Characterization and Assessment of the Linux Configuration Complexity

Ahmad Jbara and Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

***Abstract*—The Linux kernel is configured for specific uses by manipulations of the source code during the compilation process. These manipulations are performed by the C pre-processor (CPP), based on in-line directives. Such directives, and the interleaving of multiple versions of the code that they allow, may cause difficulties in code comprehension. To better understand the effects of CPP, we perform a deep analysis of the configurability of the Linux kernel. We found significant inconsistencies between the source code and the configuration control system. Focusing on the thousands of config options appearing in the source code, we found that their distribution is heavy-tailed, with some options having more than a thousand instances in the code. Such wide use seems to imply a massive coupling between different parts of the system. However, we argue that employing a purely syntactic analysis is insufficient. By involving semantic considerations, we find that in reality the coupling induced by the very frequent options is limited. Moreover, even at the syntactic level the adverse effects of CPP are limited, as there is little nesting and the expressions controlling conditional compilation are usually very simple. But it could be even better if the configuration system undergoes a clean up. On the other hand, we found that the code controlled by CPP is very heterogeneous and may exhibit intimate mingling with non-variable code. As a result the applicability of alternative mechanisms such as aspects is hard to envision.**

I. Introduction

While much attention in the wider software engineering community is (rightfully) directed towards other aspects of systems development and evolution, such as specification, design, and requirements engineering, it is the source code that contains the only precise description of the behaviour of a system. Thus ensuring that the source code is comprehensible is of prime importance. An often overlooked aspect of code comprehension is the use of pre-processors, which support various manipulations of the source code during the compilation process. The injection of pre-processor directives, and the inclusion of multiple alternative versions of the code, may be expected to exacerbate the code comprehension problem. To see if this is indeed the case, we performed a detailed analysis of the configuration control mechanisms used in the Linux kernel. In a nutshell, our findings indicate that at least in this system the situation is not so bad and the code remains largely comprehensible. Nevertheless, the configuration system could use a massive cleanup.

CPP (the C pre-processor) is a commonly used tool for expressing software variability, whereby different executables are built from common source code. It works by using directives for conditional compilation, so that during the build process the compiler can decide whether or not to compile certain code fragments, or select from among multiple versions of the same basic functionality. The conditional compilation directives typically use preprocessor constant definitions, which may be derived from configuration option values that were set at an earlier stage.

Despite its popularity and strength, CPP has been identified as problematic. In particular, the interleaving of flow control and conditional compilation directives, as well as its lexical nature, may make the code harder to understand and maintain [16], [8], [4], [20], [15], [1]. Even the CPP reference manual identifies many pitfalls, especially when using it for macro definitions [17].

This situation has spurred recent interest in how CPP is used and in possible alternatives [13], [1], [9], [10], [19]. For example, Liebig et al. have analyzed the use of CPP in 40 large open-source projects, and provide various statistics characterizing its use [9]. While work such as this provides a wide picture of CPP usage, it might miss on the details. For example, not every symbolic constant that appears in a conditional directive is relevant to variability management. We therefore set out to complement previous work with an in-depth analysis of one of the projects, namely the Linux kernel. Linux makes heavy use of conditional compilations due to the need for special customization to support different architectures and features. The version we analyzed was Linux kernel 2.6.32.3.

To study variability we focus on configuration options, which are the means for expressing the select feature set that should be included in a specific build. Importantly, we consider the entire set of config options of all architectures. Our first finding is that it is not so easy to identify the relevant config options, as there are significant inconsistencies between the source code and the configuration control system. Thus one must be careful regarding the goal of the study: if it is the control of configurations and variability, the data might be different than if it is the effect of configurability on the complexity of the source code.

After selecting the data we want to focus on, we study the use of config options in the source code. We found a few thousands of options. These options have a skewed distribution such that the 20/80 rule holds. Thus some options, members in the significant 20%, are used more than a thousand times,

An abbreviated preliminary version of this work was presented in the PhD forum of ICPC 2012.

seeming to imply massive coupling between diverse parts of the kernel. However, such a conclusion is based on purely syntactic analysis. By adding semantic considerations, we found that the imposed coupling is in fact minor. Moreover, recall that a large number of options, the least significant 80%, are used only a few times, so their contribution to the code coupling is also limited.

Despite the relatively low coupling, CPP nevertheless does have adverse effects on the code. Our measurements showed that the average (geometric) scattering degree of config options is relatively high as well as the number of code variations within a file. Another issue is therefore the possibility of replacing CPP with other mechanisms. This is found to be problematic due to the intimate mingling of code snippets using CPP, and the heterogeneity of the variable code blocks, so mechanized alternatives such as aspects are questionable.

Section II describes the CPP tool, the Linux configuration process, and related work. Section III then lists our research questions. The bulk of our study is reported in Section IV, which characterizes config options, and Section V, which presents metrics for the complexity introduced by CPP. We discuss the results and summarize conclusions in section VI.

II. Background

A. The C Language Preprocessor

Preprocessing by CPP is the first phase in compiling C programs. CPP is a powerful tool for managing configuration and portability of software, and also provides useful features such as header inclusion and macro definition [17].

In our work the most important feature of CPP is the support for *conditional compilation*. This allows a single software base to be used to compile many variants. In each variant, some fragments of the original source code are included and others are excluded, based on conditional directives. The #ifdef directive includes the controlled code in the compilation if its argument is a defined CPP constant. The complementary #ifndef directive includes the fragment if the constant is *not* defined. The #if directive resembles the if statement of the C language: the controlled code is included iff the expression evaluates to non-zero. The expression may consist of CPP constant values, tests of whether they are defined, and logical operations. The #else and #elif directives can be used to provide alternatives to #if, #ifdef, and #ifndef.

When a CPP constant controls a configurable feature of the system we call it a *config option*. In the Linux system the config options, by convention, have a CONFIG_ prefix. When we talk about specific occurrences in the code we call them *config instance*s.

CPP constants are defined internally by an explicit use of the #define directive within the source code, or externally by flags to the compiler. For example, the *gcc* compiler uses the -D option to define a new CPP constant and the -U option to undefine it.

