Empirical Software Engineering (2022) 27: 68
https://doi.org/10.1007/s10664-021-10090-6

Incremental software product line verification - A performance analysis with dead variable code

Christian Kröher[1] · Moritz Flöter[1] · Lea Gerling[1] · Klaus Schmid[1]

Accepted: 28 November 2021 / Published online: 17 March 2022

Abstract
Verification approaches for Software Product Lines (SPL) aim at detecting variability-related defects and inconsistencies. In general, these analyses take a significant amount of time to provide complete results for an entire, complex SPL. If the SPL evolves, these results potentially become invalid, which requires a time-consuming re-verification of the entire SPL for each increment. However, in previous work we showed that variability-related changes occur rather infrequently and typically only affect small parts of a SPL. In this paper, we utilize this observation and present an *incremental dead variable code analysis* as an example for *incremental SPL verification*, which achieves significant performance improvements. It explicitly considers changes and partially updates its previous results by re-verifying changed artifacts only. We apply this approach to the Linux kernel demonstrating that our fastest incremental strategy takes only 3.20 seconds or less for most of the changes, while the non-incremental approach takes 1,020 seconds in median. We also discuss the impact of different variants of our strategy on the overall performance, providing insights into optimizations that are worthwhile.

Keywords Software product line analysis · Evolution · Incremental verification · Dead variable code analysis

Communicated by: Philippe Collet, Sarah Nadi, Christoph Seidl, and Leopoldo Motta Teixeira.

✉ Christian Kröher
kroeher@sse.uni–hildesheim.de

✉ Lea Gerling
gerling@sse.uni–hildesheim.de

Moritz Flöter
moritzf@gmail.com

Klaus Schmid
schmid@sse.uni-hildesheim.de

[1] University of Hildesheim, Hildesheim, Germany

1 Introduction

Software Product Line (SPL) engineering is an industrial-ready technique to create, manage, and maintain software families (Schmid and de Almeida 2013). Its core idea is to develop software as a set of related products, which share a common infrastructure, but vary in their individual capabilities. This *variability* enables the configuration and combination of generic artifacts to create a wide range of specific product variants.

Prominent examples of generic SPL artifacts are preprocessor-based *code artifacts* (Hunsen et al. 2016; Liebig et al. 2010) and conditional *build artifacts* (Nadi and Holt 2014; Adams et al. 2007). In code artifacts, preprocessor statements, like `#ifdef` for the C language, restrict the availability of certain code fragments for a final product. They define conditions that must be satisfied in order to include the respective code fragment they restrict. In build artifacts, similar conditions exist, which control the availability of entire files and directories. The evaluation of code and build conditions during the build and preprocessing of a final product depends on the selection of configuration options provided by a *variability model* (Eichelberger et al. 2013; Eichelberger and Schmid 2013; Czarnecki et al. 2012). This model is an abstract representation of all valid product configurations of a SPL. It contains configuration options for variable product capabilities and restrictions among them to avoid inconsistent or defective configurations. The code and build conditions reference these configuration options to include or exclude the related product capabilities based on a given configuration of the variability model.

A significant challenge in SPL engineering is the correct and consistent evolution of variability information (Mukelabai et al. 2018; Hellebrand et al. 2014; Livengood 2011). One of the reasons for this is the scattering of variability information across different artifacts as outlined above. In large SPLs this includes thousands of files (Passos et al. 2016), which are connected by a vast number of variability-related configuration options, conditions and references (Dietrich et al. 2012; El-Sharkawy et al. 2017; Nadi and Holt 2014). This complexity contains the risk of unintentionally introducing variability-related defects in or inconsistencies among those files as part of an evolutionary change.

There exists a plethora of SPL verification approaches to detect variability-related defects and inconsistencies (Thüm et al. 2014; Chastek et al. 2001; Meinicke et al. 2014; Benavides et al. 2010). The execution times of these approaches vary significantly with the goal of the analysis, the number of relevant artifacts, the amount and complexity of their inherent variability information, as well as the size of the SPL in general. For example, some analyses provide their results within seconds (Mendonca et al. 2009), while others require multiple minutes (Tartler et al. 2011), hours (Dietrich et al. 2012), or even days (Kästner et al. 2011). Upon evolution of a SPL, these delays will occur for every update (even minor changes), if verification is used to ensure correctness and consistency continuously.

In our previous work, we investigated the intensity of variability-related changes in SPL evolution to identify the amount and frequency in which variability information changes over time (Kröher et al. 2018c; Kröher and Schmid 2017b, a). The result of this analysis reveals that changes to variability information occur infrequently and only affect small parts of the considered types of artifacts. Hence, a re-verification of the entire SPL is often not necessary. We therefore argue that the explicit consideration of changes by implementing a regression-based analysis approach provides substantial benefits in terms of execution times. In this way, we can bring more of the SPL verification approaches within the reach of practitioners to meet industry demands (Mukelabai et al. 2018). While other approaches exist, which target a similar

goal (Szabó et al. 2016; Hamza et al. 2018), they typically come with their dedicated internal representation of artifacts to analyze or are limited to a subset of all types of artifacts containing variability information. Approaches considering variability information in all relevant types of artifacts typically have a different goal, like semantic reasoning on the type of changes applied to those artifacts over a sequence of commits (Dintzner et al. 2018).

In this paper, we present an approach based on a regression concept over all relevant types of artifacts and the results of analyzing its performance. The approach relies on our fine-grained commit analysis (Kröher et al. 2018c) as well as the consideration of the relations between input artifacts, extracted variability information, and the core algorithm of an analysis. We explicitly omit introducing additional models for change impact analysis, but utilize already available information only, like the changes documented by individual commits of a repository. The core analysis algorithm remains unmodified.

We describe this approach along an example from the domain of family-based static analyses (Thüm et al. 2014): the *dead variable code analysis* (Tartler et al. 2011; Nadi and Holt 2012; Dietrich et al. 2012) for identifying dead variable code blocks. While dead variable code is an important issue in its own right, it is a representative of a larger group of static analyses, which rely on the syntactic structure alone and are compositional. This compositionality allows detecting defects in some part of the implementation without considering the implementation at large. Hence, the analysis of an entire SPL is a composition of individual analyses for each part of it. Other analyses in this group are feature effects (Nadi et al. 2015) or configuration mismatches (El-Sharkawy et al. 2017). We selected the dead variable code analysis as a candidate to research our expected improvements before transferring our regression-based approach to other analyses in this group.

In order to analyze potential strategies for our approach and to compare their performance, we conceptually introduce different levels of change granularity and technically realize adaptation options to switch among them. The result is a set of three incremental variants of the dead variable code analysis, which consider changes on the level of files (called *Artifact Change Variant*), file content (called *Block Change Variant*), and variability information as part of file content changes (called *Configuration Block Change Variant*). Our analysis compares these three incremental variants and the original non-incremental one with respect to their performances by applying them to a subset of the Linux kernel evolution history (Torvalds 2020). The results of this comparison will answer the following research questions:

RQ1 To what extent can we reduce the analysis time, if we explicitly consider changes?
RQ2 How does the granularity of change identification impact the analysis performance?
RQ3 What is the overhead of using change identification in incremental verification?

The approach and results in this paper represent an important step towards a solution of industrial and practical needs in the context of SPL evolution (Mukelabai et al. 2018). In particular, we aim at a better understanding of the potential benefits and limitations of regression-based verification and the necessary granularity of change identification to achieve optimal results for an analysis during SPL evolution. The basic conclusions from our performance analysis with the dead variable code analysis can also be transferred to other compositional SPL analyses at a later point in time. Hence, we make the following contributions:

– We introduce incremental variants of the dead variable code analysis as a practical example for incremental SPL verification.

- We conduct a performance analysis comparing the incremental variants with the original, non-incremental dead variable code analysis.
- We discuss the analysis results along our research questions to provide evidence for the performance improvements.

In the next section, we provide the necessary background for the remainder of this paper. This includes the motivation for using the Linux kernel as a running example and as a reference in our performance analysis as well as an explanation of its variability realization. Further, we introduce the original dead variable code analysis and delimit it from the classical notion of dead code. We also outline our previous results from analyzing SPL evolution, which represent the main motivation for this paper and lay the foundation for our approach to incremental SPL verification in Section 3. Section 4 describes the technical realization of this approach and the tools we use as part of it. In Section 5, we define the setup (hardware, software, data set), the execution, and the measurements of the performance analysis. In particular, we explain the selection of the specific subset of the Linux kernel history and the validation of the correctness of the analysis results per variant. Section 6 presents the results of the performance analysis, while Section 7 discusses these results in accordance with our research questions. We discuss threats to validity in Section 8, related work in Section 9, and provide our conclusions in the final section.

2 Background

In this paper, we compare the performance of the original, non-incremental dead variable code analysis (Tartler et al. 2011; Nadi and Holt 2012; Dietrich et al. 2012) with our incremental variants. We therefore use a subset of the Linux kernel commit history as it provides the relevant types of artifacts with their individual, but interconnected variability information. Hence, Section 2.1 motivates this usage in more detail and discusses these relevant artifacts as well as the variability realization of the Linux kernel. Further, we refer to these descriptions in the remainder of this paper to provide illustrative examples. For instance, we already use these descriptions in Section 2.2 to introduce the original dead variable code analysis. As part of this introduction, we also differentiate between classical dead code and dead variable code as considered in SPL engineering. This introduction provides the necessary understanding for our incremental analysis approach presented in Section 3. Finally, we motivate our incremental approach by summarizing previous results and proposed optimizations from our SPL evolution analysis in Section 2.3.

2.1 Linux Kernel

The Linux kernel is a prominent reference example for a large-scale, real-world Software Product Line (SPL). This status yields from the extensive use of Linux by the SPL research community: either to understand SPL development in practice (Adams et al. 2007; Hunsen et al. 2016; Liebig et al. 2010; Passos et al. 2016) or to evaluate approaches supporting that development (Dietrich et al. 2012; El-Sharkawy et al. 2017; Nadi and Holt 2014; Tartler et al. 2011). As our contribution is part of the latter category, we will also focus on Linux for evaluation in this paper. Further, we will use excerpts from the Linux kernel implementation as a running example to illustrate our approach. Hence, this section introduces the relevant

artifacts and the variability realization of the Linux kernel. We limit this description to the extent necessary to understand our approach and its scope. In particular, we consider those types of artifacts relevant for the dead variable code analysis and for illustrating the difference to classical dead code in the next section.

