SuperC: Parsing All of C by Taming the Preprocessor

Paul Gazzillo Robert Grimm

New York University

{gazzillo,rgrimm}@cs.nyu.edu

Abstract

C tools, such as source browsers, bug finders, and automated refactorings, 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. This paper presents a complete, performant solution to this problem. 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 the effectiveness of our approach on the x86 Linux kernel.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Languages, Algorithms

Keywords C, preprocessor, LR parsing, Fork-Merge LR parsing, SuperC

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 [14].

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 [8], the Astrée bug finder [9], and Xcode refactorings [10]), rely on a single, maximal configuration (e.g., the Coverity bug finder [6]), or build on incomplete heuristics (e.g., the LXR source browser [20] and Eclipse refactorings [21]). Processing one configuration at a time is infeasible for large programs such as Linux, which has over 10,000 configuration variables [38]. 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 [37]. 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 [1, 3–5, 15, 19, 29, 31, 36, 41].

Only MAPR [33] and TypeChef [25, 26] 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 [39], 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.

2. The Problem and Solution Approach

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 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.

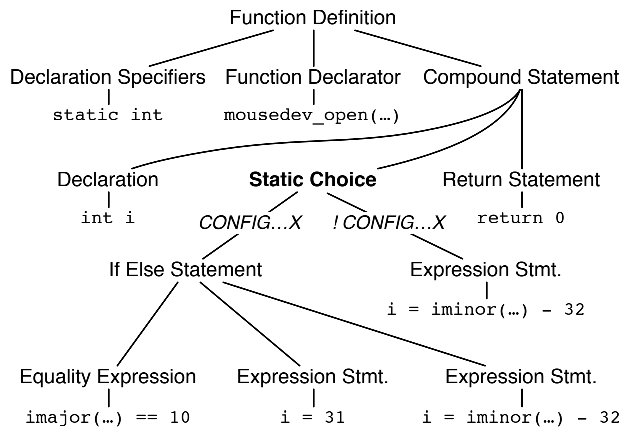
Figure 1 illustrates SuperC's configuration-preserving preprocessing and parsing on a simple example from the x86 Linux kernel (version 2.6.33.3, which is used throughout this paper). Figure 1a shows the original source code, which utilizes the three main preprocessor facilities: an include directive 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 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

```
 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 }
```

(a) The unpreprocessed source.

```
 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 }
```

(b) The preprocessed source preserving all configurations.



(c) Sketch of the AST containing all configurations.

Figure 1. From source code to preprocessed code to AST. The example is edited down for simplicity from drivers/input/mousedev.c.

on lines 5 and 9. Finally, in Figure 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 1b.

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 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

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

Language Construct	Implementation	Surrounded by Conditionals	Contain Conditionals	Contain Multiply-Defined Macros	Other
Lexer					
Layout	Annotate tokens				
Preprocessor					
Macro (Un)Definition	Use conditional macro table	Add multiple entries to macro table		Do not expand until invocation	Trim infeasible entries on redefinition
Object-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions		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		
File Includes	Include and preprocess files	Preprocess under presence conditions		Hoist conditionals around includes	Reinclude when guard macro is not false
Static Conditionals	Preprocess all branches	Conjoin presence conditions		Ignore infeasible definitions	
Conditional Expressions	Evaluate presence conditions			Hoist conditionals around expressions	Preserve order for non-boolean expressions
Error Directives	Ignore erroneous branches				
Line, Warning, & Pragma Directives	Treat as layout				
Parser					
C Constructs	Use FMLR Parser	Fork and merge subparsers			
Typedef Names	Use conditional symbol table	Add multiple entries to symbol table			Fork subparsers on ambiguous names

Table 1. Interactions between C preprocessor and language features. Gray entries in the last three columns are newly supported by SuperC.

```
#ifdef CONFIG_64BIT
#define BITS_PER_LONG 64
#else
#define BITS_PER_LONG 32
#endif
```

Figure 2. A multiply-defined macro from include/asm-generic/bitsperlong.h.

entries indicate how to overcome the complications. Blank entries indicate impossible interactions. 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.