The Emacs tool [6] enables programmers to navigate the conditional directives, allowing them to view the code with its macros expanded, and to emulate the different configurations which are controlled by preprocessor variables.

```
1 menu "Processor type and features"
2 source "kernel/time/Kconfig"
3 config SMP
4  bool "Symmetric multi-processing support"
5  ---help---
6  This enables support for systems with more than one CPU.
```

Listing 1. Example of the definition of the SMP config option in a Kconfig file.

```
1 obj-$(CONFIG_GENERIC_ISA_DMA)+=dma.o
2 obj-$(CONFIG_USE_GENERIC_SMP_HELPERS)+=smp.o
3 ifneq ($(CONFIG_SMP),y)
4 obj-y += up.o
5 endif
6 obj-$(CONFIG_SMP) += spinlock.o
```

Listing 2. Examples of using config options in a makefile to control the set of files to compile.

B. The Linux Configuration Process

The Linux kernel may be configured to run on many different platforms and provide diverse sets of features by using config options [19]. We present the config options from two different points of view: developers, who define config options and integrate them into the code, and users, who use the config options to customize the kernel for their needs.

Developer View of Config Options. Initially, each config option is defined in the *Kconfig* system. The *Kconfig* system is a set of text files placed in kernel source directories where variability is needed. These files' names start with Kconfig. Their contents are definitions of config options. An example is shown in Listing 1.

After defining the config options, developers integrate them in the *Kbuild* system or in the source code files. The *Kbuild* system is a collection of makefiles which are responsible for building the system [3]. In makefiles, config options are used to control the compiling process; an example is shown in Listing 2. In source code the config options are used as constants in conditional compilation directives to control the inclusion or exclusion of code fragments. Examples are shown below in Listings 3 and 4.

As time passes and new versions of the system come into being some of the configuration options become redundant and should be removed from the system files. As we show below this is not always done.

User View of Config Options. In order to build the Linux kernel one must first specify the desired configuration. This is done by invoking the make tool with one of three variations: make config, make menuconfig, or make xconfig. Initially, the config tool reads the *Kconfig* system files to extract the menus, config options, and dependencies between the different config options. The extracted config options are presented to the user who is asked to select the desired options according to his needs. The difference between the three config tools is the user interface. The make config is a forward-only version

that enables configuration from scratch. menuconfig displays a menu and enables the user to selectively set the configuration, so there is no need to pass all the options one by one. The third version is GUI-based.

Once done, the config tool generates the .config file which is placed in the top directory of the kernel source. This file contains all the configuration options that were set by the user. It is a text file with a line for each option. Each line is a key-value pair with the format CONFIG_*key=value*. The *key* is the name of the option, and *value* is y to indicate that a component will be built into the kernel, or m to indicate that it will be built as a loadable module [2]. Options that were not set are commented-out with # or deleted from the file.

The next stage is building the Kernel with the make command. At the very beginning of the build process the autoconf.h file is created. This file contains CPP definitions for the config options that were included in the .config file. To make these definitions available during compilation of the source code files the compiler uses its -include option. A few source code files explicitly use the inclusion mechanism of the preprocessor.

To shorten the configuration process and make it efficient the kernel source is provided with a preset configuration for each of the architectures that the kernel supports. These default configurations are kept in the arch/* subdirectories, and are called defconfig*X*, where *X* typically indicates the architecture and the developer who created the file. To utilize these defaults one should rename the default configuration file to be .config prior to the configuration process and place the new .config file in the top level directory, or pass the name of the default configuration file as an argument to the make tool.

C. Related Work

The use of CPP has attracted significant interest in recent years, Mostly related to configurability and the creation of product lines. We are specifically concerned with the resulting complexity and cognitive load on developers.

Early work also considered the ill-effects of using CPP. Spencer and Collyer [16] claim that careless use of #ifdefs is usually considered a mistake. They presented alternatives to conditional compilations, applied to their C News Package, and reduce the use of CPP by using clean interfaces and information hiding. Krone and Snelting [8] also claim that the use of conditional directives makes the code hard to understand even for experienced programmers. They suggested a visual tool which infers the configuration structure of source code, and makes it easy to discover violations of software engineering principles such as high cohesion and low coupling.

Favre [4] stated that heavy use of CPP directives can lead to unreadable programs, and makes maintenance and tool building hard. Vidács and Beszédes [20] also believe that heavy use of preprocessor directives causes problems of code comprehension due to the gap between what the programmer sees and what the compiler gets. They suggested using a tool, CANPP, for producing preprocessor schemas that can be used for information extraction such as original source, preprocessed files, and intermediate states.

The impaired readability and reduced reusability of conditional directives were also the motivation for developing C-CLR [15]. They identified the macro conditional redefinition and composition of multiple configuration options as problematic. The C-CLR tool improves readability by enabling users to perform configuration-specific navigation and enables reusability by automated identification of equivalent blocks.

Sutton and Maletic [18] presented a common configuration architecture for managing portability among three packages which were examined in their study. They also introduce configuration management patterns which they observed, including naming conventions, replaceable and parameterized inclusion, and compiler abstractions.

Based on [16] the authors of ASTEC [12] claim that macros are difficult to analyze and are error-prone. They present an alternative language which eliminates many of the potential errors. This is a syntactic language, as opposed to CPP which is purely lexical. It preserves the configurations of the program for analysis tools, while CPP produces a one-configuration program. Also, it enables tools to check errors before macros are expanded.

Liebig et al. [9], [10] studied the configurability of 40 open-source projects, including Linux. They found that 23% of the code is variable, there is no correlation between a system's size and the complexity of variable code, variable code is mostly heterogeneous which makes the use of AOP inapplicable, and the #ifdefs are mostly used in a high level granularity, enclosing entire entities such as functions and control statements.

Linux is the third-largest software system analyzed by Liebig et al. [9], both in terms of lines of code and in terms of the number of CPP constants (the two bigger ones are also operating systems: OpenSolaris and FreeBSD). Reynolds et al. [13] also studied Linux, focusing on the config options of one architecture (*i386*). In contrast, we consider the entire set of config options of all architectures. Tartler et al. [19] studied the consistency of using configuration options in Linux using an automated tool, and found 147 confirmed bugs.