The Linux kernel employs Kbuild (Passos et al. 2016) to realize variability. Figure 1 illustrates this realization by a real-world example of the Linux kernel version 4.8-rc1. It contains three code artifacts in the upper part, two build artifacts in the lower part, and a variability model artifact in the middle. Further, arrows indicate relations between some of these artifacts as established by the Kbuild variability mechanism. We start our descriptions by considering artifacts with such relations first. The explanation of unrelated artifacts (`copy.S` and `init_64.c` in Fig. 1) follows at the end of this section.

The upper part of Fig. 1 presents an example for each file type considered as code artifact in this paper. In Linux, these code artifacts consist of `*.c`- and `*.h`-files containing C-code and `*.S`-files containing assembler code. In all three file types, variability is mainly realized by preprocessor statements (Hunsen et al. 2016). Lines 780, 782, and 784 in `perf_event.h` contain such statements, which control the presence of the enclosed C-code-fragments (lines 781 and 783). The decision of which fragment will be part of the compiled kernel variant depends on the evaluation of the `#ifdef`-statement and the referenced symbol `CONFIG_X86_32` in line 780. This symbol denotes a configuration option defined in the variability model, which is indicated by the prefix `CONFIG_`. Hence, the presence of lines 781 or 783 depends on the selection or deselection of this option during the configuration process.

The `Kconfig`-file in Fig. 1 shows the definition of the `CONFIG_X86_32` symbol using the Kconfig language (Linux 2018). In the Linux kernel multiple `Kconfig`-files exist, that together represent the variability model. In such a file, the keyword `config` in lines 9 and 1811 indicates the definition of a configuration option followed by its name: `X86_32` and `KEXEC_FILE`. These names do not include the prefix `CONFIG_`, which is only used to

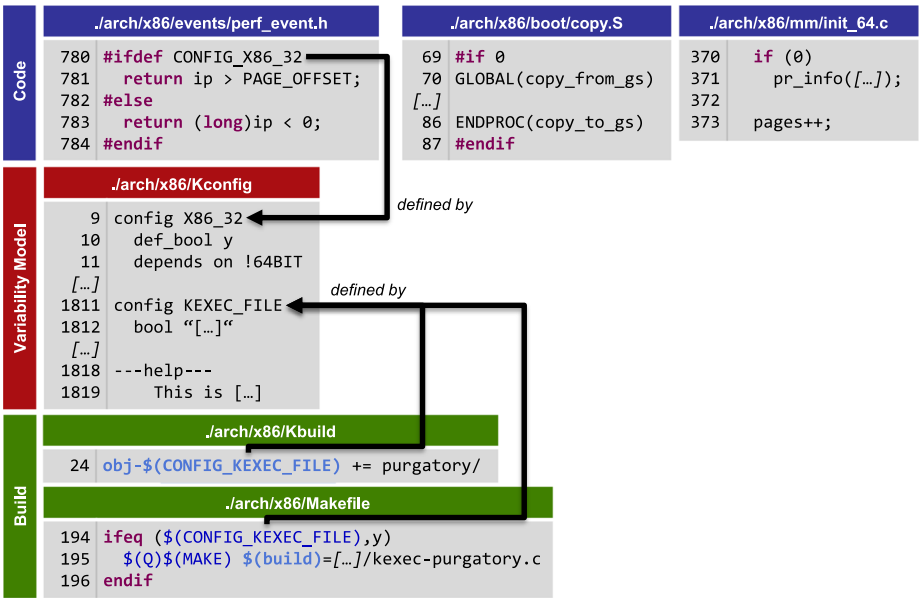

Fig. 1 Variability realization in the Linux kernel

indicate references to configuration options in other artifacts. Besides further variability information, like the type of a configuration option (e.g., Boolean in lines 10 and 1812) or dependencies between them (e.g., the `depends on` relation in line 11), Kconfig supports the definition of help texts and descriptions (e.g., lines 1818ff). They guide the end-user during the configuration process and explain the purpose and the impact of selecting a specific option.

The `KEXEC_FILE` option in our example is used to adapt the Linux kernel build process, which is mostly implemented in `Kbuild`-files and `Makefiles` using GNU Make commands (Linux 2019). In the `Makefile` the reference to the configuration option (again indicated by the prefix `CONFIG_`) is part of a condition, which restricts the presence of the enclosed command in line 195 (Nadi and Holt 2014). The `Kbuild`-file illustrates a different way to adapt the build process: The `KEXEC_FILE` option is used to modify the name of the variable `obj-`. The selection of this configuration option changes the name of the variable to `obj-y`. Hence, all files in the directory `purgatory` are compiled and linked during the build process, whereas the deselection excludes them.

Figure 1 includes two additional, but unrelated code artifacts (no incoming or outgoing arrows). The preprocessor statements (lines 69 and 87) in `copy.S` control the presence of the assembler code in lines 70 to 86. While the basic mechanism of controlling that presence in `copy.S` is similar to the one in `perf_event.h`, it differs with respect to the required information for decision-making. In `copy.S`, the decision of whether the assembler code is present in a compiled kernel variant solely depends on local information. Developers intentionally use this mechanism to exclude code fragments from compilation without removing them entirely from the code base. The evaluation of the `#if`-statement does not require additional information, like the selection or deselection of configuration options of the variability model in `perf_event.h`. Finally, `init_64.c` shows a control structure (line 370) in C-code to execute a particular code statement (line 371) conditionally. In contrast to the other code artifacts in Fig. 1, this form of control does not influence the presence of statements in a compiled kernel variant. Line 371 in `init_64.c` will always be present in this artifact. However, its actual execution depends on the evaluation of the control structure in line 370.

2.2 Dead Variable Code Analysis

The classical definition of dead code describes the presence of statements in the source code that are *never executed* under any circumstances (Aho et al. 2006). Hence, these code statements are called *dead*, like line 371 in `init_64.c` in Fig. 1. The statement in line 371 will never be executed as the related condition in line 370 is never satisfiable (always evaluates to false). The classical dead code analysis determines such unreachable code statements by computing the minimum sufficient code information required to execute them (Liu and Stoller 2003).

In the context of SPL engineering, however, dead code typically refers to code statements, which are *never present* in any product of the SPL (Tartler et al. 2011; Nadi and Holt 2012; Dietrich et al. 2012). Dead variable code and the related dead variable code analysis are fundamentally different from the classical definitions. For example, the dead variable code analysis will not identify the dead line 371 in `init_64.c` as it will be present in at least some Linux kernel variants, even though it will never be executed in any variant. In contrast, it detects dead variable code resulting from preprocessor statements, which define conditions that are not satisfiable. The code blocks enclosed by such preprocessor statements will always

be excluded during preprocessing and therefore are called *dead*. In this paper, we refer to this notion of dead variable code and the corresponding dead variable code analysis as described in the remainder of this section.

The dead variable code analysis was first introduced by Tartler et al. in (2011) and extended by Nadi and Holt (2012) as well as Dietrich et al. (2012) later. We use the extended analysis, which considers code, variability model, and build artifacts to determine whether a variable code block is dead. Dead variable code blocks typically arise from inconsistencies between the variability information in those types of artifacts. Hence, the process of the dead variable code analysis consists of two fundamental steps. First, it extracts the variability information from code, variability model, and build artifacts. Second, it combines the extracted variability information for each variable code block and analyzes it. We outline these steps in more detail below. However, for a comprehensive description, we refer to the original work on extracting variability information from code and variability model artifacts (Tartler et al. 2011) as well as from build artifacts (Nadi and Holt 2012; Dietrich et al. 2012) for dead variable code analysis.

The extraction step of the dead variable code analysis aims at providing propositional formulas as a common representation for analysis. The creation of these formulas and their semantics differ with respect to the type of artifacts they are extracted from. For code artifacts (`*.c`-, `*.h`-, and `*.S`-files in Fig. 1), the extraction considers each artifact independently to identify contained preprocessor statements, which impose a condition on the presence of a variable code block *cb*. For each variable code block, it extracts this presence condition for that block $pc(cb)$. If a (parent) block contains additional blocks, their presence condition is a conjunction of their parent block's condition and their own condition. This code extraction results in three variable code blocks and their presence conditions when applied to the example from the Linux kernel in Fig. 1:

- In `perf_event.h`, two blocks exist:
  - The first block cb_1 represents line 781, for which the presence condition $pc(cb_1)$ consists of `CONFIG_X86_32` defined by the `#ifdef`-statement in line 780
  - The second block cb_2 is the `#else`-part of that statement in line 783, for which the negation `!CONFIG_X86_32` defines the presence condition $pc(cb_2)$
- The third block cb_3 is located in `copy.S` (lines 70 to 86) guarded by the presence condition $pc(cb_3)$, which only contains `0` (`false`)
- For `init_64.c`, the code extraction does not provide any variable code block as there is no preprocessor statement defining a presence condition

The extraction for variability model artifacts (`Kconfig`-file in Fig. 1) considers all these artifacts to create a single propositional formula. This variability model formula *VM* represents all valid product configurations derived from the defined configuration options, their relations, and dependencies. As these definitions are scattered across various variability model artifacts, the derivation of all valid product configuration requires a mutual consideration of all of them. For the excerpt of the `Kconfig`-file in Fig. 1, the variability model formula *VM* will include a disjunction consisting of:

- *CONFIG_KEXEC_FILE* (derived from the definition of the respective configuration option in line 1811)
- *CONFIG_X*86_32 ∧ ¬*CONFIG_*64*BIT* (derived from the basic definition of the configuration option in line 9 and the assigned constraint in line 11, which requires `64BIT` not to be selected, if `X86_32` is selected)

The extraction from build artifacts (`Kbuild`-file and `Makefile` in Fig. 1) starts at the root of a SPL directory tree and recursively descends into the build artifacts of each subdirectory. On the paths from the root to leaf directories, the extraction collects each code artifact by its path relative to the root. Further, it assigns a condition to these code artifacts, which must be satisfied to include the respective artifact during build. This condition is a propositional formula defining the required combination of build prerequisites for a code artifact. The result of this build model extraction is a set of build model formulas *BM*. For example, the build statement in the `KBuild`-file in Fig. 1 results in a build model formula of `CONFIG_KEXEC_FILE` for each code artifact in the directory `purgatory` and its subdirectories. This configuration option must be selected to build the code files in these directories for a Linux kernel variant (cf. Section 2.1).