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*

```
// In include/linux/byteorder/little_endian.h:
#define __cpu_to_le32(x) ((__force __le32)(__u32)(x))

#ifdef __KERNEL__
// Included from include/linux/byteorder/generic.h:
#define cpu_to_le32 __cpu_to_le32
#endif

// In drivers/pci/proc.c:
_put_user(cpu_to_le32(val), (__le32 __user *) buf);
```

Figure 3. A macro conditionally expanding to another macro.

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 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 [19].

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.

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

(a) After expansion of `cpu_to_le32`.

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

(b) After hoisting the conditional.

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

(c) After expansion of `__cpu_to_le32`.

Figure 4. Preprocessing `cpu_to_le32(val)` in Fig. 3:10.

```
#define uintBPL_t uint(BITS_PER_LONG)
#define uint(x) xuint(x)
#define xuint(x) __le ## x

uintBPL_t *p = ... ;
```

(a) The macro definitions and invocation.

```
__le ##
#ifdef CONFIG_64BIT
64
#else
32
#endif
*p = ... ;
```

(b) After expanding the macros.

```
#ifdef CONFIG_64BIT
__le ## 64
#else
__le ## 32
#endif
*p = ... ;
```

(c) After hoisting the conditional.

Figure 5. A token-pasting example from fs/udf/balloc.c.

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.

Figures 3 and 4 illustrate the hoisting of conditionals. Figure 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 4 shows the three stages of preprocessing the sequence. First, in 4a, the preprocessor expands `cpu_to_le32`, which makes the conditional explicit but also breaks the nested macro invocation on line 2. Second, in 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 4c, the preprocessor recursively expands `__cpu_to_le32` on line 2, which completes preprocessing for the sequence.

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 5 illustrates this for token-pasting: 5a shows the source code; 5b shows the result of expanding all macros, including `BITS_PER_LONG` from Figure 2; and 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 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).

```
static int (*check_part[])(struct parsed_partitions *) = {
#ifdef CONFIG_ACORN_PARTITION_ICS
  adfspart_check_ICS,
#endif
#ifdef CONFIG_ACORN_PARTITION_POWERTEC
  adfspart_check_POWERTEC,
#endif
#ifdef CONFIG_ACORN_PARTITION_EESOX
  adfspart_check_EESOX,
#endif
  // 15 more, similar initializers
  NULL
};
```

Figure 6. An example of a C construct containing an exponential number of unique configurations from fs/partitions/check.c.

Since there is no known efficient algorithm for comparing arbitrary polynomials [24], 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.

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 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 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.

Typedef names. A final complication results from the fact that C syntax is context-sensitive [35]. 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 [22, 35]. 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.

Algorithm 1 Hoisting Conditionals

```
 1: procedure HOIST($c$, $\tau$)
 2:     ▷ Initialize a new conditional with an empty branch.
 3:     $C \leftarrow [\,(c, \bullet)\,]$
 4:     for all $a \in \tau$ do
 5:         if $a$ is a language token then
 6:             ▷ Append $a$ to all branches in $C$.
 7:             $C \leftarrow [\,(c_i, \tau_i a) \mid (c_i, \tau_i) \in C\,]$
 8:         else ▷ $a$ is a conditional.
 9:             ▷ Recursively hoist conditionals in each branch.
10:             $B \leftarrow [\,b \mid b \in \text{HOIST}(c_i, \tau_i) \text{ and } (c_i, \tau_i) \in a\,]$
11:             ▷ Combine with already hoisted conditionals.
12:             $C \leftarrow C \times B$
13:         end if
14:     end for
15:     return $C$
16: end procedure
```

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.

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 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 4b and for `__le ##` in Figure 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.

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. 4c:4.

3.2 Converting Conditional Expressions

To reason about presence conditions, SuperC converts conditional expressions into Binary Decision Diagrams (BDDs) [12, 42], 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 [12]. 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.

4. The Configuration-Preserving Parser

SuperC's configuration-preserving FMLR parser builds on LR parsing [2, 28], 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

Algorithm 2 Fork-Merge LR Parsing

```
 1: procedure Parse($a_0$)
 2:     $Q$.init(($true$, $a_0$, $s_0$))  ▷ The initial subparser for $a_0$.
 3:     while $Q \neq \emptyset$ do
 4:         $p \leftarrow Q$.pull()  ▷ Step the next subparser.
 5:         $T \leftarrow$ 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(Fork($T$, $p$))
12:         end if
13:         $Q \leftarrow$ Merge($Q$)
14:     end while
15: end procedure
```

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.

