluates all branches of conditionals, and hoists conditionals

that appear within statements to evaluate all possible complete statements. Because the Linux build system keeps compilation units in the `obj-y` and `obj-m`, Kmax finds all compilation units by inspecting the contents of the conditional symbol table. To record the features that control each compilation unit, Kmax's evaluator tracks the presence condition, a boolean expression of features, at each point in the Kbuild file, saving the presence condition when encountering a new compilation unit. The following describes the conditional symbol table, hoisting, and the evaluation algorithm.

A conditional symbol table maps a variable name to a list of (d, c) tuples, where d is a definition and c is a presence condition representing a boolean expression of features. The conditional symbol table is initialized to contain all selectable tristate and boolean features. A tristate feature is represented as a Makefile variable v and is initialized to

$$T(v) \leftarrow \{\,(\text{"y"}, v = \text{"y"}), (\text{"m"}, v = \text{"m"}), (\text{""}, \neg\,\text{defined}(v))\,\}$$

These initial conditions are tautologies, i.e., v expands to y when v = "y" is true and it expands to nothing when v is undefined. Once expanded, however, these initial conditions ensure variability information is carried along in presence conditions in subsequent evaluation. For instance, the following variable definition involves the expansion of a second variable in order to determine the name of variable being assigned:

```
obj-$(CONFIG_USB_UAS) += uas.o
```

Because `CONFIG_USB_UAS` has multiple definitions, the evaluator expands all possible definitions, using a conditional block to preserve the presence conditions of the expanded variable definitions:

```
obj-
ifeq (CONFIG_USB_UAS, y)
  y
else
  # empty
endif
+= uas.o
```

But such a statement with an embedded conditional is not readily able to be evaluated. To handle conditionals within statements, Kmax hoists the conditional around the entired statements, yielding two com-

plete variable assignments. Hoisting takes every possible combination of conditionals that appears within a statement and makes each complete statement explicit:

```
ifeq (CONFIG_USB_UAS, y)
  obj-y += uas.o
else
  obj- += uas.o
endif
```

Hoisting leaves a conditional block surrounding two assignment statements. Kmax's evaluator handles conditionals by entering and evaluating the contents of each branch, while tracking the presence condition that controls each statement. After evaluating the above example, the symbol table gets four new entries, two for `obj-y` and two for `obj-`, since both variable names are possible and each has two possible definitions:

$$
\begin{aligned}
T(\text{“}\texttt{obj-y}\text{”}) \leftarrow \{ &(\texttt{"uas.o"}, \text{CONFIG\_USB\_UAS=y}),\\
&(\texttt{""}, \quad \neg\,\text{defined(CONFIG\_USB\_UAS)}\,\}\\
T(\text{“}\texttt{obj-}\text{”}) \leftarrow \{ &(\texttt{""}, \quad \text{CONFIG\_USB\_UAS=y}),\\
&(\texttt{"uas.o"}, \neg\,\text{defined(CONFIG\_USB\_UAS)}\,\}
\end{aligned}
$$

Note that the symbol table's entries record not only contents of `obj-y`, but also the features that lead to it.

Algorithm 5 Evaluate the statements of a Makefile.

```
1:  procedure Statements(S, p, T)
2:      for s ∈ S do
3:          if s is a conditional (e, S_if, S_else) then
4:              ▷ Compute all possible ways to enter the if-branch.
5:              c_if ← ⋁ e′ ∧ c for (e′, c) ∈ Expand(e, p, T)
6:              Statements(S_if, p ∧ c_if, T)
7:              Statements(S_else, p ∧ ¬c_if, T)
8:          else if s is a variable assignment (e_v, e_d) then
9:              V ←Expand(e_v, p, T)
10:             D ←Expand(e_d, p, T)
11:             for (v, c_v) ∈ V do
12:                 for (d, c_d) ∈ D do
13:                     ▷ Add each possible definition to T.
14:                     T(v) ← T(v) ∪ {(d, p ∧ c_v ∧ c_d)}
15:                 end for
16:             end for
17:         else if s is an include of e then
18:             I ←Expand(e, p, T)
19:             for (i, c) ∈ I do
20:                 S_i ← parsed statements from file i
21:                 Statements(S_i, p ∧ c, T)
22:             end for
23:         end if
24:     end for
25: end procedure
```

Algorithm 6 Expand Makefile expressions, hoisting conditionals.

```
1:  procedure Expand(E, p, T)
2:      ▷ Initialize result with empty string for all presence conditions.
3:      R ← {("", true)}
4:      for subexpression e ∈ E do
5:          if e is variable expansion of v then
6:              ▷ Get all definitions, recursively expanding them.
7:              R ← R × {Expand(d_i, c_i ∧ p, T) | (d_i, c_i) ∈ T(v)}
8:          else if e is function f with args (a_1, a_2, ...) then
9:              ▷ Expand all function arguments.
10:             A_n = Expand(a_n, p, T) for all a_n ∈ (a_1, a_2, ...)
11:             ▷ Execute function for all argument combinations.
12:             R ← R × {f(a_1i, a_2j, ...) | a_1i ∈ A_1, a_2i ∈ A_2, ...}
13:         else ▷ e is a string
14:             ▷ Append e to every expanded subexpression.
15:             R ← {(re, c) | (r, c) ∈ R}
16:         end if
17:     end for
18:     return R
19: end procedure
```

Algorithm 5 is the pseudo-code for Kmax's evaluator. Statements takes a list of parsed statements S, a presence condition p, and a conditional symbol table T. It supports three kinds of statements: conditionals, variable assignment, and includes. Lines 3–7 handle conditionals by first expanding any variables or function calls in its conditional expression with a call to the Expands procedure on line 5. This returns a list of (e, c) tuples where e is an expansion of the expression and c is the presence condition of the expansion. Each conditional expression is conjoined with its presence condition, and the resulting conjunctions are unioned to produce c_{if}, which represents all the ways the conditional block's expression can be made true and the if-branch entered. Line 6 enters the if-branch with an updated presence condition, recursively calling Statements on the branch's statements. The else-branch is similarly evaluated on line 7, but with the negation of the condition that enters the if-branch as the presence condition.

Variable assignment is handled on lines 8–16. Line 9 first expands the variable name, because the variable name itself can contain variable expansions and function calls. Likewise, the definition is expanded. The nested for loops on lines 11–12 try each combination of variable name and definition that resulted from expanding them. So when a feature is used to add a new compilation unit as in the assignment `obj-$(CONFIG_USB_UAS) += uas.o`, both `obj-y` and `obj-` get assigned. The conditions of the

assignment are stored with the definition in the conditional symbol table. Line 14 updates each variable name's entry in T with a new definition under the combined presence condition that yielded the name and definition combination.

This variable assignment is a simplification of what the Kmax tool actually does, because Makefiles have several assignment operators, each with a different meaning. Variables come in two *flavors* [1]. "=" creates a *recursively-expanded* variable. Its definition gets expanded at the time of the assignment. In contrast, ":=" creates a *simply-expanded* variable whose definition is not expanded until call-time. "?=" only updates the definition if the variable isn't already defined. Variable definitions can be appended to with the "+=" operator. For the latter two operators, a previously undefined variable becomes recursively-expanded by default.

Lines 17–22 evaluate the include statement. The file named in an include statement can also come from variable expansion and function calls, so its name is expanded on line 18. Lines 19-21 parse the statements from the file and evaluate them under the presence conditions of the expanded filenames.