The analysis step of the dead variable code analysis aims at identifying dead variable code blocks. This step requires relating the extracted variability information from the different types of artifacts for each variable code block identified during code extraction. In general, a variable code block *cb* is dead, if the conjunction of its presence condition *pc*(*cb*), the build model formula in *BM*, and the variability model formula *VM* is not satisfiable:

$$dead(cb) = \neg sat(pc(cb) \wedge BM \wedge VM)$$

In the Linux kernel example (cf. Fig. 1), we identified three variable code blocks during code extraction as described above. Based on the illustrated excerpts only, the dead variable code analysis checks for cb_1 in `perf_event.h` the conjunction of $pc(cb_1)$, *BM*, and *VM* with the following values:

- $pc(cb_1)$ `CONFIG_X86_32`
- $BM = 1$ (as there is no condition in the build artifacts restricting the presence of `perf_event.h`)
- $VM =$ `CONFIG_KEXEC_FILE` ∨ `CONFIG_X86_32` ∧ ¬`CONFIG_64BIT`

The result of this check is `true` (satisfiable), which is negated to formulate that the variable code block cb_1 is not dead.

2.3 Variability Changes in SPL Evolution

The main motivation for our approach to incremental SPL verification presented in this paper is based on the results of our SPL evolution analysis (Kröher et al. 2018c; Kröher and Schmid 2017a, b). In this analysis, we investigate the frequency with which developers generally change a specific amount of variability information in code, build, and variability model artifacts. Therefore, we introduced a fine-grained analysis approach, which detects line-based changes in the commits of the Linux kernel repository (Torvalds 2020). The corresponding Commit Analysis (ComAn) tool (ComAn Team 2018) realizes this approach. In particular, it

provides a commit analysis process, which analyzes each file changed by a commit individually. The core steps of this process are:

- The general *determination of file types* using regular expressions to match the name of the changed file against the file types described in Section 2.1. This step restricts the subsequent analysis of file content changes to the prominent examples of generic SPL artifacts as introduced in Section 1.
- The individual *analysis of content changes* to code, build, and variability model files depending on the determined file type in the previous step. This step categorizes the addition or removal of individual lines into variability information and artifact-specific information changes.

In the Linux kernel example (cf. Section 2.1), these core steps of the analysis process classify any change to lines 780, 782, or 784 in the code artifact `perf_event.h` as a change to variability information. This is due to the `#ifdef`-statement in line 780, which marks the beginning of the entire preprocessor block and references a configuration option of the variability model. Changes to the enclosed `return`-statements (lines 781 and 783 in `perf_event.h`) are classified as artifact-specific changes in code artifacts. Similar classifications exist for build and variability model artifacts. For more information on the entire classification scheme and the corresponding identification, we refer to our previous publication (Kröher et al. 2018c).

The application of the ComAn tool to over 12 years of active Linux kernel development, considering 662.110 commits from initial commit `1da177e4c3` (2005/04/16) to latest commit `d528ae0d3d` (2017/03/16), reveals that variability information does not change significantly relative to the total number of changes. The results show that only 11% of all commits change variability information over all analyzed types of artifacts in general. Further, if variability changes occur, they mostly apply to code artifacts, followed by variability model artifacts and build artifacts. These changes typically affect 1-10 lines containing variability information independent of the specific type of artifacts.

The observations of our SPL evolution analysis led us to an outline of exploitation opportunities (Kröher et al. 2018c). A particular focus of these opportunities was on the group of static analyses to which the dead variable code analysis defined in Section 2.2 belongs to. For example, changes to general code statements do not influence the results of this analysis, as it only checks the presence of variable code blocks, but neither their content nor any other detailed code semantics. This avoids verification effort in about 78% of all commits in the evolution analysis. For the remaining commits, changes only affect few lines, which reduces the re-verification to exactly those affected artifacts. Further, some commits introduce changes to a particular type of artifacts exclusively, which allows reusing the information of unchanged artifacts for the re-verification of the changed ones.

In order to validate the potential of the opportunities outlined above, the next section introduces our corresponding approach to incremental SPL verification. This approach builds on the core steps of the analysis process described in this section to detect the changes and the amount of affected artifacts that require re-verification. In particular, these steps introduce different levels of change granularity to this approach. The first step regarding the determination of file types considers changes on the level of entire files only. The second step realizing the analysis of content changes considers fine-grained changes to individual file lines. This

differentiation lays the foundation for the resulting variants of the example of an incremental dead variable code analysis.

3 Incremental Verification

In this section, we introduce our approach to incremental SPL verification along the example of the dead variable code analysis. This analysis combines the presence condition of a variable code block with the build and variability model formula to determine whether a variable code block is dead (cf. Section 2.2). It therefore extracts these formulas from the respective types of artifacts. Hence, there exist relations between artifacts and their extracted formulas as well as between those formulas and the actual analysis. These relations are crucial to identify the impact of changes, such that we determining them in Section 3.1. As a result, we derive three fundamental levels of granularity at which we consider relevant changes. Section 3.2 builds on this result to detail these levels of granularity. In particular, the inspection of the relevant changes reveals the necessary re-verification effort and the expected analysis results for individual evolution scenarios at each level. Hence, we created one incremental variant of the dead variable code analysis for each level of change granularity following these descriptions. While we focus here on the dead variable code analysis, these sections describe the fundamental steps of our approach for building incremental SPL verifications in general.

3.1 Determining Relevant Relations

Figure 2 illustrates the different relations between artifacts, formulas (data models), and the actual analysis. The arrows from left to right indicate the two-step process of the dead variable code analysis as described in Section 2.2. We briefly recap the essentials of these descriptions along the arrows in Fig. 2 first. Based on this recap, we then discuss the resulting relations to derive relevant changes at different levels of granularity, which affect the results of an analysis. This lays the foundation for creating our incremental variants of the dead variable code analysis in the next section.

The extraction step of the dead variable code analysis provides a set of individual variable code blocks (cb_1 to cb_n), the variability model formula (*VM*), and the build model formula (*BM*). It therefore processes each code artifact separately to identify the variable code blocks along with their presence conditions ($pc(cb_1)$, etc.) defined by their ambient preprocessor statements. The variability model formula (*VM*) as well as the build model formula (*BM*) originate from the joint extraction of variability information from all variability model artifacts or build artifacts, respectively.

The analysis step uses these formulas as an input to its core algorithms, which analyzes each variable code block cb_x individually. It checks the satisfiability of the conjunction of a block's presence condition with the variability model and build model formula ($dead(cb_1)$ to $dead(cb_n)$ for short in Fig. 2). If this conjunction is not satisfiable, the respective variable code block is dead.

The relations (arrows) in Fig. 2 reveal that a variable code block and its presence condition are only relevant for the analysis of this particular block. In contrast, the variability model and build model formula influence the result of each analysis. In theory, it would be conceivable to only use those parts of the variability model and

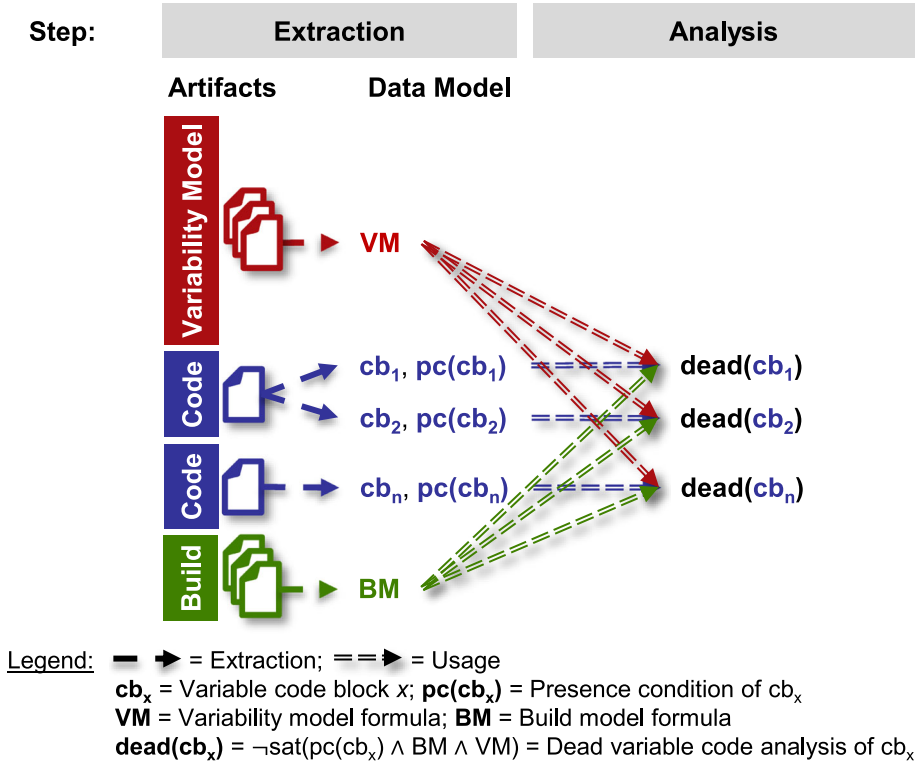

Fig. 2 Relevant relations for the dead variable code analysis

build model formula that actually control the presence of the respective variable code block under analysis. However, this is extremely difficult to do in a highly performant way due to the tight interactions among the various model parts and the way they are extracted (cf. Section 2.2).

The determination of relevant artifacts and their relations provides the scope of an analysis. Hence, changes become relevant for an analysis, if they affect artifacts in that scope in general and, in particular, those parts of their content that will be extracted. Based on the example of the dead variable code analysis, we generalize three fundamental levels of granularity at which we consider relevant changes:

1. The coarse-grained level of artifact changes, which are relevant, if the changed artifacts are in the scope of the analysis. For the dead variable code analysis, these artifacts are all code, variability model, and build artifacts.
2. The fine-grained level of content changes, which are relevant, if the changed content is in the scope of the analysis. For the dead variable code analysis, this is any content necessary to extract variable code blocks and their presence condition, the variability model or build model formula.
3. The fine-grained level of changes to the content in the scope of the analysis, which has additional characteristics, like holding specific relations to variability information. This level refines the second one and, hence, narrows the basic scope of an analysis. For the dead variable code analysis, we may only be interested in those variable code blocks,

which have an explicit reference to a configuration option of the variability model in their presence condition. In contrast to the second level, we would only re-consider blocks cb_1 and cb_2 of `perf_event.h` in Fig. 1 in this case, but not block cb_3 in `copy.S` (cf. Section 2.2).