Compared to top-down parsing techniques, such as LL [34] and PEG [7, 16], 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 [13] 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.

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 next token or conditional a, which is also called the *head*, and an LR parser 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 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,

Algorithm 3 The Token Follow-Set

```
1:  procedure FOLLOW($c$, $a$)
2:      $T \leftarrow \emptyset$   ▷ Initialize the follow-set.
3:      procedure FIRST($c$, $a$)
4:          loop
5:              if $a$ is a language token then
6:                  $T \leftarrow T \cup \{(c, a)\}$
7:                  return $false$
8:              else   ▷ $a$ is a conditional.
9:                  $c_r \leftarrow false$   ▷ Initialize remaining condition.
10:                 for all $(c_i, \tau_i) \in a$ do
11:                     if $\tau_i = \bullet$ then
12:                         $c_r \leftarrow c_r \vee c \wedge c_i$
13:                     else
14:                         $c_r \leftarrow c_r \vee$ FIRST$(c \wedge c_i, \tau_i(1))$
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:                 $c \leftarrow c_r$
21:                 $a \leftarrow$ next token or conditional after $a$
22:             end if
23:         end loop
24:     end procedure
25:     loop
26:         $c \leftarrow$ FIRST$(c, a)$
27:         if $c = false$ then return $T$ end if   ▷ Done.
28:         $a \leftarrow$ next token or conditional after $a$
29:     end loop
30: end procedure
```

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.

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. 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 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 configuration 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.

$$\text{FORK}(T, p) := \{\, (c, a, p.s) \mid (c, a) \in T \,\}$$
$$\text{MERGE}(Q) := \{\, (\textstyle\bigvee p.c, a, s) \mid a = p.a \text{ and } s = p.s\ \forall p \in Q \,\}$$

(a) Basic forking and merging.

$$\text{FORK}(T, p) := \{\, (H, p.s) \mid H \in \text{LAZY}(T, p) \cup \text{SHARED}(T, p) \,\}$$
$$\text{LAZY}(T, p) := \{\, \textstyle\bigcup \{(c, a)\} \mid \text{ACTION}(a, p.s) = \text{`}\texttt{shift}\text{'}\ \forall (c, a) \in T \,\}$$
$$\text{SHARED}(T, p) := \{\, \textstyle\bigcup \{(c, a)\} \mid \text{ACTION}(a, p.s) = \text{`}\texttt{reduce}\ n\text{'}\ \forall (c, a) \in T \,\}$$

(b) Optimized forking.

Figure 7. The definitions of fork and merge.

4.3 Forking and Merging

Figure 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* [39], which also combines equivalent subparsers, except that FMLR subparsers have presence conditions.

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 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 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 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 $\textsc{Fork}(H, p)$, 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 $c_i = \bigvee p.H(i).c$.

4.5 Putting It All Together

We are now ready to illustrate FMLR on the array initializer in Figure 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 6. We also use b_n to denote the boolean variable representing the conditional expression on line n, e.g.,

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

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

Since line 1 in Figure 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 $\textsc{Follow}$ for the conditional on lines 2–4. To this end, $\textsc{First}$ iterates over the conditionals and `NULL` token in the initializer list by updating a in Alg. 3:21. On each iteration besides the last, $\textsc{First}$ 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

$$\begin{aligned} T_2 = \{ & (b_2, \texttt{adfspart\_check\_ICS}),\\ & (\neg b_2 \wedge b_5, \texttt{adfspart\_check\_POWERTEC}),\\ & \ldots, (\neg b_2 \wedge \neg b_5 \wedge \neg b_8 \wedge \ldots, \texttt{NULL}) \} \end{aligned}$$

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

$$p_3.s = \ldots \texttt{ \{ } \textit{InitializerList}$$

It then calculates $\textsc{Fork}(H_3, p_3)$; 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

$$p'_3.s = \ldots \texttt{ \{ } \textit{InitializerList} \texttt{ adfspart\_check\_ICS ,}$$

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

$$\begin{aligned} T'_5 = \{ & (b_2 \wedge b_5, \texttt{adfspart\_check\_POWERTEC}),\\ & \ldots, (b_2 \wedge \neg b_5 \wedge \neg b_8 \wedge \ldots, \texttt{NULL}) \} \end{aligned}$$

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 $\textsc{Fork}$, 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!