Algorithm 6 defines the Expand procedure that finds all possible expansions of an expression. It takes an expression E, a presence condition p, and a conditional symbol table T. Expand returns a list of all possible expansions of the expression as (e, c) tuples, where e is an expanded expression and c is its presence condition that leads to the expansion. Line 3 initializes the result R with an empty string and the true condition, since it is the only possible expansion so far. A Makefile expression can contain several subexpressions that are either variable expansions, function calls, or string constants. Once expanded, the resulting subexpressions get concatenated. Lines 4–17 loops through each subexpression and hoists its expansions with R to find all possible feature expressions of the expanded subexpressions. The operator $\times$ represents hoisting, which is formally defined as

$$R \times E = [\, (e_1 e_2, c_1 \wedge c_2) \mid (e_1, c_1) \in R \text{ and } (e_2, c_2) \in E \,]$$

where $e_1 e_2$ is concatenation.

Lines 5–7 handle the expansion of a variable v. Line 7 first gets all definitions from the conditional symbol table T. Each definition is recursively expanded, since it may contain more variables or function calls. The resulting expansions are then hoisted with R. Lines 8–12 handle function calls where f is a function name and $a_1, a_2, \ldots$ are its arguments. Its arguments first get expanded on line 10, potentially yielding multiple expansions for each one. Line 12 evaluates f for all possible combinations of arguments

and hoists the results with *R*. Finally, string constants are appended to all subexpressions expanded so far in *R* on lines 13–15. Line 18 returns the final list of expansions *R*.

2.4 Empirical Evaluation

Kmax is evaluated for correctness, by comparing to previous work, and for performance. Section 2.4.1 uses the Linux source code to evaluate Kmax's completeness and correctness. First, all C files in the source tree are reconciled with Kmax's compilation units or confirmed to be non-kernel compilation units. Second, Kmax's found compilation units are each mapped to its corresponding source file. Section 2.4.2 compares the compilation units found by Kmax, KBuildMiner, and GOLEM, and running time performance is compared for all three tools.

2.4.1 Kmax Correctness

Kmax's correctness is evaluated with a two-pronged approach. On the one hand, if kmax gets all possible compilation units from variables, then we can be sure that there are none missing. On the other, we start with all C files in the kernel source code, and ensure that no possible compilation units are missed. If both hold true, then Kmax correctly identifies all compilation units. That Kmax collects all units from Makefile variables is matter of correctness of implementation. Since Kmax evaluates all possible variable definitions of `obj-y` and `obj-m` even in all conditional branches, Kmax collects all compilation units defined in Kbuild files by assignment to these variables. Reconciling all C files in the Linux kernel is more tricky. We need to ensure that any C files not identified by Kmax are truly not kernel compilation units. To do so, we start with all C files contained in codebase. Then we remove all C files Kmax identifies as compilation units. If Kmax is complete, the remaining C files should not be kernel compilation units. This is verified this by hand and with tool support where stated. There are many different C files in the codebase that are not kernel compilation units, and the following is a description of the process of eliminating those C files.

Type of C File	**Count**
Found by Kmax	
Compilation units	19,651
Library compilation units	200
Unconfigurable units	13
Host programs	9
Extra targets	12
Found by hand or additional scripts	
From non-kbuild directories	524
Architecture-specific tools	150
ASM offsets files	31
Included C files	147
Helper programs	13
Skeleton files	3
Staging compilation units	4
Orphaned compilation units	27
Other non-Kbuild	18
Make targets	2
TOTAL C FILES	20,804
All C files in source tree	20,804

Table 2.3: Reconciling C files Linux v3.19 source tree with Kmax's compilation units.

There are two main ways we show a C file is not a kernel compilation unit. The first way is to take the C file name and check by hand that it is not in any Kbuild file for the kernel or that it is in directory that is not part of the kernel. For instance, the `scripts` directory contains the Kbuild and Kconfig tools themselves, including a C program that parses and evaluates Kconfig constraints. This program is not part of kernel program; it is used only used during the build proess. The second way to rule out a C file uses Kmax's ability to collect variable definitions. The Kbuild files are used to compile helper programs such as hex-to-binary converters used during the build process. These non-kernel C files are identified in

other reserved Kbuild variables such as `hostprogs-y`. As with kernel compilation units, we collect these compilation unit names with Kmax and rule them out as kernel compilation units.

Table 2.3 lists the results of accounting for all C files in the kernel source tree. The first column lists the type of C file identified, and the second column lists its count. The C file types are divided into those found from Makefile variables using Kmax and those verified by hand. At the bottom of the table are the number of C files verified followed by the number of C files contained in the entire kernel source tree, computed by running the unix `find` command from the root of the source tree: `find linux/ -name "*.c"`. The vast majority of C files are kernel compilation units, with 19,651 files corresponding to compilation units identified in `obj-y` or `obj-m`. Library compilation units account for another 200 C files identified in `lib-y` and `lib-m`, special Kbuild variables used to build libraries.

There are three types of non-kernel compilation units that Kmax identifies, unconfigurable units, host programs, and extra targets. An unconfigurable unit cannot be activated because of the feature that controls it. Several compilation units in `drivers/acpi/acpica/` are controlled by the Makefile variable `ACPI_FUTURE_USAGE`, which is not a feature. A unit can also become unconfigurable if controlled by an unreachable or unselectable feature. For example, `arch/cris/arch-v32/kernel/smp.o` is controlled by the feature `SMP` that is not defined in the cris architecture's Kconfig files. Even though Kmax excludes these from the list of compilation units, it still finds them in Kbuild Makefiles.

Host programs are tools compiled and run during the build process but not compiled into the kernel. `sound/oss/hex2hex.c`, for example, is a stand-alone program that converts hexadecimal codes to a C array. Kmax finds these in several special Kbuild variables such as `hostprogs-y`. Other programs compiled by Kbuild that are not part of the kernel are put in the special variable 'extra-y', which is used during `make clean` to remove the compiled programs.

Four directories do not contain kernel source, as confirmed both by their omission from Kbuild files and by manually inspecting the directories and Linux documentation. These are `Documentation/`, `samples/`, `scripts/`, and `tools/` and account for 524 C files. Similarly, there are architecture-specific `tools/` directories and bootloader code that is also not part of the kernel proper. ASM offsets files are those used to generate the `asm-offsets.h` header file for the given architecture by compiling the C file to assembly. 147 files have the `.c` extension, but are included via the `#include` directive like headers in other C files. Helper programs are test code or template files that were manually confirmed not to be referenced by Kbuild files and usually contain comments that describe their purpose. Similarly, skeleton files have the

word "skeleton" in their name and are templates for driver writers.

Some C files look like kernel compilation units, but are not referenced by Kbuild. Four of these appear in the `staging/` directory. Drivers in this directory are pending inclusion into the mainline kernel, and may not be completely integrated with Kbuild. 27 unreferenced files are orphaned, perhaps representing dead code or bugs in the Kbuild files. All orphans were investigated manually to confirm their omission from Kbuild. The other non-Kbuild files come from real-mode and user-mode Linux directories and were also manually confirmed not to be used by Kbuild. Lastly, some compilation units do not have the same name as their C counterpart, because the Makefiles use `make` rules to build the unit instead of Kbuild's special `obj-y` variables. These represent a true limitation of Kmax, since it does not evaluate `make` targets.