We discuss these levels of granularity in detail in the following section using the example of the dead variable code analysis.

3.2 Identifying Relevant Changes

Different levels of granularity exist on which we can identify changes and determine their relevance for a specific analysis. In the previous section, we defined three fundamental levels of granularity based on the example of the dead variable code analysis. This section provides a detailed description for each of these levels. In particular, we take the perspective of the dead variable code analysis to illustrate how these different levels influence the relevance of a change and, hence, the necessary re-verification effort. We will structure our discussion along the following dimensions:

- **Granularity:** The basic description of the level of granularity defining the available details about a particular change
- **Relevant changes:** The general identification of changes triggering a re-analysis, if only the details of the respective granularity are known; here, we use the example of the dead variable code analysis and the changes illustrated in Fig. 3, which base on the example in Section 2.1
- **Extent of re-analysis and its results:** The detailed discussion on the required re-analysis effort for a particular change; based on the example of the dead variable code analysis and

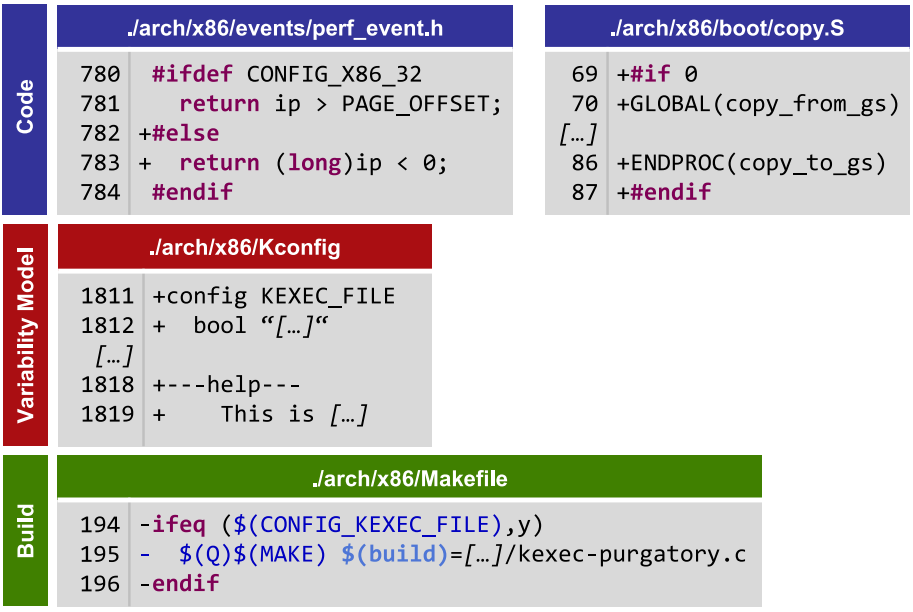

Fig. 3 Example changes for explaining incremental dead variable code analysis

the example changes in Fig. 3, we differentiate between *full analysis* (of the entire SPL), *partial analysis* (of changed parts only), or even *no analysis*

The results of this section are our three incremental variants of the dead variable code analysis, one for each level of change granularity.

3.2.1 Artifact Changes

This granularity represents the most coarse-grained level of change consideration in this paper. The corresponding incremental variant of the dead variable code analysis is called **artifact change variant**. It supports the granularity of change identification and performs incremental verification as described in this section.

Granularity: Artifact changes summarize basic changes to artifacts in the scope of an analysis. At this level of granularity, we do not consider any details about which information in those artifacts changes exactly. We focus on the type of artifacts only, e.g., if the extension or name of a changed file matches those that the extraction step of the analysis requires as input. This excludes fine-grained analyses of the changes and, hence, supports a fast determination of whether a change is relevant. However, the accuracy at this level of granularity is low as the actually changed content may be irrelevant for an analysis. In this case, the performed re-analysis will not provide any new results although its input artifacts have changed.

Relevant changes: The dead variable code analysis extracts information from code, variability model, and build artifacts (cf. Section 2.2). Hence, each change to at least one of these artifacts is relevant. Figure 3 indicates changed artifacts by a leading "+"- or "−"-sign for the addition of the respective line or its deletion, respectively. While the illustrated details are not available at this level of granularity, the files in Fig. 3 will all be classified as being changed. Each of these changes is therefore relevant for the analysis.

Extent of re-analysis and its results: A change of either variability model or build artifacts requires re-extracting the respective formulas completely. This is due to the dependencies between individual variability model or build artifacts (cf. Section 2.2). They potentially yield multiple changed clauses in the new variability model formula *VM* or the new build model formula *BM*, respectively, even if only a local change in one artifact occurs. This results in a *full analysis* of all variable code blocks due to the relations of *VM* and *BM* to all analyses in Fig. 2. However, all unchanged variable code blocks extracted during previous analyses can be reused. Further, we can also reuse *VM*, if only build artifacts change, or *BM*, if only variability model artifacts change. The expected result of that full analysis is the set of all dead variable code blocks in all code artifacts available at the current revision of the SPL.

The sole change of a code artifact requires the extraction of all variable code blocks from exactly that artifact and the re-analysis of those blocks only. For example, if only the changes to `perf_event.h` exist as shown in Fig. 3, the analysis only has to consider that file again. Hence, it re-analyzes the two variable code blocks reusing the unchanged variability model and build model formulas. In general, such a *partial analysis* provides the set of all dead variable code blocks in all changed code artifacts only. All other blocks from unchanged code artifacts are not affected as indicated by the relations in Fig. 2.

A combined change may affect code artifacts and either the variability model or build artifacts. This always results in a *full analysis* of all variable code blocks. However, reusing variable code blocks of unchanged code artifacts as well as any unchanged formula *VM* or *BM* is again possible.

In case of changes that do not affect any code, variability model, and build artifacts, *no analysis* is performed.

3.2.2 Block Changes

This granularity considers fine-grained changes on the level of individual file lines. Hence, the focus is on content changes instead of mere artifact changes as on the previous level. We call the corresponding incremental variant of the dead variable code analysis **block change variant**. It supports the granularity of change identification and performs incremental verification as described in this section.

Granularity: Block changes summarize all changes that affect specific parts of the content of artifacts in the scope of an analysis. In our particular example of the dead variable code analysis, this is any content related to the presence of variable code blocks; hence the name of this granularity. In general, at this level of granularity, we inspect each changed artifact in the scope of the analysis in detail. We analyze each changed line for containing information, which is subject to the extraction step of the analysis. This fine-grained change analysis typically requires more time, but provides higher accuracy regarding the relevance of changes than the previous level of granularity. For example, a change to an artifact in the scope of the analysis does not trigger a re-analysis, if the actually changed content in that artifact is irrelevant for the analysis. This omits the redundant executions of the previous analysis variant.

Relevant changes: The dead variable code analysis extracts individual variable code blocks, the variability model formula, and the build model formula from the respective types of artifacts (cf. Section 2.2). Hence, each change is relevant, which manipulates the variability information in those artifacts. In Fig. 3, only a subset of the illustrated changes matches this criteria:

- In `perf_event.h`, only line 782 represents a relevant change as it contains a preprocessor statement, which introduces a new variable code block. The change in line 783 only affects C-code, which is not in the scope of the dead variable code analysis. Hence, if only line 783 in `perf_event.h` changes, we do not need to re-analysis the blocks in that file. For the same reason, only the changed lines 69 and 87 in `copy.S` are relevant and trigger a re-analyis. The remaining changes in that file only affect mere assembler code (no re-analysis required).
- In the `Kconfig`-file, all changed lines are relevant except for lines 1818 and 1819. These two lines only introduce a help text, which does not influence the configuration options or their relations (cf. Section 2.1). All other lines define the configuration option `KEXEC_FILE`, which, for example, controls the execution of line 195 in the `Makefile`. A need for re-analysis therefore only exists, if these lines change. Changes to help texts as in lines 1818 and 1819 will not trigger a re-analysis.
- In the `Makefile`, lines 194 and 196 represent relevant changes, which trigger a re-analysis. This is due to the reference to a configuration option of the variability model in line 194. As the statement in line 196 closes the statement in line 194, it also has a

(indirect) relation to that reference. The change in line 195 does not affect any variability information and, hence, is irrelevant (no re-analysis required).

Extent of re-analysis and its results: A sole change of the variability information in any variability model or build artifact always requires the re-extraction of the respective formula *VM* or *BM*. As *VM* and *BM* influence the result of each analysis in Fig. 2, all variable code blocks have to be re-analyzed (*full analysis*), if such a change occurs. However, the unaffected variable code blocks as well as the unaffected *VM* or *BM* (if only one of them is affected by a change) can be reused. The expected result of that full analysis is the set of all dead variable code blocks in all code artifacts available at the current revision of the SPL.

A sole change of the variability information in code artifacts only requires the local re-extraction of variable code blocks cb_x from those code artifacts. For example, if only line 782 in `perf_event.h` changes as shown in Fig. 3, the analysis only has to consider the resulting new block from line 782 to line 784. It will not re-analyze the old block from line 780 to (now) line 782 as its presence condition does not change. However, the analysis will update the end line of that block from the previous analysis results in accordance to that change. In general, such a *partial analysis* only has to check the new or updated presence conditions $pc(cb_x)$ for satisfiability, while reusing the unchanged formulas *VM* and *BM*. Hence, it only provides the set of all dead variable code blocks in all code artifacts, in which variability information has changed. Previous results for unchanged code blocks remain unaffected.

A combination of the relevant changes described above always requires the re-analysis of all code blocks (*full analysis*). This is due to the relations of *VM* and *BM* to all analyses in Fig. 2. However, it is again possible to reuse unaffected elements, like unchanged cb_x, *VM* or *BM*.

This level of granularity extends the set of situations of the previous section, in which *no analysis* needs to be performed. Even if changes to code, variability model, or build artifacts occur, a re-analysis depends on their actually affected content. Hence, changes to irrelevant parts of their content do not trigger any re-analysis. On the previous level of mere artifact changes, such changes also trigger a re-analysis.

3.2.3 Configuration Block Changes

This granularity represents a refinement of the previous one. It also considers fine-grained changes on the level of individual file lines, but adds another filtering step to determine their relevance with respect to additional criteria. The corresponding incremental variant of the dead variable code analysis is called **configuration block change variant**. It supports the granularity of change identification and performs incremental verification as described in this section.