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 [27] to generate the lexer and on Bison [17] 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 [35], 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 [22]. 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 5.1) and context management (Section 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.

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 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.

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.

6. Evaluation

To evaluate our work, we explore three questions. Section 6.1 examines how prevalent preprocessor usage is in real-world code. It measures preprocessor directives and feature interactions in the Linux kernel. Section 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 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, outrunning TypeChef by more than a factor of four and out-scaling it for complex compilation units.

	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 2. A developer's view of x86 Linux preprocessor usage.

6.1 Preprocessor Usage and Interactions

Table 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 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 2b shows the top five most frequently included headers; `module.h` alone is included in nearly half of all C files.

Table 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 2, including macro invocations. Table 3 loosely follows the organization of Table 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 3 confirms that preprocessor usage is extensive. It also confirms that most interactions identified in Section 2 occur in real-world C code.

Language Construct	Total	Interaction with Conditionals		Other Interactions	
Macro Definitions	34k · 45k · 122k	Contained in	34k · 45k · 122k	Redefinitions	23k · 33k · 111k
Macro Invocations	98k · 140k · 381k	Trimmed	16k · 21k · 70k	Nested invocations	64k · 97k · 258k
		Hoisted	154 · 292 · 876	Built-in macros	135
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	Computed includes	34 · 56 · 168
				Reincluded headers	1,185 · 1,743 · 5,488
Static Conditionals	8k · 10k · 29k	Hoisted	331 · 437 · 1,258	With non-boolean expressions	509 · 713 · 1,975
		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		

Table 3. A tool's view of x86 Linux preprocessor usage. Entries show percentiles across compilation units: 50th · 90th · 100th.

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.

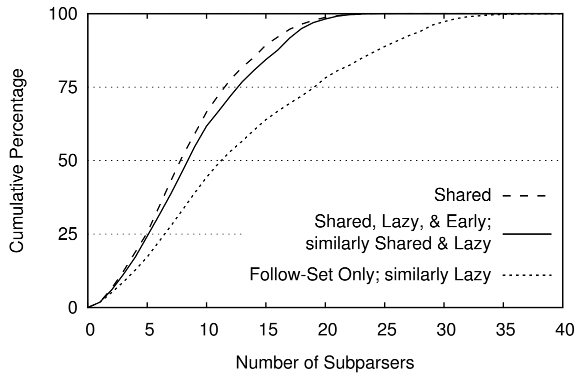
6.2 Subparser Counts

According to Table 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 8 shows the cumulative distribution of subparser counts per FMLR iteration for the x86 Linux kernel under different optimization levels: 8a identifies the maxima and 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 [33]. Figure 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.

	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 8. Subparser counts per main FMLR loop iteration.

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 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 Type-

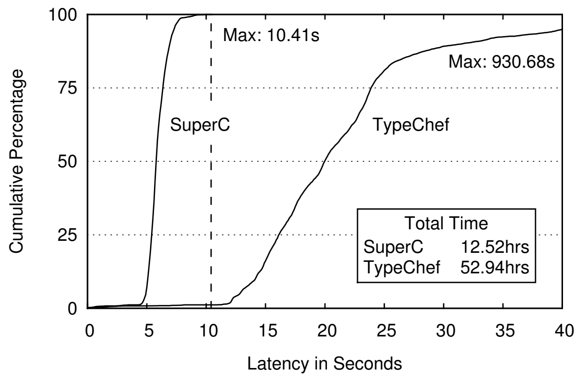


Figure 9. SuperC and TypeChef latency per compilation unit.

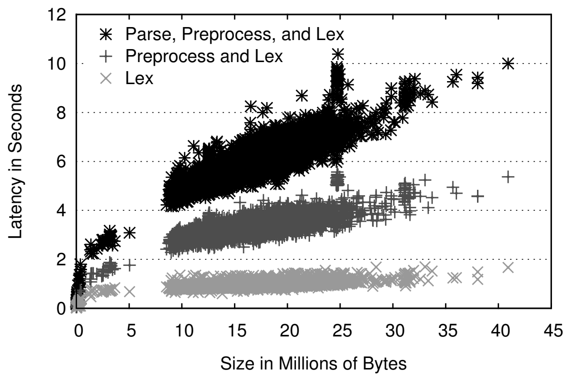


Figure 10. SuperC latency by compilation unit size.