The limitations of this approach are that compilation units without a source file behind it are not accounted for. Also, gathering all compilation units depends on the correctness of the implementation, which can have bugs, in spite of a correct algorithm that collects all variable definitions.

Type of Unit	**Count**
C files	19,651
ASM files	687
Library files	604
Generated	48
Other non-C files	156
No corresponding source	10
TOTAL UNITS	21,158

Table 2.4: The total number of compilation units found in Linux v3.19 by Kmax with a breakdown by types of unit.

The second evaluation of Kmax correctness starts instead with the compilation units and associates them with their corresponding source files. Table 2.4 shows a breakdown by type of units and their counts. Most compilation units are C files, but there are also hundreds of assembly files. The library compilation units are mostly assembly, as Table 2.3 shows only 200 are C files. 48 of the compilation units do not exist in the source because they are generated while building the kernel, as confirmed by investigating

their Kbuild files. The other types of non-C files are firmware binaries and device tree blobs which are loaded by the bootloader along with the kernel. 10 compilation units are defined in Kbuild, but have missing source code, representing errors or regressions.

Tool	Units	C File Units	Archs Failed
Kmax	21,158	19,651	0
KbuildMiner	17,812	16,948	6
GOLEM	19,601	18,404	0

(a) The total number of compilation units found in Linux v3.19 by each tool, the number of C file units, the number architectures the tool failed to process.

Tool	Found	Missing	Misidentified
Kmax	15,124	–	–
KBuildMiner	14,606	518	450
GOLEM	14,627	497	404

(b) A summary of previous work's precision in extracting compilation units from the x86 version of Linux v3.19, with the number compilation units misidentified for x86.

Tool	Units	Archs Failed	x86	Misidentified
Kmax	13,510	0	9,344	–
KbuildMiner	11,220	0	9,136	195
GOLEM	11,325	0	9,145	185

(c) Comparison with Linux 2.6.33.3.

Table 2.5: A comparison of tools running on Linux v3.19.

2.4.2 Comparison

Kmax is compared to the previous tools KBuildMiner and GOLEM for both correctness and performance. For correctness, each tool was run on the Linux v3.19 source code for each architecture, and their resulting compilation units collected. Since previous work shows both tools running successfully on Linux

v2.6.33.3 [2, 25], the same experiment was repeated for that version. For performance, each tool was repeatedly run on the x86 architecture alone to collect its latency.

Table 2.5 compares the compilation units found by Kmax with those found by the other tools. Table 2.5a list the total number of compilation units extracted by each tool across all architectures, how many correspond to C files, and the number of architectures, if any, the tool failed to process. Kmax extracts more compilation units when compared to both KBuildMiner and GOLEM by about 3,000 and 1,500 respectively. KBuildMiner, however, fails on six out of the 30 architectures, which is partly responsible its low numbers.

To control for these failures, the tools are also compared on the x86 architecture alone in Table 2.5b. This table lists the number of x86 compilation units extracted by each tool, how many are misidentified as being part of the x86 architecture, and the actual number of correct compilation units found. While the number of units found is similar for each tool, this number does not reflect tool precision. When compared to Kmax's results, both KBuildMiner and GOLEM misidentify more than 400 compilation units each as being part of the x86 source code. These units were spot-checked to confirm the misidentification.

Nearly all of these misidentified units appear in Kmax's compilation units for other architectures. For instance, both KBuildMiner and GOLEM identify `drivers/block/ps3disk.c`. This compilation unit is controlled by the `PS3_DISK` feature defined only in `arch/powerpc/platforms/ps3/Kconfig`, making it only available when building for the PowerPC architecture. Another example is `drivers/ide/icside.c`, illustrated in Figure 2.4f as being only available for the arm architecture.

Some misidentified units can never be compiled into the kernel. For example, compilation units from `drivers/acpi/acpica` such as `hwtimer.o` are controlled by `ACPI_FUTURE_USAGE`, apparently a non-feature used a a placeholder for future use. There are seven such compilation units for KBuildMiner and three for GOLEM. The remaining misidentified compilation units were all found in other architectures by Kmax.

To account for tool regressions on newer versions of Linux, the same experiments were conducted on a version used in each tool's own previous work. Table 2.5c shows the number of units, failures, x86 units, and misidentifications for the 2.6.33.3 version of Linux. KBuildMiner does not fail on any architecture, and finds about as many compilation units as GOLEM. However, Kmax's relative performance is comparable to v3.19, finding more than 2,000 more compilation units. The number of misidentifications by both KBuildMiner and GOLEM is also comparable. While there are about half the misidentifications

compared to v3.19, there are also about 40% fewer compilation units overall.

Tool	Language	Min	Mean	Max
Kmax	python	46.69 sec	46.75 sec	46.80 sec
KBuildMiner	java/scala	11.82 sec	12.32 sec	12.87 sec
GOLEM	python	53.96 min	54.56 min	55.04 min

(a) Latency for Linux v2.6.33.3 x86.

Tool	Language	Min	Mean	Max
Kmax	python	84.03 sec	84.15 sec	84.25 sec
KBuildMiner	java/scala	44.17 sec	45.00 sec	45.87 sec
GOLEM	python	3.41 hrs	3.42 hrs	3.43 hrs

(b) Latency for Linux v3.19 x86.

Table 2.6: Latency of each tool to compute the compilation units for the x86 architecture of two Linux versions, v3.19 and v2.6.33.3. Each tool was run five times, plus a warm-up run for KBuildMiner. The minimum, average computed by the mean, and maximum are listed in "sec" for seconds, "min" for minutes, and "hrs" for hours.

The latency of all three tools was tested by running each five times for the x86 architecture of both Linux v2.6.33.3 and v3.19. These experiments were run on a development machine with an Intel Core i5 3.30GHz processor and 8GB of RAM. Table 2.6 lists the tool, the language its written in, and the latency. Since KBuildMiner uses the Java Virtual Machine (JVM), a warm-up run was performed before collecting the five tests to avoid the additional latency incurred by the first run.

Table 2.6a shows the results for Linux v2.6.33.3, listing the minimum, average computed by mean, and maximum of the fives runs. KBuildMiner is the fastest, since it parses the Kbuild Makefiles without having to evaluate them and is written in Java. It takes on average 12.32 seconds for the x86 architecture, while Kmax takes 46.75 seconds on average, nearly four times slower. But both take less than a minute, while GOLEM takes nearly an hour. While also written in python, GOLEM repeatedly executes `make` on each Kbuild Makefile for one or more features at a time. Given the large number of Makefiles and features in each, this process is time-consuming without much better results than the faster KBuildMiner

parsing approach.

Table 2.6b shows the results for the same experiment on Linux v3.19. Linux v2.6.33.3 has 9,344 compilation units for x86, while v3.19 has 15,124, more than 60% more. As expected, all tools take longer. KBuildMiner is still the fastest at 45 seconds, but takes almost four times longer than on v2.6.33.3. GOLEM takes 3.42 hours on average, about 3.5 times longer. Kmax scales somewhat better, taking about twice as long with 84.15 seconds on average.

KBuildMiner's fuzzy parsing is the fastest, while GOLEM is orders of magnitude more time-consuming than both of the other tools. Given the added complexity of Makefile evaluation across software product lines, Kmax incurs a relatively small latency compared to parsing alone and scaled better for a larger version of the Linux kernel. The trade-off is an accurate and precise set of compilation units.