Granularity: Configuration block changes represent a specific subset of block changes (cf. Section 3.2.2). They change the relevant part of the content of artifacts in the scope of an analysis (equal to block changes) and fulfill additional criteria. Regarding the example of dead variable code analysis, this additional criterion is the explicit reference of blocks to configuration options defined in the variability model. Therefore, this variant is called configuration block change variant and the aforementioned blocks are called configuration blocks.

This reduction-by-design is beneficial, if only inconsistencies arising from these relations are of interest (as we will explain in more detail below). In general, this requires the analysis of each changed line not only to check whether a line is subject to the extraction step of the analysis, but also whether it includes a specific type of information. This extension may increase the time to determine whether a change is relevant. The accuracy regarding the relevance of a change depends on the additional criteria to consider. In the example above, we intentionally miss some dead variable code blocks, if we only focus on changes to configuration blocks. However, with respect to the additional criteria, the accuracy increases as the analysis will not consider any block, which does not satisfy the criteria.

Relevant changes: A particular goal of the dead variable code analysis is the detection of dead variable code blocks, which arise from inconsistencies between the variability information in code, variability model, and build artifacts (cf. Section 2.2). Hence, an additional criterion to consider during change analysis is whether a change affects the relations among variability information in these artifacts. A change therefore is relevant, if it manipulates the variability information in code, variability model, or build artifacts (equal to block changes) and that information either is referenced by other types of these artifacts or references the variability information in one of these other types. In Section 2.1, we described that variability information in variability model artifacts always defines configuration options, which are referenced in code and build artifacts. Consequently, configuration block changes affect the variability model formula *VM* and the build model formula *BM* as block changes do[1]. For code artifacts, the additional criterion excludes some of the changes illustrated in Fig. 3 from being relevant although affecting variability information. The changed lines 69 and 87 in `copy.S` are not relevant as opposed to identifying block changes. The presence condition of that variable code block does not include a reference to information defined in a different type of artifact. Hence, this block is not considered a configuration block and the change will not trigger a re-analysis on this level of granularity. The changes in `perf_event.h` are relevant as the resulting new block references the configuration option `CONFIG_X86_32` of the variability model in its presence condition (re-analysis required).

Extent of re-analysis and its results: A sole change of the variability information in any variability model or build artifact results in a *full analysis* of all configuration blocks. The previous extraction results of unaffected configuration blocks can be reused. Variable code blocks that are not configuration blocks are out of scope as defined above. Further, a reuse of the previous extraction results of an unchanged *VM* or *BM* is possible. The expected result of that full analysis is the set of all dead variable configuration blocks in all code artifacts available at the current revision of the SPL.

A sole change of a configuration block requires the re-extraction of that block. In Fig. 3, the changes in `perf_event.h` trigger the re-extraction and re-analysis of the added configuration block. In contrast, the changes in `copy.S` will be ignored due to the absence of configuration blocks. Hence, the resulting *partial analysis* only provides the set of all dead variable configuration blocks in all code artifacts, in which configuration blocks are changed. Previous results for unchanged configuration blocks remain unaffected.

A combination of the previous changes always requires a *full analysis* as described above. This is due to the relations of *VM* and *BM* to all analyses in Fig. 2. However, the same reuse potential as for block changes exists (cf. Section 3.2.2). If none of the changes

[1] The additional criterion, we use as an example here, does not change the relevance of changes to the contents of variability model and build artifacts with respect to the previous level of block changes.

above occurs, *no analysis* needs to be performed. In particular, this comprises all changes, which neither affect variability information in variability model and build artifacts nor configuration blocks.

4 Realization

The technical realization of the concepts introduced in Section 3 relies on the existing SPL analysis infrastructure KernelHaven (KernelHaven Team 2020; Kröher et al. 2018, 2018b) implemented in Java. It explicitly separates the fundamental steps of extraction and analysis in terms of individual plug-ins connected via internal data models of the infrastructure. In order to realize our three incremental variants of the dead variable code analysis, we extend this infrastructure by two additional steps:

- The **preparation** step before the extraction and
- The **post-extraction** step between the extraction and the actual analysis

The respective components enable the explicit consideration of changes and the reduction of analysis effort during SPL evolution.

We structure this section along the resulting four steps of an incremental dead variable code analysis as illustrated in Fig. 4 from left to right. For each step, we describe its purpose as well as its input and output data. Further, we discuss the processes and the configurability (if available) to realize the different variants of Section 3.2. The implementation as well as the exact release of the realization described here are publicly available.[2]

4.1 Preparation

The preparation component updates the SPL artifacts and manipulates the extraction of data models from them based on change information. This is the first step in our incremental analysis process as shown in Fig. 4. It receives the necessary change information before the actual application to SPL artifacts in terms of a *Diff*-file indicated by the delta in this figure. This file contains an entire Git commit (Git 2020) describing line-based changes to individual SPL files. The preparation component classifies these changes into relevant or irrelevant for the dead variable code analysis. If a change is relevant, this component prepares the re-extraction of the affected data models (second step in Fig. 4). For this classification and to only trigger required re-extractions, the preparation can use two different filters depending on the desired level of granularity of change identification (cf. Section 3.2):

- The **artifact change filter** detects **artifact changes** by scanning the input *Diff*-file for changed files via their names and extensions. The preparation component collects all changed code artifacts (`*.c`-, `*.h`-, and `*.S`-files) for individual re-extraction. Further, it triggers the re-extraction of the variability model or build model formula, if it detects the first changed variability model artifact (`Kconfig`-file) or build artifact (`Makefile` or `Kbuild`-file), respectively. We use this filter for the realization of the **artifact change**

[2] https://github.com/SSE-LinuxAnalysis/IncrementalAnalysesEvaluation/releases/tag/4.0

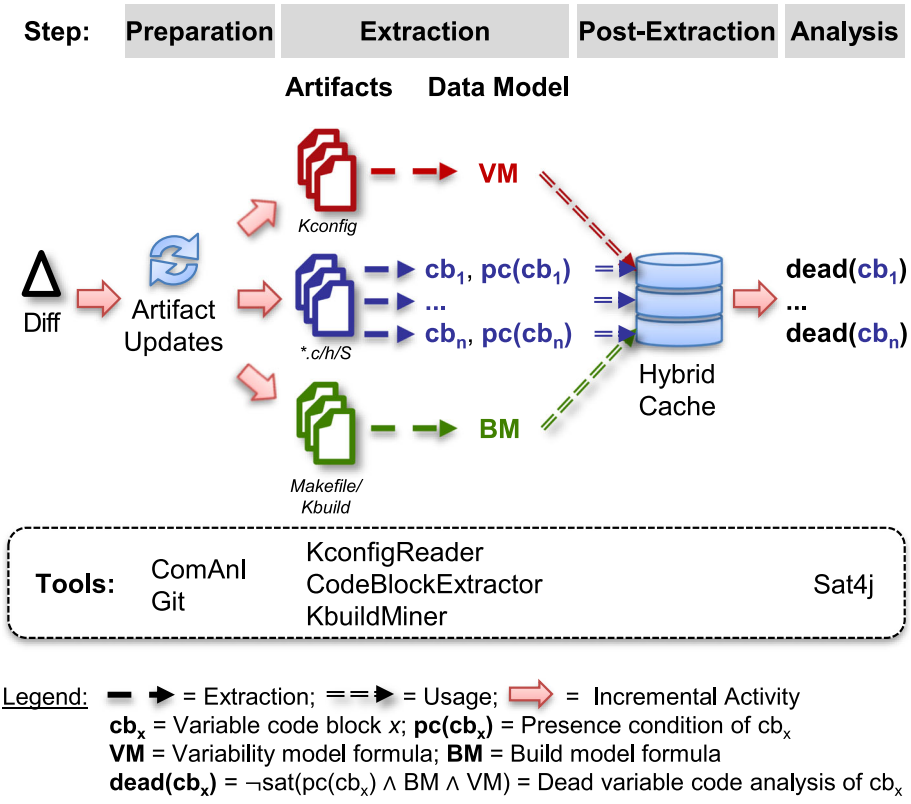

Fig. 4 Realization of the incremental variants of the dead variable code analysis

variant described in Section 3.2.1. It enables detecting exactly those changed files as required by that variant.

– The **variability change filter** enables the detection of **block changes** and **configuration block changes** by applying the commit analysis process of our Commit Analysis Infrastructure (ComAnI) (ComAnI Team 2019) to the input *Diff*-file. ComAnI represents an extended and configurable version of the tool used for our SPL evolution analysis (cf. Section 2.3). It provides the necessary level of details on which parts of the content of an artifact in the scope of the dead variable code analysis changes exactly (Kröher et al. 2018c). Hence, it detects the set of code artifacts in a given *Diff*-file in which variability information changes. These code artifacts are then subject to re-extraction in the next step. Further, it indicates changes to variability information in variability model and build artifacts via respective flags. If a flag is set, it triggers the re-extraction of the variability model formula or the build model formula, respectively. We use this filter for the realization of the **block change variant** (cf. Section 3.2.2) and the **configuration block change variant** (cf. Section 3.2.3).

The application of ComAnI as part of the variability change filter only provides a basis for detecting configuration block changes. The line-based analysis of the *Diff*-file identifies changes to individual variable code blocks reliably, but does not consider their nesting. Nested

variable code blocks may inherit references to configuration options of the variability model from their parent blocks during extraction. Hence, we shift the identification to the analysis after the extraction, which we describe in Section 4.4.

The preparation step ends after analyzing the given *Diff*-file regarding relevant changes by applying them to the previous revision of the SPL.[3] This application uses `git-apply` (Git 2021) (Git for short in Fig. 4), which reads the content of the *Diff*-file and applies it to the root directory of the SPL. The `git-apply` command takes care of correctly applying the inherent changes to the respective SPL artifacts. This step always performs the application of *Diff*-files to keep the SPL artifacts up-to-date, independent of the results of its change analysis.