Chef. Kästner et al. assembled it by analyzing Linux' configuration database [26]. (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 3 for preprocessor usage and in Figure 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 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 [25]; this representation is required by TypeChef's SAT solver, which TypeChef uses instead of BDDs.

Figure 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.

7. Related Work

Our work joins a good many attempts at solving the problem of parsing C with arbitrary preprocessor usage [1, 3–5, 15, 19, 25, 26, 29, 31, 33, 36, 41]. Out of these efforts, only MAPR [33] and TypeChef [25, 26] come close to solving the problem. Since we already provided a detailed comparison to MAPR and TypeChef in Sections 1, 2 and 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 [41], that process source code one configuration at a time, after full preprocessing. This approach is also taken by Apple's Xcode IDE [10]. 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 [19], that employ a fixed but incomplete algorithm. This approach is also taken by the Eclipse CDT IDE [21]. 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 [31], 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. [14], Tartler et al. [38], and this paper's Section 6.

Considering parsing more generally, our work is comparable to efforts that build on the basic parsing formalisms, i.e., LR [28], LL [34], and PEG [7, 16], and seek to improve expressiveness and/or performance. Notably, Elkhound [30] explores how to improve the performance of generalized LR (GLR) parsers by falling back on LALR for unambiguous productions. Both SDF2 [11, 40] and *Rats!* [22] 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 [32] 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 [23], *Rats!*, SableCC [18], 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 [35]. 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!*.

8. Conclusion

This paper 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*.

Acknowledgements

We thank Martin Hirzel, Christian Kästner, Julian Rosse, and the anonymous reviewers for their helpful feedback on earlier versions of this paper. We also thank Jinyang Li and Russell Power for letting us use their cluster for data collection. This work is supported by NSF CNS-0448349, CNS-0615129, and CCF-1017849.

References

[1] B. Adams et al. Can we refactor conditional compilation into aspects? In *Proc. 8th AOSD*, pp. 243–254, Mar. 2009.

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

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

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

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

[6] 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.

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

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

[9] B. Blanchet et al. A static analyzer for large safety-critical software. In *Proc. PLDI*, pp. 196–207, June 2003.

[10] R. Bowdidge. Performance trade-offs implementing refactoring support for Objective-C. In *Proc. 3rd WRT*, Oct. 2009.

[11] M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proc. 19th OOPSLA*, pp. 365–383, Oct. 2004.

[12] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *TOC*, C-35(8):677–691, Aug. 1986.

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

[14] M. D. Ernst et al. An empirical analysis of C preprocessor use. *TSE*, 28(12):1146–1170, Dec. 2002.

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

[16] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. 31st POPL*, pp. 111–122, Jan. 2004.

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

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

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

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

[21] E. Graf et al. Refactoring support for the C++ development tooling. In *Companion 22nd OOPSLA*, pp. 781–782, Oct. 2007.

[22] R. Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pp. 38–51, June 2006.

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

[24] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. In *Proc. 35th STOC*, pp. 355–364, June 2003.

[25] C. Kästner et al. Partial preprocessing C code for variability analysis. In *Proc. 5th VaMoS*, pp. 127–136, Jan. 2011.

[26] C. Kästner et al. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. 26th OOPSLA*, pp. 805–824, Oct. 2011.

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

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

[29] B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proc. 10th ESEC*, pp. 21–30, Sept. 2005.

[30] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. 13th CC*, vol. 2985 of *LNCS*, pp. 73–88, Mar. 2004.

[31] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. 18th CC*, vol. 5501 of *LNCS*, pp. 109–125, Mar. 2009.

[32] T. Parr and K. Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proc. PLDI*, pp. 425–436, June 2011.

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

[34] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proc. 1st STOC*, pp. 165–180, May 1969.

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

[36] D. Spinellis. Global analysis and transformations in preprocessed languages. *TSE*, 29(11):1019–1030, Nov. 2003.

[37] R. Tartler et al. Configuration coverage in the analysis of large-scale system software. *OSR*, 45(3):10–14, Dec. 2011.

[38] R. Tartler et al. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. 6th EuroSys*, pp. 47–60, Apr. 2011.

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

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

[41] M. Vittek. Refactoring browser with preprocessor. In *Proc. 7th CSMR*, pp. 101–110, Mar. 2003.

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