Czarnecki et al. also studied the configuration system of Linux and transformed it to a feature model for benchmarking purposes [14]. They examined only the Kconfig files of one specific architecture (*x86*). Diettrich et al. suggested an approach for extracting implementation variability from the Linux build system, as it contains more than 65% of all config options [3]. In contrast, we examine *all* configuration contexts (source code, default configuration, makefiles, and Kconfig) of the whole system (all architectures and subsystems), with the goal of evaluating the impact of CPP on comprehensibility. Our study thus includes many observations not made previously by others.

III. Research Questions

Our ultimate goal is to gain some insights about the increased complexity of the code that results from configuration

variability management with CPP. Considering two of the studies cited above raises some important methodological questions. The work of Liebig et al. [9], [10] consists of a wide survey of 40 large-scale open-source projects, comprising 30 million lines of code. By necessity, such a study is based on automated tools and a high-level view of average metric values. Tartler et al. [19] focus on only one system, namely Linux, and show that this system suffers from bugs resulting from inconsistent use of configuration options. This leads us to the following specific questions:

1) Can the *relevant* configuration options indeed be identified by straightforward lexical means?
2) Are all config options equally important, and are average values generally representative?
3) Can the effect of config options be evaluated by syntactic analysis alone?
4) Do config options affect the whole codebase in a uniform manner?
5) Does conditional compilation based on config options introduce significant complexity to the code?

Thus we wish to take the reservations of Tartler et al. into account, and perform a more detailed study than that performed by Liebig et al. This is enabled by focusing on a single system, and considering complete distributions rather than averages.

IV. Characterization of the Linux Kernel Configuration

We are specifically interested in variability that is part of the system's design, namely explicit support for different configurations. Studying such variability in a system like Linux has two possible points of departure: either collect information on all the CPP constants used to control conditional compilation, or else start with all the known configuration options. As we show below, neither is completely satisfactory: there are many CPP constants that do not reflect real variability, and there are many config options that do not appear in the code. We therefore actually want the intersection of the two. But even this is not enough, as the code may include definitions of derived CPP constants that depend on other config options.

A. Source Configurability

As noted above, configurability is implemented in the source code using conditional compilation. To study its effect on the code we must first identify the CPP constants that are part of the configuration control mechanism. We therefore started by extracting all the CPP constants from all the #ifdef expressions[1] in the Linux source.

We found 10,988 different constants that appear in #ifdefs, of which 4731 start with CONFIG_ and may therefore represent config options (but some are false positives, as we show below). In this we use Linux-specific knowledge, and depart from Liebig et al. who considered all CPP constants. We claim this is important for reliable results concerning code variability, and thus that variability cannot be studied using only syntactic means. For example, 4330 of the non-config constants are defined by #define directives in the code itself, which means that they typically do not contribute to variability and always provide the same path.

[1]Here and in the sequel we use #ifdef to also mean #ifndef, #if, and #elif.

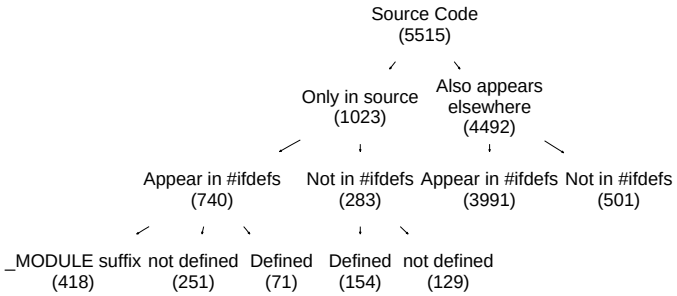

Fig. 1. Distribution of CONFIG_ options in the source code.

However, we found that some of these defined constants actually depend on configuration options in one of two ways: they are either defined as an alias of a config option, or else their very definition occurs within a code fragment that is only compiled conditioned on a config option. This means that their definition is *derived* from the config options. We found 23 thousand such derived definitions, but only 1009 of them are relevant because they are subsequently used in #ifdefs. This means that in total there are actually 5740 constants that may reflect configurability, and not 4731 as a merely syntactic analysis suggests.

Another large group of constants is due to the CPP inclusion mechanism: it is customary to avoid double-inclusion of header files by protecting such files with an #ifdef based on a constant defined in the same file. We ignore these constants and #ifdefs as they are idiomatic and do not reflect variability. The rest of the constants are not of real interest. Many of them deal with debug issues, while others are not defined at all (even not in makefiles).

B. Inconsistent Use of Config Options

As we mentioned above Linux configuration options may appear in four different contexts. The config options are initially born in the *Kconfig* files, the settings in the *defconfig* files are derived from the Kconfig files, and the options in *source code* files or *makefiles* are a subset of the total config options. In an ideal world all these sources of the config options would be synchronized. In practice, they are not.

We identified potential config options as follows. In the Kconfig files, they are introduced by the config keyword (see Listing 1). In the other files they are string constants that start with CONFIG_. When we checked all the header and implementation files (C and assembly) of the source code we found 5,515 such string constants (Figure 1). In the Kconfig system files we found 9,342 options, and in the defconfig files we found 8,696 options. Finally, we got 6,325 config options in makefiles. Altogether we have 11,303 unique config options from all these sources. These results and the logical relations between them are presented in Figure 2.

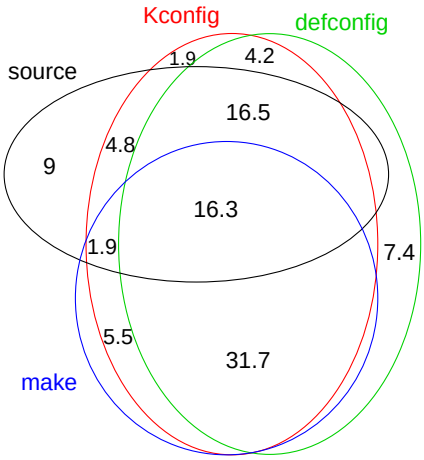

Fig. 2. The overlap relations between the different sources of the config options clearly indicate they are inconsistent. Values are percent of the total number; values less than 1% not shown.