2.5 Limitations

Kmax does not evaluate the complete Make language. It supports variable assignment and expansion, most function calls, and the include statement. Missing are Makefile rules. Rules build a target file by running shell commands and user-defined functions. Rules are used to run helper programs in Kbuild files, but do not limit Kmax's ability to find compilation unit names, which are specified in special Kbuild variables like `obj-y`. This limitation did prevent Kmax from finding two C files that correspond to compilation units with a different name, because rules are used to compile them instead of Kbuild. The `shell` function, like rules, can also be used to call external programs. An external program could potential take features and perform build steps outside the Kbuild specification. This would be problematic for any attempt to find all compilation units from the build system. External programs are used to generate header files like `asm-offsets`, creating an issue for software tools like bug finders that try to process all possible source code, including headers.

Some non-boolean variables are globally defined in non-Kbuild Makefiles, e.g., the `BITS` variable from Figure 2.4b is used to generate some compilation unit names. Other non-booleans are features. `CONFIG_WORD_SIZE` is used to construct compilation unit names in the PowerPC architecture. Not defined in any Makefile, one way get the range of values for this non-boolean is to look at the `default` construct in its Kconfig definition, although not all non-boolean features have explicit defaults. Kmax requires the non-booleans to be preloaded in the conditional symbol table. There are only three such

multiply-defined variables needed by Kmax for Linux v3.19 including the two described above. The last one is `MMU` which, as defined under the microblaze architecture, is either set to `-nommu` or the empty string and is used to choose between two compilation units depending on support for a memory management unit.

2.6 Related Work

Both GOLEM and KBuildMiner are part of greater efforts not just to find compilation units but also to map features to compilation units. Dietrich et al compares GOLEM to other tools including KBuildMiner to evaluate their coverage of compilation units [25]. KBuildMiner is a standalone tool described by Berger et al. [11]. Both GOLEM and KBuildMiner are part of greater efforts not just to find compilation units also map features to compilation units. Tartler et al describes using Undertaker to remove dead code from compilation units, i.e., code that can never be enabled in any software product line [61]. Andersen et al used KBuildMiner to create a feature model for Linux [8]. KBuildMiner has also been used solely as a source for the set of compilation units. Liebig et al uses them for analyzing Linux's variability [46, 47]. Gazzillo and Grimm [35] and Kastner et al [42] tests their parsers on this set of compilation units as well. But since KBuildMiner yields incomplete results, these analyses are not of the complete kernel. Nadi and Holt's Makex, takes a similar approach to KbuildMiner, using fuzzy parsing. Dietrich et al. found it yielded only 75 percent coverage, underperforming both KbuildMiner and GOLEM at 95 percent [**?**]. After adding support for Makefile conditionals, Nadi and Holt report a yield of 85% [**?**].

There are many studies on Linux's feature model, its build system, and its variability mechanisms. Sincero et al identified Kconfig as a feature model [58], and several publications demonstrate building feature models from Kconfig. Berger et al compared Kconfig and another modeling language called CDL to illustrate real-world use of variability modeling [12]. She et al built a formal hierarchy of features for Linux [57]. Dietrich et al quantified the granularity of features in the Linux kernel [26]. Dintzner et al tracked changes in Linux's feature model over time [27]. Tartler et al calculated code coverage for single software product line and maximized coverage with a minimal set of features [60]. Nadi and Holt analyzed Kbuild Makefiles to find anomalies such as unused compilation units [50]. Thum surveys software product line analysis techniques, categorizing methods for modeling features as well as techniques for software tools to deal with variability in software [62].

Formal module systems define variability by decomposing features into modules, achieving variability by combining modules to form software variations [22]. Kastner et al argues that formal models of module composition are not realistic for variability in C projects, and describes a new module language that permits variability within modules and allows for crosscutting, which would otherwise cause an exponential explosion of module combinations for software tools to process [43].

Kmax uses techniques from other tools that process source code containing variability. Garrido et al discuss refactoring C code containing preprocessor directives and macros and introduces conditional symbol tables for storing multiple definitions across combinations of features [33]. Gazzillo and Grimm formalize and use hoisting to evaluate language constructs across feature combinations in their configuration-preserving C parser [35].

2.7 Conclusion

Kmax is a building block for variability-aware software engineering tools that extracts Linux compilation units and their variability information accurately, making heuristic approaches unnecessary. This building block is key to project-wide static analysis tools, such as bug-finders, code browsers, and refactoring tools. The core of Kmax is its algorithm to evaluate `make` language across all combinations of features simultaneously. It collects all possible variable definitions, evaluates all conditional branches, and maintains a presence condition that determines the features controlling the compilation units. Linux's complex build process adds extras challenges, because each architecture forms the root of its own source code hierarchy. Kmax uses the Selectable algorithm to deduce which features belong to which architectures. Kmax is empirically evaluated on the Linux 3.19, demonstrating the completeness and correctness of Kmax's results. Morever, it is compared to two previous solutions that approximate the complete set of compilation units, revealing the limitations of heuristic solutions. A comparison of running time shows that the added complexity needed by Kmax only incurs a small trade-off in running time compared to previous work.

Chapter 3

Bug Finding

3.1 Introduction

SuperC and Kmax, described in Chapters 1 and 2 respectively, are components of all variability-aware software tools. Using the configuration-preserving parsing of SuperC and the project-wide analysis of Kmax, this chapter develops simple bug finders that work across all configurations at the same time. First, Kmax determines the complete set of compilation units and their presence conditions, then SuperC's capacity for implementing semantic analysis is used to detect errors within those compilation units. Abal et al. showed that bugs are caused by Linux's variability and lack automated tool support; they found already-patched bugs by looking through the Linux kernel mailing list [4].

There are five challenges to implementing a cross-configuration bug finders for all variations of a C project like Linux. (1) The tool needs to find the feature model that defines constraints on feature selection, because not all configurations are valid builds of the software product line. Kmax extracts Linux's feature model from the Kconfig files using the Kconfig parser bundled with the Linux build system. (2) The tool needs to find all compilation units comprising the source code, in order to perform project-wide analyses. (3) The tool needs to find the presence conditions of the compilation units, because the build system chooses compilations units according to feature selections. Kmax also handles these two challenges by extracting all compilation units and their variability information. (4) The tool needs to first parse the compilation units across all configurations, which SuperC does. (5) Finally, the tools needs to

perform cross-configuration static analysis. This chapter shows that SuperC enables static analysis across all configurations with support for semantic actions. More sophisticated static analyses are possible with data- and control-flow analyses, but this chapter details bug finders based only on semantic analysis to illustrate the power of Kmax and SuperC alone. It is future work to develop further variability-aware static analyses, and the data structures developed for semantic analysis move towards that future work.

To evaluate our approach, we build a linker error bug finder. This requires not SuperC's semantic actions, but also Kmax's ability to extract variability from the build system. Linker errors happen when one compilation unit calls a function that has no definition. C's separate compilation is used for modularity, and a compilation unit roughly defines a set of related functions. Compilation units that use these functions include a header that declares the imported functions, but the definition of the function is not available until link time. Even if the function definition exists in some compilation unit, a linker error is still possible if there is a configuration in which that compilation unit is excluded by the build system. Such errors are typically only discovered when building the errant configuration in which the error appears, making it difficult to check for such errors. With Kmax's ability to glean the presence conditions of compilation units we can check for linkers errors in all configurations simultaneously.