4.2 Extraction

The extraction step is responsible for providing the data models from SPL artifacts in the scope of an analysis. This step requires individual components to process the different types of these artifacts as their realization relies on different languages. In the specific context of this paper, these components need to process code, variability model, and build artifacts, which are present in the Linux kernel. We therefore reuse the following, existing extraction plug-ins for the KernelHaven infrastructure:

- The KconfigReader extractor wraps the KconfigReader tool (Kästner 2016). It takes all variability model artifacts (`Kconfig`-files in the Linux kernel) as input and produces the variability model formula as described in Section 2.2. Based on the comparison of different `Kconfig`-tools by El-Sharkawy et al. in (2015), KconfigReader is the most reliable one regarding the correct and complete extraction of propositional formulas from such files.
- The CodeBlockExtractor is a platform-independent re-implementation of the extraction algorithm for code artifacts of the Undertaker tool (VAMOS/CADOS Team 2015). Undertaker realizes the original dead variable code analysis (Tartler et al. 2011) and, hence, provides exactly the code extraction capabilities required for our incremental variants of that analysis[4]. The basic algorithm takes a single code artifact (`*.c`-, `*.h`-, or `*.S`-file in the Linux kernel) as input, identifies the variable code blocks in that file, and determines their individual presence conditions as described in Section 2.2. The extractor calls this algorithm for each code artifact, if the input for the extractor is a directory, which may contain multiple code artifacts. The result of the extractor is the set of all variable code blocks with their presence condition for each code artifact. In order to support their identification, each block is represented by the path of its parent code file relative to the root directory of the SPL, the name of that file, a unique block number (cf. Section 2.2), and its start and end line in that file.
- The KbuildMiner extractor wraps the KBuildMiner tool (Berger and Kästner 2016). It takes all build artifacts (`Kbuild`-files and `Makefiles` in the Linux kernel) as input and produces the build model formula as described in Section 2.2. While other tools for this

[3] We use Git for compliance with the commits of the Linux kernel used in our performance analysis. However, any other system with equal capabilities regarding the documentation and application of changes could be used as well.

[4] We do not reuse Undertaker directly for our realizations as it does not consider variability information from build artifacts, the extraction from variability model artifacts is less accurate in comparison to KconfigReader (El-Sharkawy et al. 2015), and it is not maintained since 2015.

purpose exist, the parsing-based approach of KbuildMiner is (presumably) faster (Dietrich et al. 2012).

The actual execution of the individual extraction components in an incremental variant depends on the results of the preparation step. Further, the amount of code artifacts to re-extract changed variable code blocks and their presence conditions relies on that previous step. For example, a *Diff*-file may only contain the changes to `perf_event.h` as illustrated in Fig. 3. In the **artifact change variant** (cf. Section 3.2.1), the preparation step uses the artifact change filter to detect that a single code artifact (`perf_event.h`) has changed. It applies that change to the SPL and provides the path and name of the changed code artifact to the extraction step. That step only executes the code extraction from `perf_event.h`. All other extraction components will not be executed. Hence, the extraction step only provides the newly extracted variable code blocks from `perf_event.h` to the next step in this situation. In a similar way, only the variability model or build extraction components will produce their results, if the preparation step provides the respective flags indicating changes to these types of artifacts. If these are the sole change in a given *Diff*-file, the code extraction component will not be executed.

4.3 Post-Extraction

The post-extraction component acts as a mediator between the KernelHaven data models and their processing by the actual analysis. In general, it receives the entire data models from the previous extraction step. If one of the preparation filters presented in Section 4.1 is active, the received models only contain the respective results of their partial re-extraction. The component therefore provides a so-called **hybrid cache** (dedicated directory on the HDD). This cache contains the data models extracted from the previous revision of the SPL artifacts and their upcoming changes in terms of those data models resulting from re-extractions triggered by the content of the input *Diff*-file. In order to ensure that the data models in the hybrid cache always represent the recent state of the SPL artifacts, it updates its models as follows:

- If it receives a (new) build or variability model formula from the extraction step, it saves it, which potentially replaces the entire previous version. These formulas are global information of the entire SPL. Hence, the hybrid cache maintains each formula as a comprehensive instance, which it replaces and provides completely.
- If it receives variable code blocks of a code artifact and their presence conditions from the extraction step, it saves these presence conditions for the respective blocks and artifacts. Hence, the hybrid cache maintains each code block and its presence condition separately based on the unique information per block provided by the extraction step (cf. Section 4.2). In case that a particular variable code block is already available in the hybrid cache, it replaces this old information with the new one of the recent extraction. This does not affect any other blocks or their presence conditions.
- If the preparation component detects the deletion of variable code blocks (no extraction possible), it informs the post-extraction component directly (bypassing the extraction step). The hybrid cache removes the corresponding blocks; again, this does not affect any other blocks. The correct removal of deleted variable code blocks relies on matching the block information provided by the preparation step (e.g., in which lines variability

information is deleted) with the information provided by the extraction step for the previous revision.

The actual execution and the amount of necessary model updates depend on the incremental variant and the changes in a *Diff*-file. For example, any change to `copy.S` as illustrated in Fig. 3 will cause a hybrid cache update of the variable code block from line 69 to 87 in the **artifact change variant** (cf. Section 3.2.1). In contrast, this update will not occur in the **block change variant** (cf. Section 3.2.2), if only lines 70 to 86 change in `copy.S` (no variability information changes). For the same variant, an update of the variability model or build model formula is only necessary, if lines 1811 to 1817 in the `Kconfig`-file or line 194/6 in the `Makefile` change, respectively. Changes to other lines in these files for that variant do not require an update.

4.4 Analysis

The analysis is the last step in Fig. 4, which receives the hybrid cache introduced in the previous section. It uses the data models of this cache to execute its algorithms. This execution depends on the variant of the dead variable code analysis and, hence, on the selected filter for the preparation component (cf. Section 4.1). If no filter is active, the analysis considers all variable code blocks available in the current state of the hybrid cache. This execution corresponds to an original, non-incremental dead variable code analysis. An active preparation filter reduces the analysis to changed elements only, while reusing unchanged elements as defined in Section 3.2. Further, we introduce two additional **analysis options**, which enable to ignore all variable code blocks with presence conditions that do not reference any configuration option of the variability model. Details on these options can be found in the documentation of the incremental analysis infrastructure (KernelHaven Team 2019a, b). This realizes the additional filtering for **configuration block changes** (cf. Section 3.2), which the **variability change filter** does not support (cf. Section 4.1).

The resulting configurability in terms of selected filters and analysis options technically enables the original, non-incremental and our three incremental variants of the dead variable code analysis. For executing the original, non-incremental analysis, we disable the preparation and post-extraction step as well as the analysis options. This results in a full analysis of the entire SPL independent of the changes of a *Diff*-file. All three incremental variants make use of the hybrid cache in the post-extraction step. The artifact change variant of the dead variable code analysis uses the artifact change filter during the preparation step, while the block change and the configuration block change variant both use the variability change filter. The configuration block change variant is the only variant that additionally enables the analysis options.

The core analysis algorithm for detecting dead variable code blocks as described in Section 2.2 is not affected by the configurability. It always iterates the input code artifacts and individually checks the satisfiability of the conjunction of the presence condition, the variability model, and build model formula for each variable code block[5]. As this conjunction defines a Boolean satisfaction (SAT) problem, the analysis algorithm employs the KernelHaven SAT utilities for this purpose in our realization. They provide access to auxiliary methods, like the conversion of propositional formulas to their conjunctive normal forms or the creation of conjunctions of such formulas. Further, the SAT utilities use the actual SAT solver Sat4j (Sat4j Team 2021), which has already been successfully applied to check large-scale SPL models

[5] Only the input depends on the selected analysis variant, but not the core algorithm.

(Mendonca et al. 2009). The analysis algorithm creates a new instance of the entire SAT utilities (including the SAT solver) for the check of the first variable code block of a code artifact. Hence, the instantiation of these utilities only occurs, if a code artifact contains at least one variable code block. The new SAT utilities instance receives the presence condition of the variable code block, the build model formula for the related code artifact, and the entire variability model formula. The auxiliary methods convert and combine these individual formulas to the final conjunction, which the SAT solver checks for satisfiability. For the remaining code blocks of the same code file, the analysis algorithm reuses the SAT utilities instance. In this way, the utilities also allow reusing the build model and variability model formula for the checks of each variable code block of the same code artifact as only the block presence condition changes. If the check of a particular block returns `false` (the conjunction is not satisfiable), all information about that dead block will be cached first and printed to a result file later after the termination of the analysis step in Fig. 4.

5 Performance Analysis

In the performance analysis, we executed our technical realization described in Section 4 in a virtual machine. The exact image used for our analysis along with a description of our host system is publicly available on GitHub[2]. This repository also contains our suite of implementations and configurations as well as all (intermediate) results. These artifacts enable the execution of the (non-)incremental dead variable code analysis and the reproduction of our results out of the box. In this section, we first describe the **hardware** and **software** constituting the particular setup used in this paper. We then discuss the **data set** (commits from the Linux kernel) and explain the selection of this specific subset. Further, we describe the **measurements** and the **execution** of the individual analysis variants. The **validation** of our results follows at the end of this section.

The **hardware** of the host for our virtual machine consists of two Xeon E5-2650 v4 CPUs with 2.20GHz and 12 cores each. We enabled hyperthreading, which results in 48 logical CPU cores. Further, the host offers 384GB registered DDR4 ECC RAM (2400MHz) and a 1.8TB SAS HDD with 128MB buffer. Based on these resources, we configured the virtual machine with access to all 48 logical CPU cores, 128GB RAM, and 128GB mass volume storage. While the performance analysis is also executable on consumer hardware, like an Intel J4105 CPU and 32 GB RAM, the more powerful configuration accelerates the execution over a large number of commits. Moreover, compared to the continuous integration scenario, which is one of the motivations for our work, this is not an unusual machine size.

The **software** is based on Ubuntu 16.04.3 as operating system for the virtual machine. For executing our implementation, installations of OpenJDK 1.8.0_91, Git 2.21.0, and some KernelHaven-specific libraries (KernelHaven Team 2020) are used. Further, we assign 10GB as initial heap memory size and 50GB as its maximum to the Java virtual machine.

The **data set** consists of 440 commits, from commit `0c744ea4f7` (2017/01/01) to `ea6200e841` (2017/03/08), of the main branch of the Linux kernel repository (Torvalds 2020). We saved these commits as individual files before the performance analysis and then applied them successively in historical order as described as part of the execution below. The majority of these commits represent large merges from different contributors to the main Linux kernel development branch. Only some commits in our data set represent smaller commits, which individual developers typically create in practice. We would have liked to use these

smaller commits only as they are the smallest unit of precisely documented developer changes publicly available. In that sense, we would have been able to reproduce individual evolution steps successively and apply our approach in a continuous-integration-fashion to large-scale, real-world SPL development. However, skipping the intermediate merges of the main branch would have invalidated the correct evolution during reproduction. Further, the consideration of a developer branch of a particular kernel module with more consecutive developer commits would not represent the development of the entire SPL and, for this paper, of the entire Linux kernel. Hence, the commits of the main branch bring us as close as possible to our target evolution scenarios.