Obviously, just less than half of the options we have found are used in the source code. This is not unreasonable because an additional 37.5% are used in makefiles to control which source files are included in the build process. Strangely, 6.1% of the options in the Kconfig system do not appear at all in the source or the makefiles. These options are therefore effectively no-ops, with no effect on the configuration. Stranger yet, 7.4% appear only in defconfig files. This contradicts the assumption that the defconfig files are derived from the Kconfig files, and may reflect legacy options in architectures that are not well-supported any more.

In the context of our interest in code quality, the biggest anomaly in Figure 2 is that around 9% of the total config options occur only in the source code and not in any of the configuration files as expected. Further examination of the source code helped to explain this and revealed several different sub-categories of options (left branch of Figure 1).

First, we found that more than 40% of these options have a _MODULE suffix. These are derived from options that were defined in the Kconfig files without this suffix. The suffix is appended during the build process, when creating the autoconf.h file, if the user configured an option to be compiled as a module. So, for a given option *X* in the Kconfig system, we may see both CONFIG_*X* and CONFIG_*X*_MODULE in the source code.

Second, we found that 22% of these options are defined by the #define directive in the source code, but not in the configuration process. In other words, these CPP constants are false positives that do not reflect true configurability (unless they are derived from true config options).

Next, around 25% of these options seem not to be defined at all, although they are in fact used by conditional compilation directives. We believe that most such options are leftovers from previous versions that should have been removed. A small number may be bugs, where a config option name was misspelled [19].

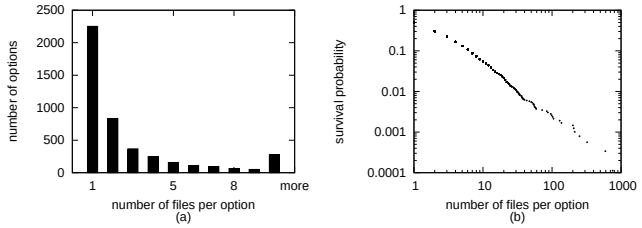

Fig. 3. a) Distribution of config option number of files. b) LLCD plot showing the distribution has a heavy tail.

Finally, most of the remaining CONFIG_ strings were defined and used at the C language level with no relation to the preprocessor at all. For example, elements of an enum data type might have names that start with CONFIG_. These have nothing to do with our research and we ignore them.

The conclusion from all the above is that the configuration process settings are not expressed cleanly in kernel code which is loaded with redundant options as well as misleading naming and usages. Such behavior soils the code and as a result makes the work of developers harder. On the basis of this conclusion we gain a significant insight: *A blind syntactic analysis of a large software system may miss its goal*. Some of the previous analyses of Linux may have had this problem. Our results corroborate and extend those of Tartler et al. [19] who specifically study the inconsistency of using config options.

Our goal is to eventually study the effect of conditional compilation directives based on config options on code comprehension. We will therefore focus on the real options that have a real influence on configurability. These options are those that occur in the source code #ifdefs as well as in the Kconfig files. To this set we will add the derived constants which were presented earlier. There are 4,440 options that are shared by Kconfig and the source code, out of which 3,941 are used in ifdefs. When adding to the 1,009 derived constants that also appear in #ifdefs we get 4,950 config options in total.

Note that config options with the _MODULE suffix are not included because each time they are used in the code their counterpart options, those without _MODULE, are also used in the same expression. Thus they do not add any independent configurability. Config options that are used only in makefiles are also not included, because they do not affect the difficulty of comprehending the code. From now on whenever we refer to config options we mean the set of real configuration options as it was defined here.

C. Heavy-Tailed Distribution of Config Options

Figure 3(a) shows how the configuration options distribute over the files of the source code. More than 2,700 options are used in one file only; these are the more specific options. On the other hand 285 options are used in more than 10 files; these are the cross-cutting options. This latter category of options may cause difficulties in maintenance because the developer must potentially think about many files when processing a single configuration option. In the most extreme case, we

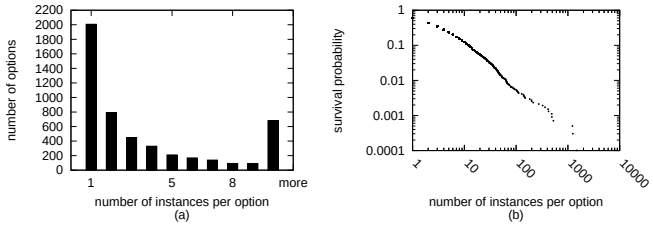

Fig. 4. a) Distribution of config option number of instances. b) LLCD plot showing the distribution has a heavy tail.

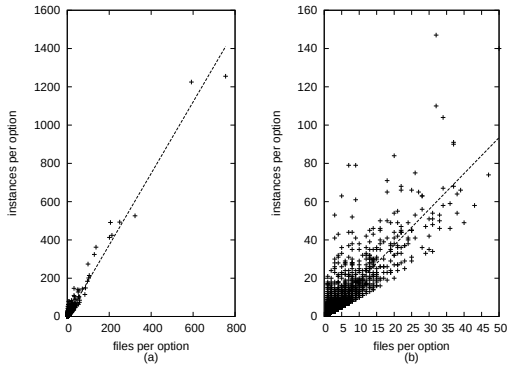

Fig. 5. a) Correlation between config options instances and the number of files they occur in. b) zoom in on the cluttered area.

found that the most frequent option is used in 753 different files.

The histogram in Figure 3(a) indicates that the distribution of config options across files is highly skewed. We therefore checked whether this is a heavy-tailed distribution. To do so we use an LLCD plot, as shown in Figure 3(b). This shows that the survival function of the distribution is approximately linear in log-log axes, which means that it decays according to a power law and is indeed heavy-tailed. The tail index, which is given by the slope of this line, is approximately 1.3. In heavy-tailed distributions the tail index ranges between 0 and 2, and smaller tail indexes indicate a heavier tail.

Next we considered the number of instances of each config option (note that a config option can have several instances in the same file). The results are illustrated in Figure 4(a). As may be expected this also shows that we have many config options with a small number of instances and few config options with a large number of instances. And again, the distribution of the config options in terms of instances is found to be heavy-tailed as illustrated in Figure 4(b) using an LLCD plot.