The contributions of this chapter are the following:

1. Data structures for implementing semantic analysis across all configurations with SuperC's cross-configuration parsing framework,
2. Implementations of a project-wide linker error bug finder, and
3. An evaluation of the linker error finder on the complete Linux kernel.

3.2 Semantic Analysis

Semantic actions are functions that run when the parser recognizes a specific grammar construct. As with the bison parser generator, SuperC supports actions written directly in the grammar specification file. Semantic actions are often used to build an AST during parsing. SuperC supports declarative AST generation, so writing such semantic actions is unnecessary. However, semantic actions can also be used for bug-finding. A single-configuration bug finder asks a question such as "is this symbol being used undefined?" or "does this function call have the correct number of arguments?". But we are asking the question on

```
1  #ifdef CONFIG_CRYPTO_BLKCIPHER
2  void *crypto_alloc_ablkcipher()
3  {
4    return (void*)0;
5  }
6  #endif
7  #ifdef CONFIG_CRYPTO_TEST
8  static void test_cipher()
9  {
10   crypto_alloc_ablkcipher();
11 }
12 #endif
```

Figure 3.1: An example of a variability bug from the variability bug database by Abal et al [3].

variable code, so the answer depends on which configuration is selected; some configurations may have an error and some not. Figure 3.1 is an example from Abal et al's variability bug database [3]. On line 2, the function `crypto_alloc_ablkcipher` is defined only if the `CONFIG_CRYPTO_BLKCIPHER` feature variable is defined. Line 10 makes a call to `crypto_alloc_ablkcipher` inside the function `test_cipher`. But `text_cipher`, defined on line 8, is only defined when `CONFIG_CRYPTO_TEST` is enabled. With two boolean features, there are four possible configurations of this code block, and all of these configurations compiles correctly except one. When `CRYPTO_CRYPTO_TEST` is enabled but `CONFIG_CRYPTO_BLKCIPHER` is not, there is an unidentified symbol error; line 10 calls `crypto_alloc_ablkcipher`, which is not defined in this configuration. To capture variability, instead of asking a true or false question, variability bug finders ask if there is *any* configuration under which there is a bug, e.g., "is there a valid configuration where this symbol is undefined but not used?". This question is modeled as a boolean expression, built from the presence conditions of each line of source code involved in the potential bug. In Figure 3.1, the relevant parts are the presence conditions under which the symbol is defined and under which it is used. Then, checking for a bug in some configuration is checking whether the boolean expression is there some combinations of features that leads to the bug, i.e., satisfiability.

```
1 #ifdef A
2 duped int x;
3 #else
4 int x;
5 #endif
6
7 int main() {
8   int y;
9
10  x *y;
11 }
```

Figure 3.2: An example of the same C identifier declared as a typedef name in one configuration, but a variable in another.

Since SuperC's parser already tracks configurations, performing bug checking during parsing is ideal. Each subparser maintains the current presence condition it's parsing, and semantic actions in the grammar are executed by the subparsers as usual; actions are written in-line in grammar productions and are executed after the grammar construct is recognized. Each subparser maintains its own parsing context, allowing it not only to parse constructs from a different configurations but to record configuration-specific semantic information, such as symbol definitions. Subparsers are temporary, being created and destroyed by forking and merging as new configuration are encountered. Being managed by the subparser, the parsing context must follow suit, and SuperC provides an interface for implementing cross-configuration a parsing context. It has hooks to fork and and merge corresponding to subparser forking and merging. To store semantic information for bug finders, the parsing context is used to manage a conditional symbol table. As in SuperC and Kmax, this a conditional symbol table maps identifiers to each possible value across all configurations.

To illustrate how cross-configuration semantic analysis works in practice, we illustrate the implementation of typedefs, because SuperC's C parse already performs some semantic analysis to support them. This context-sensitive aspect of the C language requires maintaining a table of typedef names and ref-

erencing it, to reclassify identifier tokens as typedef names during parsing. Implementing this behavior using semantic actions. Typedef declarations take a C identifier and convert it to a typename, making C a context-sensitive language, which cannot be recognized by context-free parser generators without extra support. Typedefs are context-sensitive, because the same string can be recognized with a different grammar construct depending on whether an identifier has been declared a typedef name earlier in the program. Figure 3.2 illustrates this context-sensitivity. Line 2 defines `x` as a typedef name if `A` is true, otherwise it is a variable. The statement on line 10 is either a multiplication expression, when both `x` and `y` are C identifiers, or it is a pointer declaration when `x` is a typedef name. Implementing typedefs for a single configuration is simple: the parsing context maintains a symbol table of typedef declarations, mapping C identifiers to a boolean flag. A semantic action embedded with the declaration grammar construct looks for the typedef keyword and maps the declared identifier to true in the symbol table. When reading tokens from the lexer, the parser consults this symbol table and reclassifies identifiers to typedef names tokens as necessary. Like variables, typedef declarations may appear in any lexical scope, so the parsing context maintains scope. Our implementation of the context uses a stack of symbol tables to represent scope, which makes sharing context between forked subparsers as it does with the LR state stack. As a further optimization, even forked subparsers point to the same symbol table, possible because the subparsers' presence conditions are always mutually exclusive; any updates to the symbol table are independent. The only time subparsers point to different symbol tables is when they enter a new scope or leave the scope after forking.

SuperC's parsing context interface support arbitrary implementations of cross-configuration semantic information, with hooks called by the parser upon forking and merging. The interface contains the following methods:

1. `forkContext` creates a new context and is called when SuperC forks a subparser.

2. `mayMerge` determines whether two contexts allow merging, if not, SuperC will delay merging subparsers until their contexts allow, for instance by delaying a merge until the subparsers return to the same lexical scope.

3. `mergeContexts` combines two contexts, merging their state, and is called when SuperC merges two subparsers.

4. `reclassify` takes a token and changes or adds the token and is used to implement typedef names.

We illustrate how semantic state is processed and stored while parsing the typedef example Figure 3.2. On line 1, SuperC forks two subparsers, one to enter the `#ifdef` branch under the A presence condition and one to enter the `#else` under the mutually-exclusive $\neg A$ presence condition. Initially, the parsing context contains an empty table. After parsing their respective branches, each subparser encounters the same semantic action for declarations. By default, the `x` keyword is mapped to false. The first subparser, seeing the typedef keyword, updates the entry for `x` in the symbol table. The subparser computes the new entry by disjoining its own presence condition, $C_{\text{subparser}}$ with the original presence condition in the table, C_{original}, i.e.,

$$C_{\text{new}} \leftarrow C_{\text{original}} \vee C_{\text{subparser}}$$

The new condition for `x` in the table becomes $\bot \vee A$. This means that `x` is a typedef whenever the expression A is true. The second subparser sees that that `x` is declared as a variable and removes this configuration from the entry by conjoining the negation of its presence condition, i.e.,

$$C_{\text{new}} \leftarrow C_{\text{original}} \wedge \neg C_{\text{subparser}}$$

Since the `#else` branch's presence condition is $\neg A$, the new condition becomes $(\bot \vee A) \wedge \neg(\neg A)$, which when simplified, is still A. After the static conditional, the subparsers merge, leaving a single parser on line 7. Parsing continues until line 10, which uses the `x` identifier. The parser consults the symbol table to find that the identifier is a typedef in only some configurations, and forks two subparsers, one for the typedef presence condition and one for the non-typedef presence condition.