The selection of the specific commit sequence as our data set has several reasons. First, we chose this sequence as it has significant length while being feasible with our resources in an appropriate time frame (especially for the original, non-incremental variant). Second, as the Linux kernel grows over time (Israeli and Feitelson 2010; Adams et al. 2007; Godfrey and Tu 2000), we aim at a rather recent sequence to consider the current complexity of a real-world SPL. However, latest versions of the Linux kernel also introduce updates to the concepts of Kbuild and, in particular, of the Kconfig language. These updates cause some of our tools for extracting data models (cf. Section 4.2) to fail. For example, the integrated third-party tool KconfigReader fails to extract the variability model formula for more recent versions of the Linux kernel at least sometimes. Hence, we selected our commit sequence as a compromise between up-to-date development and reliability of our technical realization. Finally, we wanted to use a sequence, which is a representative of the longer history of the Linux kernel in terms of the percentage of variability-related changes. We therefore applied our analysis of the change frequency described in Section 2.3 to our data set. The comparison of relative percentages of variability-related changes in that sequence with the results from the entire Linux kernel history in Kröher et al. (2018c) confirms that this sequence is a reasonable representative.

The **measurements** for the results in this paper collect several timings and the results of each analysis. In order to measure the performance of the analysis variants, we use the `time` command to gather the entire execution times per commit. Further, we extract the wall clock time from timestamps at the beginning and the end of the individual analysis steps in the implementation (cf. Section 4). These times enable us to present general and detailed performance results in the next section. We also saved the identified dead variable code blocks for each analysis variant per commit. They are input to our validation as described at the end of this section.

In general, the **execution** applies the commits in our data set successively in historical order to the latest version of the Linux kernel before the first commit in this sequence. After each of these applications, it executes one particular variant of the dead variable code analysis once. Hence, we perform this execution four times, once for each variant. A set of bash scripts automates this commit application (if necessary) and the actual executions of the analysis variants by starting KernelHaven with the respective configuration (cf. Section 4.4) for each commit. For the original, non-incremental variant, a script applies the changes introduced by a commit to the Linux kernel before the execution of that variant. The scripts for the incremental variants provide the commit as an input *Diff*-file to KernelHaven, which applies the contained changes as described in Section 4.1 before an analysis.

The aim of our **validation** is to ensure that the performance results of the individual analysis variants do not stem from an unintended loss of accuracy regarding the identification of dead variable code blocks. Our measurements therefore provided the sets of identified dead

blocks as described above. The blocks identified by the original, non-incremental variant served as a baseline, which provides all dead variable code blocks that the basic analysis can identify in the entire Linux kernel after each evolution step (simulated by the application of commits of our data set). For each incremental variant, we automatically compared its identified dead variable code blocks for each commit with the corresponding results of the baseline. This comparison did not reveal any false-positive or missing dead variable code blocks, except for those blocks that are intended to be absent in the configuration block change variant (cf. Section 3.2.3). Hence, we ensure the expected level of accuracy for each incremental variant and no unintended influences on the performance results presented in the next section.

6 Results

In this section, we present the results of our performance analysis that applied the four dead variable code analysis variants to the 440 Linux kernel commits as described in Section 5. However, these results exclude the execution time for the first commit in this sequence, as it initializes the components enabling the incremental variants during the first analysis. This does not represent a typical incremental verification scenario. Hence, Fig. 5 presents the performance results of all variants for the remaining 439 commits and structures them into the following categories:

- *Execution Time per Commit in Seconds*: The four plots in the upper part of the figure combine a barcode plot with a violin plot for each analysis variant. One horizontal line represents one data point, which illustrates the execution time in seconds of the respective dead variable code analysis variant for one commit. The bold horizontal line is the median execution time for this variant. The background area illustrates the probability density function of the data using the Epanechnikov kernel (Epanechnikov 1969) and the Silverman's rule-of-thumb bandwidth estimator (Silverman 1986). The value in the upper right corner of each plot is the standard deviation of the execution times for this variant, with the relative deviation in parentheses. This category is specifically relevant to answer the research questions **RQ1** and **RQ2**.
- *Number and Results of Analyzed Blocks*: The three values below the legend of each violin plot describe the average number of analyzed blocks, the standard deviation of the number of analyzed blocks (with the relative value in parentheses), and the minimum and maximum number of found dead variable code blocks for the respective variant. Note that the configuration block change variant detects only the dead variable code blocks with reference to a configuration option, therefore the terminology is slightly changed to dead variable configuration blocks. This category is partly relevant to explain the differences in the execution times.
- *Average Execution Time Distribution (in Seconds) per Variant*: The stacked bar chart in the left part of Fig. 5 shows the average execution time distribution in seconds for the different analysis steps of our technical realization described in Section 4. The chart contains also an additional category called *Transition Time*, that illustrates the time difference between the whole execution time and the sum of the four steps, resulting from the different measurement techniques explained in Section 5. This category is specifically relevant to answer research question **RQ3**.

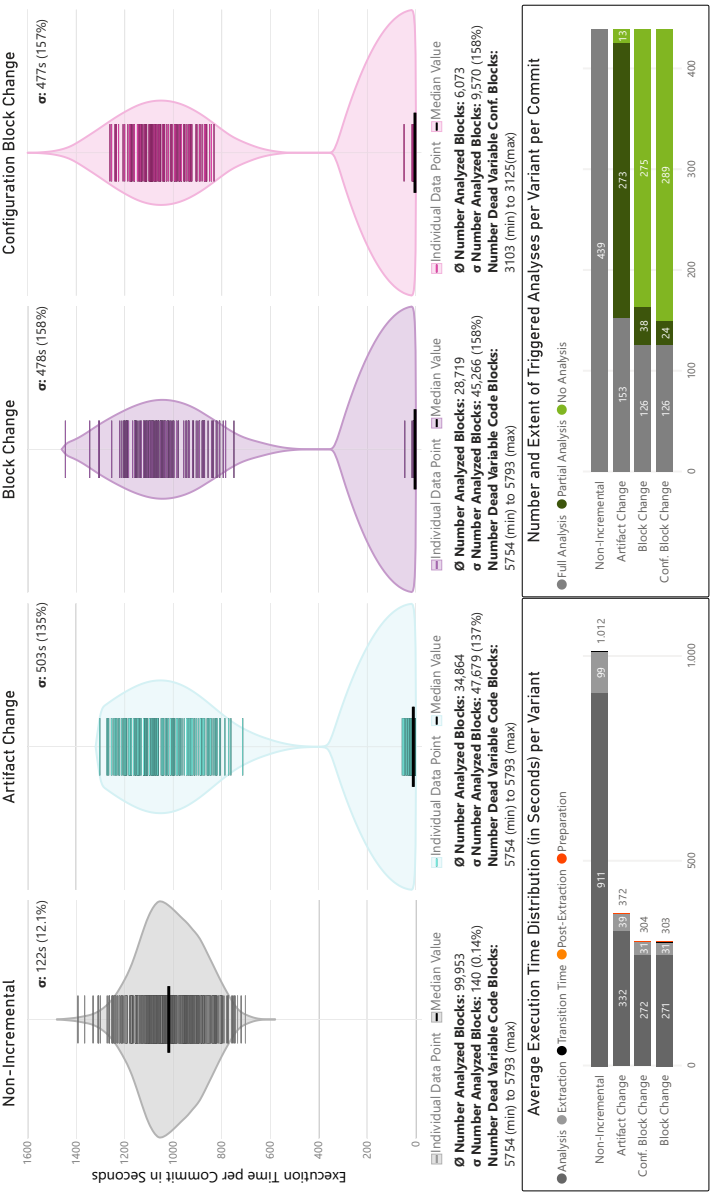

Fig. 5 Performance results summary for each analysis variant

– *Number and Extent of Triggered Analyses per Variant per Commit*: The stacked bar chart in the right part of Fig. 5 illustrates how often the changes of a commit trigger a full, partial, or even no analysis of the SPL for each analysis variant. This category is specifically relevant to explain the differences in the execution times.

In the following sections, we describe the performance analysis results in detail for each analysis variant. Section 6.1 presents the original, **non-incremental variant** of the dead variable code analysis and Sections 6.2 to 6.4 proceed with the individual **incremental variants**. For each of them, we first present their summarized results and compare them to the variants in previous sections. Further, we use the results of the individual analysis variants to explain the respective performances as well as additional findings.

6.1 Non-Incremental Dead Variable Code Analysis

The original, non-incremental variant of the dead variable code analysis always performs a *full analysis* for each of the 439 commits, as shown by the uppermost bar in the right bar chart of Fig. 5. For each commit, it analyzes about 99,953 code blocks within about 1,012 seconds on average (ranging between 702 and 1,393 seconds). The median of this execution time has a value of 1,020 seconds. The results of the dead variable code analysis show a minimum number of 5,754 dead variable code blocks, up to a maximum of 5,793, which are about 17% of all code blocks.

The respective ranges show a relatively low variation in the number of code blocks, but a relatively high variation in the overall execution time with a standard deviation of about 0.14% for the number of code blocks, but about 12.1% for the execution time. Hence, we calculated the correlation between those variables and identified almost no relationship (correlation coefficient of about 0.09). This rejects our general assumption that the execution times depend on the number of code blocks to analyze in this variant.

In order to find a reason for this observation, we inspected the execution times of the individual steps of the overall analysis process, which are presented by the uppermost bar in the left bar chart of Fig. 5. The *extraction* of the necessary data models (formulas) only consumes about 10% (99 seconds) of the overall execution time, while their actual *analysis* requires about 90% (911 seconds) on average. Hence, we focused on the *analysis* step and identified variations in the order of the elements of the extracted variability model formulas. While the resulting formulas are logically equivalent, this variation also occurs, if the input to the *extraction* step consists of the same, unchanged artifacts. We therefore tested the standard deviation of the execution times by repeating the entire analysis process ten times with a cached variability model formula for all repetitions and another ten times with re-extracted formulas for each repetition. This test results in a more than three times lower relative standard deviation, if we use the same formula (3.3%) compared to a re-extracted one (11%). Therefore, we argue that the *extraction* of the variability model formula causes the variation in the performance at least to some extent. As we rely on a third-party tool for this extraction (cf. Section 4.2), a more elaborated explanation or even a correction of this behavior is not possible.