To relate these two distributions, we found that the number of files and the number of instances of a config option are correlated with a correlation coefficient of 0.97. It is easy to see this in Figure 5. In addition, the same config options appear at the top in both cases, and almost in the same order. These highly-used config options are listed in Tables I and II. They can be classified into three main groups:

- Related to cross cutting operating system features, such as power management (PM) or SMP support.
- Related to a specific operating system feature, such as the /proc file system.
- Related to a specific architecture or device, such as Intel X86, IBM PPC, or the PCI bus.

Option	Files
PM	753
SMP	591
DEBUG	591
PROC_FS	322
PCI	250
COMPAT	213
64BIT	206
MMU	201
X86_64	137
X86_32	129
BITS_PER_LONG	129
NET_POLL_CONT.	105
SYSCTL	102

TABLE I
CONFIG OPTIONS THAT APPEARED IN THE MOST FILES.

Option	Instances
DEBUG	1498
PM	1246
SMP	1221
PROC_FS	525
PCI	488
64BIT	490
COMPAT	424
MMU	413
X86_64	362
X86_32	324
PPC64	274
BITS_PER_LONG	200
NET_POLL_CONT.	214

TABLE II
CONFIG OPTIONS THAT HAD THE MOST INSTANCES.

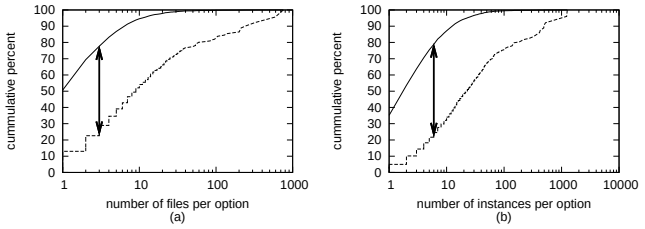

Fig. 6. The ”count” (upper) and ”mass” (lower) distributions of option occurrence. a) Mass-count disparity plot of number of options per file. b) Mass-count disparity plot of number of instances per option.

Skewed distributions may also be characterized by their mass-count disparity [5]. In our context this means that a small number of configuration options represent the majority of the files and instances, while at the same time most of the config options together account for only a small fraction of the files and instances. Such a phenomenon, when it exists, helps to focus on those items that have the most impact.

In Figure 6 we show the disparity between the mass (files or instances) and count distributions. In both cases the joint ratio is close to the 20/80 rule, which means that around 20% of the configuration options are responsible (present in) for 80% of the different files and instances, while the other 80% of the options lead to only 20% of the files and instances.

The same results and insights were obtained when we investigated the concentration of config options in different files. We counted the number of different configuration options that appear in each file as well as the number of instances. The results show that 4,169 files contain only one option while 237 files contain at least 10 different options. When examining the instances of the options we get that 2,496 files contain only one instance, and 858 files contain at least 10 instances. In both cases we found that the distributions are heavy-tailed, with tail indices of 1.4 and 1.7 respectively. The files in the tail are hot-spots of configurability.

D. Syntactic vs. Semantic Analysis

Our work so far, and previous work as well (e.g. [9], [13]), was performed at the syntactic level. However, we

```
1  static netdev_tx_t eexp_xmit(struct sk_buff *buf, struct
       net_device *dev)
2  {
3    short length = buf->len;
4  #ifdef CONFIG_SMP
5    struct net_local *lp = netdev_priv(dev);
6    unsigned long flags;
7  #endif
8  #if NET_DEBUG > 6
9   printk(KERN_DEBUG "%s: eexp_xmit()\n", dev->name);
10 #endif
11   if (buf->len < ETH_ZLEN) {
12     if (skb_padto(buf, ETH_ZLEN))
13       return NETDEV_TX_OK;
14     length = ETH_ZLEN;
15  }
16  disable_irq(dev->irq);
17 #ifdef CONFIG_SMP
18       spin_lock_irqsave(&lp->lock, flags);
19 #endif
20 {
21    unsigned short *data = (unsigned short *)buf->data;
22    dev->stats.tx_bytes += length;
23    eexp_hw_tx_pio(dev,data,length);
24 }
25    dev_kfree_skb(buf);
26 #ifdef CONFIG_SMP
27    spin_unlock_irqrestore(&lp->lock, flags);
28 #endif
29    enable_irq(dev->irq);
30    return NETDEV_TX_OK;
31 }
```

Listing 3. The blocks of the CONFIG_SMP option are orthogonal to the rest of the code in the function (drivers/net/eexpress.c).

claim that syntactic measures are not sufficient. By looking more closely at the conditional blocks, one may reveal semantic relationships between blocks that are governed by configuration options and other free blocks. Such interactions, may have a significant impact on the feasibility of applying aspects technology to reduce tangled code when handling cross-cutting concerns [7]. Indeed, Adams et al. [1] have shown that conditional compilation can be partially refactored into aspects.

To be more concrete we present code snippets to show that semantic involvement is required when analyzing code in the context of config options. We focus on a few config options and investigate them in different semantic contexts.

Orthogonality versus Coupling. We start with two basic cases: optional blocks that are orthogonal to the surrounding functionality and optional blocks that have a tight connection with the rest of the code, namely coupled code. The key point here is that the two behaviors occur for the same config option. This means that a single config option, which is considered a concern in AOP terminology, behaves differently regarding its connectivity to other concerns where it appears.

Listing 3 shows a function that has three blocks that are controlled by the CONFIG_SMP option. In line 20 the function acquires the lock and disables interrupts on the local processor while saving the state of the interrupts in flags. The

```
1  static int show_stat(struct kmem_cache *s, char *buf,
       enum stat_item si)
2  {
3   unsigned long sum = 0;
4   int cpu;
5   int len;
6   int *data=kmalloc(nr_cpu_ids*sizeof(int),GFP_KERNEL);
7   if (!data)
8               return -ENOMEM;
9   for_each_online_cpu(cpu) {
10   unsigned x = get_cpu_slab(s, cpu)->stat[si];
11   data[cpu] = x;
12                   sum += x;
13  }
14  len = sprintf(buf, "%lu", sum);
15 #ifdef CONFIG_SMP
16  for_each_online_cpu(cpu) {
17   if (data[cpu]&&len<PAGE_SIZE-20)
18    len += sprintf(buf + len, " C%d=%u", cpu, data[cpu]);
19  }
20 #endif
21  kfree(data);
22  return len + sprintf(buf + len, "\n");
23 }
```

Listing 4. The blocks of the CONFIG_SMP option are coupled with the rest code of the function (mm/slub.c).

corresponding unlock is called in line 30. Lines 5 and 6 define two variables which are needed only for the sake of these locking/unlocking operations. This is a classical case where aspects would be feasible especially with the knowledge that this function's aim is to transmit a packet and this concern is orthogonal to the CONFIG_SMP one. To gain such an insight one should delve into the code.