The same principles used to support typedef names apply to cross-configuration bug finders, albeit with more semantic information and extra semantic actions. For example, to support detection of undefined symbol uses, the bug finder deduces whether there exists some combination of features where the undefined symbol gets used. It models the bug by taking the presence condition C_{def} under which the symbol is defined and the presence condition C_{use} for a use of the symbol and constructs the following expression:

$$C_{use} \wedge \neg C_{def}$$

If the above expression is satisfiable, then there is some configuration where the bug exists. Further constraints to the set of configurations may be conjoined to the expression, for example, the Kconfig

```
1 #ifdef CONFIG_TRACING
2 void trace_dump_stack(int skip) {
3   // do something
4   return;
5 }
6 #else
7 static inline void trace_dump_stack(void) { }
8 #endif
9
10 int main(int argc, char** argv) {
11   trace_dump_stack(0); // ERROR
12   return 0;
13 }
```

Figure 3.3: An example of an error caused by the wrong number of arguments to a function that only appears on one configurations found by Abal et al [3].

feature model. The undefined symbol finder updates the parsing context in the same way that the typedef implementation does, except that the symbol table stores the conditions in which the symbol is defined. To use this information, a new semantic action for C expressions, where functions and variables get used, constructs the model for the bug's presence condition and uses a SAT solver to determine whether the bug appears in any configuration.

To store more semantic information for more sophisticated bug finders, a conditional symbol table is useful. First described by Garrido for use in configuration-preserving parsing [33], the conditional symbol table is useful for all variability-aware tools, including Kmax and SuperC themselves. A conditional symbol table maps keys to a list of values, where each value is tagged with a presence condition. Figure 3.3, also from Abal et al, is a function that has a different number of arguments depending on the configuration. To create a finder for this bug, a conditional symbol table stores an entry for each possible number of arguments and its presence condition. After parsing the mutually exclusive function definitions on lines 1–8, the symbol table maps the `trace_dump_stack` function to two entries, one entry records one argument under the `CONFIG_TRACING` presence condition and other entry records zero arguments for

¬ CONFIG_TRACING. A semantic action function calls checks for this bug. It takes the presence condition at the call site on line 11, which passes one argument to trace_dump_stack. The finder collects the presence conditions for all symbol entries other than the entry recording one argument, and conjoins it with the presence condition at the call site to deduce whether any configurations have a bug. Figure 3.3 does have a bug when CONFIG_TRACING is not enabled.

3.3 Project-Wide Analysis

To support project-wide tasks, software tools need more than analysis tools for one compilation unit. They need to work on the entire project as a whole, i.e., all compilation units. Worse, compilation units have their own presence conditions, while the build system adds constraints on legal configurations. Kmax extracts all of this information for the Linux kernel build system. To illustrate project-wide analysis, we demonstrate a bug finder for linker errors. Because compilation units are linked into one global namespace, any compilation unit can call a function from any other. With variability, linker errors may only appear in some configurations, because there are calls to function whose definitions are omitted by the selection of features. These linkers error are not due to a missing function definition per se, but are caused by bugs in the variability specification itself. Take Figure 3.1. This example is only a simplified version of the bug. Lines 1–6 and 7–12 actually appear in different compilation units. Lines 2–5 appear in the crypto/ablkcipher.c, while lines 9–11 appear in crypto/tcrypt.c. The conditionals in the example correspond to the presence conditions of the compilation units as defined in the Kbuild Makefiles. crypto/ablkcipher.c is only compiled and linked when CONFIG_CRYPTO_BLKCIPHER is enabled, and crypto/tcrypt.c when CONFIG_CRYPTO_TEST is enabled.

Fixing this kind of variability bug may be as simple as adding a new constraint to Kconfig making the configuration illegal, but finding it in the first place is hindered by the enormous number of possible configurations. In this section, we describe how Kmax and SuperC are used together to implement a finder for project-wide linker errors in Linux. The general approach to addressing project-wide analysis consists of (1) finding the feature model, (2) identifying all compilation units, (3) finding the compilation units' presence condition, (4) performing analysis on the compilation units. For a linker error finder for Linux, Steps 1–3 are handled by Kmax, while SuperC's preprocessor, parser, and semantic analysis support handle step 4 as described in the Section 3.2. First, Kmax is run on the Linux kernel source code,

producing the feature model, the compilation units, and their presence conditions. Then SuperC is run to collect all function calls and global function definitions. To find function calls, we start with the bug finder for undefined functions, adapting it to find the name and presence condition of each call an externally-defined function. Then, semantic actions for function definitions record the function name and presence condition for those that are not static and not inline, since these are not exposed to other compilation units. These analyses are run on all compilation units, producing a database of calls to functions missing a definition and global function definitions.

To check for linkers errors, we take each function call and match it to its function definition in another compilation unit, and check each call for configurations that have a bug. We build a boolean expression represents the bug and use a SAT solver to test for a possible configuration. This expression involves presence conditions from many sources: the Kconfig model C_{kconfig}, the compilation unit that calls the function $C_{\texttt{callunit}}$, the function call itself $C_{\texttt{call}}$, the defining compilation unit $C_{\texttt{defunit}}$, and the definition itself $C_{\texttt{def}}$. The boolean expression representing the configurations in which the bug can appear is as follows:

$$C_{\text{kconfig}} \wedge C_{\texttt{callunit}} \wedge C_{\texttt{call}} \wedge \neg(C_{\texttt{defunit}} \wedge C_{\texttt{def}})$$

This asks whether there are any configurations where the function is called, but either the compilation unit is not included or the function is not defined. As usual, a SAT solver finds whether the bug is possible.

3.4 Evaluation

There are four pieces of information necessary for checking for errors. First, we use Kmax to extract the Kconfig feature model as a list of boolean expressions representing the constraints it imposes, e.g., if `CONFIG_XX` then `CONFIG_YY` must also be enabled. To use with a SAT solver, these constraints are converted will be converted to conjunctive normal form (CNF) when the finder runs. Second, Kmax collects the presence conditions controlling each compilation unit, producing a file that maps compilation unit to a boolean expression of configuration variables. These expressions too will be converted to CNF. Third, SuperC is used to identify global function definitions and calls from all compilation units. Fourth, SuperC records the presence conditions of these definitions and calls. These presence conditions are found using the conditional symbol table maintained during semantic analysis. To run the experiment,

Kmax is first run to gather the compilation units and their presence conditions. Then SuperC is run on each compilation unit to record all function calls and definitions along with their presence conditions. Then a program matches calls with their definitions in other compilation units. For each call, it builds the boolean expression containing all Kconfig constraints, the compilation units' presence conditions, and both the call-site and definition-site presence condition. Each is converted into conjunctive normal form and tested for satisfiability useing the `Sat4j` SAT solver library for Java [56].