However, when examining Listing 4 the local variable len is defined and initialized outside the CONFIG_SMP block. Then, it is used twice and changed once within the CONFIG_SMP and finally used again outside this block. Moreover, the pointer buf is changed three times: before, within, and after the CONFIG_SMP block. One more point to mention is that buf is a parameter which is passed by address, so any change in its value will be returned back to the caller function. These tight dependencies between the function body, the config option block, and the caller function imply a semantic connection and logical dependency to the other parts of the function. Thus it is hard to extract out the block of the CONFIG_SMP automatically.

This function is even a bit more complex than was described. Line 12 calls a function with a body which is controlled by CONFIG_SMP. Also, lines 11 and 21 call the same macro where in line 11 the macro is free while in line 21 it is controlled by the #ifdef. This macro, in the file where it is defined, is controlled by CONFIG_SMP. This means that in fact lines 11–16 are controlled by the CONFIG_SMP option but this is not visible.

Do Many Instances Imply Coupling? The CONFIG_SMP option appears in 591 different files with 1,221 instances within those files. Seemingly, it is a hard case of a cross-cutting concern due to its wide scattering and the potential

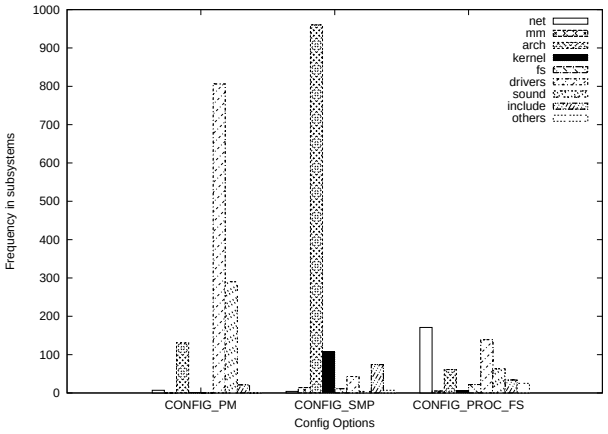

Fig. 7. Distribution of top config options over different subsystems.

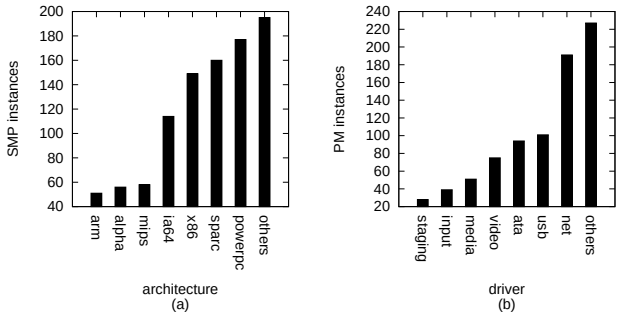

Fig. 8. a) Distribution of the CONFIG_SMP over architectures. b) Distribution of the CONFIG_PM over drivers.

coupling that this may imply. We argue that it is not as bad as it sounds.

The Linux Kernel is composed of subsystems residing in distinct subdirectories. These subsystems include, among others, the initialization code (init), memory management (mm), file systems (fs), and support for diverse architectures and devices (arch and drivers). Each subsystem (directory) is, to a large extent, an independent module. This means that despite the fact that the same config option may appear in multiple subsystems, there is not much coupling between the subsystems and one can treat each subsystem separately.

Other config options are logically related to specific subsystems, so they are largely localized in those subsystems. Thus even if they appear in other subsystems, they create only slight coupling with them. To demonstrate this, we looked at 3 of the most frequent options: CONFIG_PM, CONFIG_SMP, and CONFIG_PROC_FS. Figure 7 shows how these options distribute over the different subsystems of the Linux Kernel. CONFIG_PM occurs significantly in the drivers and sound modules while in the other modules it does not. Similarly, almost all the instances of CONFIG_SMP occur in the arch directory. When looking at the CONFIG_PROC_FS, about 60% of its instances occur in the net and drivers subsystems.

Actually, these results are not surprising. The designation of the *SMP* option is to enable *symmetric multi-processing support*. So, one expects a heavy use of this option in the directory where all the architectures are defined. In the case of CONFIG_PM most instances reside in the drivers directory due to the fact that power management is accomplished on the system components and peripherals. These examples again demonstrate the fact that syntactic analysis should be complemented by its semantic counterpart.

Moreover, Figure 8(a) shows how CONFIG_SMP distributes over the different architectures, and Figure 8(b) shows how CONFIG_PM distributes over the different drivers. Ultimately, one of these orthogonal architectures as well as a combination of the appropriate orthogonal drivers are built for each variant of the system. This means that the real number of instances of a specific config option that a developer needs to tackle at the same time is not the total number, but only those related to the specific architecture or driver he works on.

Due to the modularity of the Linux system and these insights we argue that the coupling between the different entities in the context of each config option is minimal.

V. Code Metrics

To get insights about the complexity resulting from CPP and examine the feasibility of alternative techniques we studied various code metrics. Here we focus on the dominant subset of the configuration options. These are the most frequent config options, which constitute approximately 20% of the different config options and account for approximately 80% of the instances.

Metric measurements with a normal distribution are well described by the arithmetic mean and the standard deviation. But when metrics have a skewed distribution, commonly with a low mean and a large variance, it is more appropriate to use the geometric mean and the multiplicative standard deviation [11]. The geometric mean, denoted by $\bar{X}^*$, is defined as the nth root of the product of n positive numbers: $\bar{X}^* = e^{(\frac{1}{n}\sum_{i=1}^{n} logX_i)}$. The formula is presented this way to allow computation and avoid overflows when the product is large. In contrast to the arithmetic mean, which is dominated by the large values, the geometric mean responds equally to changes in large and small values. The multiplicative standard deviation, denoted by $S(X)^*$, is essentially the average of the quotients of the samples and the mean: $S(X)^* = e^{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\log \frac{X_i}{\bar{X}^*})^2}}$. On the basis of these two descriptive values, the distribution is characterized by the range $\bar{X}^* \ ^{\times}\!/ \ S(X)^*$ (that is, from $\bar{X}^*/S(X)^*$ to $\bar{X}^* \times S(X)^*$).

In using full distributions and noting their skewed nature we again depart from previous practices. Looking at the data of [9] we see that in practically all cases the standard deviation is (much) larger than the mean. This indicates that the distributions are skewed, and that the arithmetic mean and standard deviation do not provide a good summary of the data.

The metrics we checked are the following.

Scattering Degree of Config Options. The scattering degree metric, denoted by SD, was defined by [9] as the number of instances of CPP constants in different expressions. This

metric measures the spread of configuration options, and high values are expected to reflect difficulty in comprehending the code and managing the configurations. In practice they actually used the average number of instances per config option. We have shown the distribution in Figure 4. Focusing on the dominant options, we found that the geometric mean of the scattering degree is 15.2, with multiplicative standard deviation of 3. This means that the bulk of the scattering degrees of the dominant config options are covered by the interval 5 to 45.6.

Tangling Degree of Config Options. The tangling degree metric, denoted by TD, was defined by [9] as the number of different CPP constants that occur in an expression. This metric measures the mixing of different configuration options within the expressions of #ifdefs. To evaluate this metric we measure the occurrences of the dominant configuration options in the CPP expressions. We found that the geometric mean is 1.035 with deviation of 1.185. This means that usually we have only one config option and the expressions are quite simple.

Conditional Blocks. The code blocks which are controlled by #ifdefs are classified in the literature as homogeneous or heterogeneous. This is important because alternatives such as aspects are only applicable to homogeneous blocks. To compare the blocks in different #ifdefs we squeeze whitespaces and concatenate the lines to one string and then make the comparison. We found that 92% of the conditional blocks of the config options of the dominant subset are heterogeneous. In particular, the options CONFIG_PM and CONFIG_SMP each have more than one thousand heterogeneous blocks.

As such comparisons are very stringent we also made some manual checks. We manually looked at one hundred blocks of the CONFIG_PM option and got the same results. Moreover, we expected that the automatic comparison will have problems with long code blocks. Our manual work revealed that in fact the automatic comparisons identified identical blocks of 10 lines. This probably indicates that a copy-paste mechanism was used by the developers wherever the same functionality was needed. But we cannot infer that this is the general case and more work should be done here.

Complexity of Expressions. We found 68,524 lines of conditional statements in CPP directives. More than 36% of them reference at least one configuration option of the dominant set. We examined these expressions and counted the number of logical operators in each of them. The results are presented in Figure 9(b). We found that about 79.2% use #ifdef or #ifndef, and therefore do not really have any expression. About 18.3% use the #if construct, and the #elif construct captures the remaining 2.3%.

In the #if and #elif expressions, we found that about 53.3% were compound with an average of 1.2 logical operators. The or operator dominates with more than 72% of the operator instances, the and operators captures about 25%, and the not operator is almost non-existent. However, it should be noted that about half of the or occurrences have operands with a _MODULE suffix. This observation shows that the use of or partially follows a simple predefined pattern that is supposed to reduce its complexity.

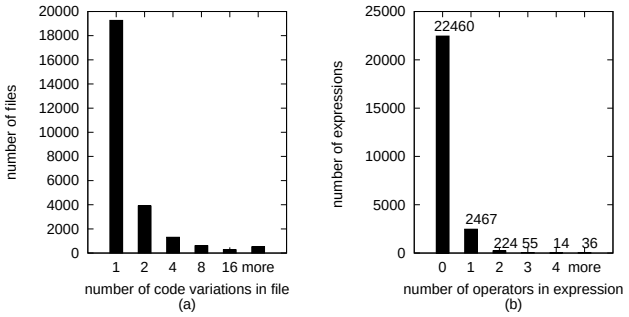

Fig. 9. a) Distribution of code variation in source files. b) Complexity of the expressions in #ifs.

Code Variations. When looking at the code base, the developer must consider all the different combinations in which the code may be built in the future. These combinations are concatenations of code segments that are created on the basis of the evaluations of the preprocessor conditionals. It is obvious that the number of the potential variations has a significant impact on the code understanding. To get insights about the code understanding complexity we counted the number of code variations for each of the source code files. As a first step we only took into account basic conditionals (#ifdef, #ifndef) that contain one configuration option. We have noted earlier that #ifdef and #ifndef constitute more than 79% of the conditionals of interest, so they may be expected to reflect the full picture. The results are shown in Figure 9(a). While obviously most files have few variations, some have an extremely large number of variations, with a maximal value of 316,659,348,799,488.

Nesting. One more aspect to look at is the nesting of conditionals. Initially we checked the nesting of general conditionals that are not related only to configuration options. Figure 10(a) describes the results. The average depth of conditional statements is 1.08. Afterward, we measured the depth of nesting for conditionals which contain at least one configuration option. The results are presented in Figure 10(b). The average depth of conditional statement in the configuration context increased slightly to 1.10, which is still low.

The metrics presented here provide evidence for the existence of many config options scattered around the code. These are bad news due to potential coupling that such characteristics may impose on the code. The good news are that these scattered config options are used in a simple manner. This stems from the low value of the tangling metric and the simple structure (low number of logical operators) of the expressions of the conditional constructs. Moreover, the intensive use of the simple form of the conditional construct as opposed to its composite form is additional evidence for simplicity.