For Linux v4.0, the linker error finder takes roughly two days to complete on a commodity PC with an Intel Core i5 and 16GB of RAM. Half of the time is devoted to parsing the files with SuperC and matching function calls. The bug finder processes 2,893,236 function calls that into other compilation unit and determines that 996,984 of them are potential linker errors, which include false positives. Matching function calls and testing with the SAT solver takes surprisingly little time considering how many function calls there are and even though there are 12,673 Kconfig constraints to add to each function call test. The linker error bug checker was tested with a small set of known errors from Abal et al [4] and by reintroducing known linker errors by modifying Kconfig constraints. Since the main objective of the linker error bug finder, however, is to scale bug finding to the massive variability of the Linux source code, little has been done to prevent false positives or to ensure soundness. Making the bug finder more functional requires further development. Inspecting the potential bugs discovered via SAT solving its boolean expressions, there were several false positives found. At least some of these are engineering problems.

Figure 3.4 shows three such types of false positives. (a shows two different compilation units defining the same function, `vm_brk`. The bug finder as currently engineered does not unify these multiple definitions, resulting in two separate checks for each call to `vm_brk`. In some cases, the call to the definition in `mm/nommu.c` is never possible in any configuration, even while the other definition satisfies the call. This is erroneously labeled a bug, and unioning the presence conditions of all possible definitions of the function would avoid this false positive. (b) is an example of a function only defined if a macro of the same name is undefined. Presumably this allows the function to be defined as a macro, and the bug finder determines that this is a potentially undefined function. Without knowing the possible external macro definitions, such a false positive is difficult to handle. Last of the known false positives, (3.4c) is a case where a function definition is guarded by a macro, and `strcmp` will not be defined here if the `__HAVE_ARCH_STRCMP` macro is defined. The bug finder labels the configuration where this macro is

```
// defined in mm/nommu.c
unsigned long vm_brk(unsigned long addr, unsigned long len) {
  return -ENOMEM;
}
// defined in mm/mmap.c
unsigned long vm_brk(unsigned long addr, unsigned long len) {
  // ...
  return ret;
}
```

(a) Global functions may be defined in several compilation units.

```
#ifndef div_s64
static inline s64 div_s64(s64 dividend, s32 divisor) {
  s32 remainder;
  return div_s64_rem(dividend, divisor, &remainder);
}
#endif
```

(b) Some functions may be specified with a macro instead of a C function. From include/linux/math64.h.

```
#ifndef __HAVE_ARCH_STRCMP
int strcmp(const char *sc, const char *ct) {
  // ...
}
#endif
```

(c) A function definition guarded by a macro. From lib/string.c

Figure 3.4: Examples of false positives in the linker error bug finder.

defined as a bug, since the function definition is missing, even though the function may be provided inline by gcc. Extra information is needed by the bug finder to avoid such a false positive.

Even though finding variability bugs is equivalent to SAT, these results show that the boolean expression, while containing tens of thousands of clauses, is within easy reach of modern SAT solvers. Further work on the bug finder, however, is needed to reduce false positives and make it effective for real-world C code.

3.5 Related Work

Abal et al found variability bugs in the Linux kernel manually and noted the lack of automated tools available to find them [4]. They found bugs that had already been patched and reported to the Linux kernel mailing list and produced a database on such bugs that details the causes of the bugs and shows an simplified version of the code that causes the bugs. This database is a perfect set of test cases for bug finders.

Linux's Kconfig files that describe the constraints on legal configurations have long been identified as a feature model, and Sincero et al make one of the earlier arguments for treating Linux as a software product line [58]. Dintzner et al track changes in the Linux feature model with the FMdiff tool [27]. Much previous work has also represented Linux configurations as boolean expression [33, 35, 42] and used SAT solvers on these expressions. Tartler et al extracted Kconfig constraints and compilation unit presence conditions [61]. This variability information is used to identify hundreds of instances of superfluous or dead code in the Linux source code. It uses SAT solving to confirm preprocessor conditionals surround code that belong to no legal configurations, but goes no further in analyzing the source code itself. Brabrand et al, however, described variability-aware data-flow analysis [18]. They compared the latency cross-configuration data-flow analysis versus one configuration at at time, but did not use it for bug checking.

This chapter's project-wide analysis and bug finders for linker errors is related to previous work on module systems for modeling separate compilation and variability. Cardelli provides a formal specification of for modularization [22]. Kastner et al build on this work to describe a new module specification language appropriate for C systems and checking for type errors in BusyBox [43]. The work reconciles formal module theory with practice, arguing that module systems need to permit intramodule variability

and expect features to crosscut modules, because C developers do not treat compilation units as independent, composable modules like formal systems do. More previous work related to static tools and parsing can be find in in Section 1.7 the chapter of SuperC

3.6 Conclusion

SuperC and Kmax are components of all variability-aware software tools. This chapter describes how these components enable semantic analysis across all configurations on real-world C code, including bug finders, which lack good support. It introduces the fork-merge parsing context, which enables a cross-configuration parser to maintain state while subparsers fork and merge. Symbol tables for semantic analysis, as with other configuration-preserving tools, are conditional symbol tables that maintain state for all configurations simultaneously. With only SuperC and these data structures, cross-configuration bug finders are possible by modeling the conditions of bugs with a boolean expression and using a SAT solver to discover the erroneous configurations. Kmax enables project-wide finders, because it extracts the feature model and collects all compilation units and their presence conditions. We show that this approach is feasible by evaluating it on the Linux kernel with a finder for linker errors.

Conclusion

Large-scale C software needs good tool support, such as bug finders, code browsers, and refactorings. Variability impedes creating such tools, because of its ad-hoc implementation with the C preprocessor and `make`. Because the preprocessor extends the C language itself, new variability formalisms alone are not enough to support static tools for C. To provide a solid foundation for software engineering tools, we focus on the most basic tasks tools need: project-wide analysis, parsing, and semantic analysis. Simple for the C language alone, these tasks are complicated by the preprocessor and variability, requiring new variability-aware algorithms. With a thorough analysis of the challenges of parsing, SuperC's preprocessor handles all preprocessor usage without heuristics, and its parser preserves all configurations of the source in its AST. Cross-configuration semantic analysis allows some support for bug checking, including undefined symbol errors. Kmax extracts the Linux feature model, and uses a configuration-preserving `make` evaluator to capture project-wide variability from the build system. Together, these components form the beginnings of static tools that support real-world C usage in systems software and that scale to large codebases. Data- and control-flow analysis are important future work as they are essential for more powerful tools. Refactorings especially depend on flow analyses for more sophisticated code improvements, such as creating a new function out a code snippet. Because variability is preserved the AST, data-flow analyses need to work with this variability. For instance, a variable may be live in one configuration but not another. These analyses may benefit from the variability-aware data structures used by the bug checkers. Static analysis is difficult enough without involving the preprocessor and software product lines. With Kmax and SuperC, the most fundamental challenges are solved.

Future Work

Further engineering is needed for bug finders built with semantic analysis to make them effective for real-world C code and eliminate some of the found false positives. Since SuperC's parser supports arbitrary semantic actions, full type checking can be implemented with them. Type checking, however, is a challenge because symbols may be declared as different types in different configurations. Checking such symbols when usd in an expression creates implicit conditionals, just like with multiply-defined macros do, that require checking each possible type of the symbol wherever it is used. Cross-configuraion type-checking would help find other types of compile-time errors and also provide support for software engineering tasks, such as refactoring.

In general, SuperC and Kmax work with boolean configuration variables. Support for non-boolean configuration variables is acheived by giving SuperC a list of all possible non-boolean definitions ahead of time. Borrowing from work on satisfiability modulo theories should help SuperC and all tools that reason about software variability support non-boolean configuration information.

Kmax is designed around the Linux build system, but its Makefile evaluator is fairly general purpose. Future work to modify Kmax to work on any project's Makefile will bring variability analysis to more software. Moreover, there are many other build systems besides `make`, including `cmake` and Apache Maven. Any build process that conditionally includes parts of the software have the same challenges that Kmax solves for the Linux build system.

Bug finding with semantic analysis is only the beginning. Kmax and SuperC enable further static analysis algorithms such as data- and control-flow analysis. These analyses in turn are the basis for more sophisticated bug finders and refactoring tools. Along with full typechecking, these analyses will bring software engineering even close to support software variability.

Bibliography

[1] GNU make Manual. `https://www.gnu.org/software/make/manual`.

[2] Kbuildminer. `https://code.google.com/p/variability/`.

[3] I. Abal, C. Brabrand, and A. Wasowski. The variability bugs database. `http://vbdb.itu.dk/`.

[4] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pp. 421–432, New York, NY, USA, 2014. ACM.

[5] B. Adams et al. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, pp. 243–254, Mar. 2009.

[6] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, Aug. 2006.

[7] R. L. Akers et al. Re-engineering C++ component models via automatic program transformation. In *Proceedings of the 12th WCRE*, pp. 13–22, Nov. 2005.

[8] N. Andersen, K. Czarnecki, S. She, and A. Wąsowski. Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pp. 106–115, New York, NY, USA, 2012. ACM.

[9] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Software: Practice and Experience*, 30(8):907–924, July 2000.

[10] I. D. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the 8th WCRE*, pp. 281–290, Oct. 2001.

[11] T. Berger, S. She, K. Czarnecki, and A. Wasowski. Feature-to-code mapping in two large product lines. Tech. report, University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[12] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 73–82, New York, NY, USA, 2010. ACM.

[13] A. Bessey et al. A few billion lines of code later: Using static analysis to find bugs in the real world. *CACM*, 53(2):66–75, Feb. 2010.

[14] A. Birman and J. D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, Aug. 1973.

[15] A. M. Bishop. C cross referencing and documenting tool. `http://www.gedanken.demon.co.uk/cxref/`.

[16] B. Blanchet et al. A static analyzer for large safety-critical software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 196–207, June 2003.

[17] R. Bowdidge. Performance trade-offs implementing refactoring support for Objective-C. In *Proceedings of the 3rd WRT*, Oct. 2009.

[18] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pp. 13–24, New York, NY, USA, 2012. ACM.

[19] M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 365–383, Oct. 2004.

[20] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[21] BusyBox. `http://www.busybox.net/`.

[22] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pp. 266–277, New York, NY, USA, 1997. ACM.

[23] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 73–88, Oct. 2001.

[24] F. DeRemer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *TOPLAS*, 4(4):615–649, Oct. 1982.

[25] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. A robust approach for variability extraction from the linux build system. In *Proceedings of the 16th International Software Product Line Conference*, pp. 21–30, Sept. 2012.

[26] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding linux feature distribution. In *Proceedings of the 2012 Workshop on Modularity in Systems Software*, pp. 15–20, Mar. 2012.

[27] N. Dintzner, A. Van Deursen, and M. Pinzger. Extracting feature model changes from the linux kernel using fmdiff. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pp. 22:1–22:8, New York, NY, USA, 2013. ACM.

[28] M. D. Ernst et al. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, Dec. 2002.

[29] J.-M. Favre. Understanding-in-the-large. In *Proceedings of the 5th IWPC*, pp. 29–38, Mar. 1997.

[30] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pp. 111–122, Jan. 2004.

[31] Free Software Foundation. Bison. `http://www.gnu.org/software/bison/`.

[32] É. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, Mar. 1998.

[33] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[34] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proceedings of the 21st ICSM*, pp. 379–388, Sept. 2005.

[35] P. Gazzillo and R. Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 323–334, June 2012.

[36] A. G. Gleditsch and P. K. Gjermshus. The LXR project. `http://lxr.sourceforge.net/`.

[37] E. Graf et al. Refactoring support for the C++ development tooling. In *Companion 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 781–782, Oct. 2007.

[38] R. Grimm. Better extensibility through modular syntax. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 38–51, June 2006.

[39] java.net. JJTree reference documentation. `http://javacc.java.net/doc/JJTree.html`.

[40] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. In *Proceedings of the 35th ACM Symposium on Theory of Computing*, pp. 355–364, June 2003.

[41] C. Kästner et al. Partial preprocessing C code for variability analysis. In *Proceedings of the 5th ACM Workshop on Variability Modeling of Software-Intensive Systems*, pp. 127–136, Jan. 2011.

[42] C. Kästner et al. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 805–824, Oct. 2011.

[43] C. Kästner et al. A variability-aware module system. In *Proceedings of the 27th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 773–792, Oct. 2012.

[44] G. Klein et al. JFlex: The fast scanner generator for Java. `http://jflex.de/`.

[45] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec. 1965.

[46] J. Liebig et al. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32th International Conference on Software Engineering*, pp. 105–114, May 2010.

[47] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 81–91, New York, NY, USA, 2013. ACM.

[48] B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proceedings of the 10th European Software Engineering Conference*, pp. 21–30, Sept. 2005.

[49] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proceedings of the 13th International Conference on Compiler Construction*, vol. 2985 of *LNCS*, pp. 73–88, Mar. 2004.

[50] S. Nadi and R. Holt. Make it or break it: Mining anomalies from linux kbuild. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pp. 315–324, Washington, DC, USA, 2011. IEEE Computer Society.

[51] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proceedings of the 18th International Conference on Compiler Construction*, vol. 5501 of *LNCS*, pp. 109–125, Mar. 2009.

[52] T. Parr and K. Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 425–436, June 2011.

[53] M. Platoff et al. An integrated program representation and toolkit for the maintenance of C programs. In *Proceedings of the ICSM*, pp. 129–137, Oct. 1991.

[54] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proceedings of the 1st ACM Symposium on Theory of Computing*, pp. 165–180, May 1969.

[55] J. Roskind. Parsing C, the last word. The comp.compilers newgroup, Jan. 1992. `http://groups.google.com/group/comp.compilers/msg/c0797b5b668605b4`.

[56] Sat4j. Sat4j. `http://www.sat4j.org/`.

[57] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 461–470, New York, NY, USA, 2011. ACM.

[58] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the linux kernel a software product line? In *Proceedings of the International Workshop on Open Source Software and Product Lines*, SPLC-OSSPL, pp. 134–140, 2007.

[59] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, Nov. 2003.

[60] R. Tartler et al. Configuration coverage in the analysis of large-scale system software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, Dec. 2011.

[61] R. Tartler et al. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the 6th European Conference on Computer Systems*, pp. 47–60, Apr. 2011.

[62] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014.

[63] M. Tomita, ed. *Generalized LR Parsing*. Kluwer, 1991.

[64] uClibc. `http://www.uclibc.org/`.

[65] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.

[66] M. Vittek. Refactoring browser with preprocessor. In *Proceedings of the 7th IEEE European Conference on Software Maintenance and Reengineering*, pp. 101–110, Mar. 2003.

[67] J. Whaley. JavaBDD. `http://javabdd.sourceforge.net/`.