VI. Discussion and Conclusions

The CPP has been identified over the years as a potentially harmful tool, in particular when it is used to implement variability in a large scale system like Linux. In this study we performed a detailed characterization of the Linux kernel

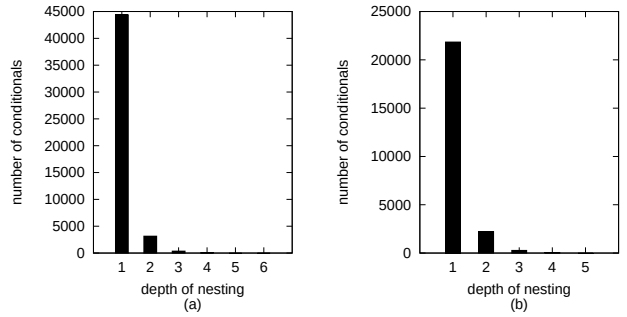

Fig. 10. Depth of the nesting of conditional constructs. a) General Conditionals. b) Conditionals with at least one config option. .

configuration as done using CPP, and presented different metrics in order to better understand CPP's usage and effect.

The approach taken in most previous work was breadth-first, and provided a survey of CPP usage in many systems. Such an approach necessarily comes at the expense of deeper analysis of each system. We therefore complemented this by using a depth-first approach on one particular system, namely the Linux kernel. This allowed us to employ domain-specific knowledge and semantic analysis, and led to various insights that would not be possible in a more general study.

In order to study the variability built into the system, we focus on Linux's configuration options. Our first finding is that it is not trivial to identify the effect of config options on the code, and that their usage is inconsistent. For example, we found more than one thousand options that appear only in the source code, and more than a thousand CPP constants that are derived from config options even though they are not themselves direct config options.

We found that the distribution of the config options is skewed (even heavy-tailed), and manifests the mass-count disparity phenomenon. These realizations are useful because they indicate the existence of a relatively small but prominent group of cross-cutting options. This helped us to focus on this subset when assessing their impact on the code. The skewness of the distribution also indicates that one should be careful to use the right descriptive metrics: geometric mean and multiplicative standard deviation.

Overall we found nearly 5000 real config options. This is worrying because it suggests extensive use of the CPP across the code, with adverse consequences for code comprehension. However, a large portion of these options have only a few instances, so their effect is actually very localized. The small fraction of the options that have many instances appear to create only slight coupling, due to the modularity of the system. The logical expressions used to control conditional compilation are typically very simple, and there is very little nesting. All these findings indicate that variability management with CPP does not cause excessive code degradation. However, the blocks that are controlled by these options were classified as heterogeneous which is bad due to the difficulty of using alternative techniques. Moreover, the code is soiled with garbage and misleading config options so an initiation of a clean up process is vital.

Our work suffers from several threats to validity. The complexity imposed by CPP usage can be measured using additional metrics, such as McCabe's cyclomatic complexity. Another issue is whether to focus on the most prominent config options, or to always consider all of them. Finally, we do not know whether these characterizations are true for other systems except Linux. Nevertheless, we think that this case study is important in its own right even if specific findings do not generalize. We think that it is valuable to characterize more systems in the domain of operating systems and then move to other domains. It is also interesting to study the evolution of CPP usage across the many versions of the Linux kernel itself.

REFERENCES

[1] B. Adams, H. Tromp, W. D. Meuter, and A. E. Hassan., "*Can we refactor conditional compilation into aspects?*" In 8th *Intl. Conf. Aspect-Oriented Softw. Dev.*, pp. 243–254, 2009.

[2] J.-M. de Goyeneche and E. A. F. de Sousa, "*Loadable kernel modules*". *IEEE Softw.* **16(1)**, pp. 65–71, Jan/Feb 1999.

[3] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "*A robust approach for variability extraction from the Linux build system*". In 16th *Proc. Intl. Software Product Line Conf.*, 2012.

[4] J.-M. Favre, "*The CPP paradox*". In 9th *European Workshop on Softw. Maintenance*, 1995.

[5] D. G. Feitelson, "*Metrics for mass-count disparity*". In 14th *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 61–68, Sep 2006.

[6] "*The GNU project emacs homepage*". www.gnu.org/software/emacs/emacs.html.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "*Aspect-oriented programming*". In 11th *European Conf. Object-Oriented Prog.*, pp. 220–242, 1997.

[8] M. Krone and G. Snelting, "*On the inference of configuration structures from source code*". In 16th *Intl. Conf. Softw. Eng.*, pp. 49–57, 1994.

[9] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "*An analysis of the variability in forty processor-based software product lines*". In 32nd *Intl. Conf. Softw. Eng.*, pp. 105–114, 2010.

[10] J. Liebig, C. Kästner, and S. Apel, "*Analyzing the discipline of preprocessor annotations in 30 million lines of C code*". In *Intl. Conf. Aspect-Oriented Softw. Dev.*, pp. 191–202, Mar 2011.

[11] E. Limpert, W. A. Stahel, and M. Abbt, "*Log-normal distributions across the sciences: Keys and clues*". *BioScience* **51(5)**, pp. 341–352, May 2001.

[12] B. McCloskey and E. Brewer, "*ASTEC: A new approach to refactoring C*". *SIGSOFT Software Engineering Notes* 2005.

[13] A. Reynolds, M. E. Fiuczynski, and R. Grimm, "*On the feasibility of an AOSD approach to Linux kernel e xtensions*". In *Proc AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Softw.*, 2008.

[14] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "*The variability model of the Linux kernel*". In 4th *Workshop on Variability Modeling of Software-Intensive Systems*, 2010.

[15] N. Singh, C. Gibbs, and Y. Coady, "*C-CLR: A tool for navigating highly configurable system software*". In 6th *Workshop on Aspects, Components, and Patterns for Infrastructure Softw.*, 2008.

[16] H. Spencer and G. Collyer, "*#ifdef considered harmful, or portability experience with C news*". In *Proc. USENIX Technical Conf.*, 1992.

[17] R. M. Stallman and Z. Weinberg, "*The C preprocessor*". GNU project, Free software foundation, 2010.

[18] A. Sutton and J. Maletic., "*How we manage portability and configuration with the C preprocessor*". In *Intl. Conf. Softw. Maintenance*, pp. 275–284, 2007.

[19] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "*Feature consistency in compile-time–configurable system software: Facing the Linux 10,000 feature problem*". In 6th *EuroSys*, pp. 47–60, Apr 2011.

[20] L. Vidács and A. Beszédes, "*Opening up the C/C++ preprocessor black box*". In 8th *Symp. prog. lang. & softw. tools*, 2003.