A possible solution to mitigate the identified variation would be to realize a caching of the variability model formula, which enables the reuse of an already extracted formula as long as the artifacts input to the *extraction* step do not change. However, this would change the core process of the original dead variable code analysis towards a variant similar to our incremental

ones. Further, we use the same extraction capabilities for all dead variable code analysis variants in this paper. Hence, the variation applies to all performances, in particular, during *full analysis*, independent of the considered analysis variant. As a consequence, the execution time for the same commit can vary noticeable. For example, all analysis variants perform a *full analysis* for the changes introduced by commit 382 in our sequence. For this analysis, the non-incremental variant requires 1,330 seconds, the artifact change variant 888 seconds, the block change variant 1,158 seconds, and the configuration block change variant 1,254 seconds.

6.2 Artifact Change Variant

The incremental artifact change variant performs about 2.72 times faster than the non-incremental variant. It requires about 372 seconds (ranging between 2.93 to 1,303 seconds with a standard deviation of 135%) on average and has a median execution time of 10.5 seconds as shown by the second plot in the upper part of Fig. 5. The reason for this improvement is the reduced number of analyzed code blocks. This variant only considers about 34,864 code blocks on average with a standard deviation of 137%. This reduction is due to the shift from always performing a *full analysis* to performing a *partial analysis* for the majority of the commits (273 out of 439) and even *no analysis* for 13 commits (see second bar in the right part of Fig. 5). The positive correlation coefficient of 0.88 between the execution time and the number of analyzed code blocks supports this rationale. The results of this dead variable code analysis are the same as the non-incremental variant, with a minimum number of 5,754 dead variable code blocks, up to a maximum of 5,793.

The extent of triggered analyses results in either very fast execution times, or about similar execution times compared to the non-incremental variant, as the distribution of the data points in the artifact change plot in Fig. 5 shows. For the 153 commits requiring a *full analysis*, 99,977 code blocks are analyzed in about 1,052 seconds on average, which is around 3,80% slower than the non-incremental variant. While the additional execution of the *preparation* and the *post-extraction* step only introduce an overhead between 0.01% and 0,03% to the general performance, we assume that this increase in the execution time is at least partially due to the identified variations in Section 6.1.

273 commits trigger a *partial analysis* that analyzes on average 32 code blocks in approximately 8.90 seconds. For example, the analysis of commit 407 takes 1,393 seconds for the non-incremental variant, but only 8.70 seconds for this analysis variant. The remaining 13 commits, which require *no analysis*, do neither change any build or variability model artifacts nor any code artifacts and, hence, no code blocks. In order to detect this, the infrastructure needs to load completely and execute the *preparation* step, which results in an average of 3.66 seconds execution time for the *no analysis* part. For example, the analysis of commit 438 took 1,249 seconds for the non-incremental variant, but only takes 2.99 seconds for this analysis variant. In combination, the *partial analysis* and *no analysis* accelerate the analysis for about 65% of the commits (273 + 13 out of 439) by a factor of 121 (average execution time for *partial* and *no analysis* weighted by their respective number of commits, divided by average execution time for full analysis of this variant).

6.3 Block Change Variant

The incremental block change variant performs on average about 1.23 times faster than the previous one and about 3.34 times faster than the non-incremental one. It requires on average

around 303 seconds (ranging between 2.91 to 1,445 seconds with a standard deviation of 158%) for analyzing an individual commit with a median value of only 3.20 seconds. The distribution of the performance data is modeled in the second right plot in Fig. 5. The reason for this improvement is again the reduced number of analyzed code blocks: This variant reduces the average number of code blocks to consider to about 28,719 on average with a standard deviation of 158%, which are approximately 17.63% fewer blocks than the artifact change variant considers. This is caused by a significant shift to performing *no analysis* for a commit (275 out of 439), as illustrated in the block change bar in the right bar chart in Fig. 5. The positive correlation of 0.68 between the number of code blocks and the execution time supports this explanation, although it is not as strong as the correlation of the previous variant. The results of this dead variable code analysis are the same as the non-incremental variant, with a minimum number of 5,754 dead variable code blocks, up to a maximum of 5,793.

Similar to the artifact change variant, the execution time per commit is either very low, or comparable to the non-incremental variant. There are only 126 commits that require a *full analysis* that takes on average 1,048 seconds to analyze about 99,982 blocks. Similar to the artifact change variant, the execution time for these analyses is a bit slower (3.43%) mostly due to the identified variations of the *extraction step* as explained in Section 6.1. The left bar chart of Fig. 5 shows that the additional incremental steps of *preparation* and *post-extraction* add just a minor overhead of about 0.19 and 0.12 seconds on average to the execution time of the block change variant.

In contrast to the previous variant, the number of commits which require a *partial analysis* decreases significantly to 38, as the different preparation filter of this variant's configuration triggers such analyses only, if changes affect individual code blocks instead of entire code files. These partial analyses analyze around 84 code blocks in 6.23 seconds on average. For example, commit 256 in our sequence triggers a *full analysis* of the non-incremental and the artifact change variant, which requires 1,055 seconds and 1,122 seconds, respectively. For the same commit, this analysis variant only requires 9.43 seconds due to performing a *partial analysis*. The majority of the commits (275) trigger *no analysis*, as they do not change information relevant for the analysis of any block. Hence, this variant only requires about 3.50 seconds on average to reject further analyses. The performance values for commit 407 in our sequence illustrate the significance of this reduction. The non-incremental variant performs a *full analysis* for this commit, which requires 1,393 seconds, while the artifact change variant requires 8.70 seconds due to a *partial analysis*. This analysis only requires 3.10 seconds for that commit as it performs *no analysis*. In combination, the *partial analysis* and *no analysis* accelerate the analysis for about 71% of the commits (38 + 275 out of 439) by a factor of 273.

6.4 Configuration Block Change Variant

The incremental configuration block change variant takes on average around 1 second or 0.33% more time than the block change variant, but is still about 3.33 times faster than the non-incremental one. The average execution time for each commit is approximately 304 seconds (ranging between 2.87 to 1,261 seconds with a standard deviation of 157%) and takes a median of 3.62 seconds, as illustrated in the right plot in Fig. 5. Similar to the other two incremental variants, the number of analyzed blocks is further reduced: The configuration block change variant analyzes only about 6,073 code blocks on average, with a standard deviation of 158%. This is achieved by a slightly reduced number of *partial analyses* (24), and

an increased number of *no analyses* (289), as shown in the right bar chart for this variant in Fig. 5.

Furthermore, this variant only considers configuration blocks instead of all blocks. Therefore, the results of this dead variable code analysis are different from those of the other variants, with a minimum number of 3,103 dead variable configuration blocks, up to a maximum of 3,125. The correlation coefficient for the number of analyzed blocks and the average execution time is strong positive with a value of 0.70, which is comparable to the block change variant. However, these numbers do not explain why the configuration block change variant is faster than the non-incremental and the artifact variant, but not faster than the block change variant.

The average execution times of the individual steps in the left bar chart of Fig. 5 present a first explanation for the missing impact of the more fine-grained filters on the general performance: All steps perform almost equal to the block change variant. In particular, the performance of the *analysis* step, which has a significantly reduced number of code blocks to analyze due to the focus on configuration blocks instead of all blocks, shows almost no change compared to the block change variant. We therefore inspected this step in detail with regard to the extent of the triggered analysis. For *no analysis*, there was no significant discrepancy in the actual versus the expected results: The analysis needs on average 3.55 seconds to reject a further analysis of 289 commits, which is in line with the other incremental variants. For example, the analysis of commit 407 takes 1,393 seconds for the non-incremental variant, 3.10 seconds for the block change analysis variant (triggered *no analysis*), and 3.01 seconds for this variant (triggered *no analysis*). For the *full* and *partial analysis*, however, we detected two main reasons for their unchanged performance.

First, the *analysis* step uses a new instance of the SAT utilities for each code file as described in Section 4.4. While the individual satisfiability checks of the integrated SAT solver are extremely fast, the initialization of the entire SAT utilities for the analysis of the first variable code block of a code file requires most of the analysis time. Hence, the reduced number of variable code blocks to analyze in this variant does not have a significant effect on the analysis time due to spending most of it on the initialization of the required SAT utilities. This explanation is supported by the performance data of the triggered 126 *full analyses* in this variant: One *full analysis* requires on average about 1,051 seconds to analyze around 21,140 blocks, which is a similar amount of time in comparison to the other variants, but a major reduction of the number of blocks.

Second, the detection of references to configuration options in a presence condition during the analysis step adds an overhead to the analysis of the respective code block compared to analyzing it directly. This overhead has its effect, in particular, on the *partial analysis*, where the configuration block change variant is about 34% slower than the block change variant. It takes an average execution time of 9.57 seconds, although the number of changed code files does not change, resulting in the same time for the initialization of the SAT utilities. For example, the analysis of commit 256 takes 1,055 seconds for the non-incremental variant, 9.43 seconds for the block change analysis variant (triggered a *partial analysis*), and 15.67 seconds for this analysis variant (triggered a *partial analysis*). This is in contrast to the reduced number of analyzed code blocks: The *partial analysis* has to consider only about 34 code blocks, which is 40% of the code blocks the block change variant considers.

The two main reasons discussed above arise from our technical realization. While the algorithms realizing the additional filtering, which causes the overhead, may offer some optimization potential, it will not be possible to completely avoid the overhead as filtering is

a core feature of this incremental variant. The creation of a new instance of the SAT utilities for each code artifact reveals a bottleneck regarding the performance of this variant. We aim at an improvement by realizing other strategies, like extending the reuse of an instance as long as possible, or using a pool of constantly available instances (no re-instantiation during analysis) in the future. Currently, the usage of the SAT utilities is the same for all variants of the dead variable code analysis with the respective effect described above. An optimization of this usage, which puts the focus on the actual code blocks to consider, will mostly affect our incremental variants in a positive manner as indicated by the average number of code blocks. However, even without these optimizations, this variant accelerates the analysis by a factor of 261 for about 71% of the commits (24 + 289 out of 439).

7 Discussion

The results of our performance analysis show a significant potential of the incremental verification approach proposed in this paper. Table 1 summarizes the key results from Section 6. It presents the average and median execution times per analysis variant as well as the percentage of detected newly introduced dead variable code blocks. Further, the table highlights the advantages and drawbacks of each variant.

All three incremental variants provide a significant speed-up of the average execution time, which is up to 3.3 times faster for the block and configuration block change variant in comparison to the the non-incremental variant of the dead variable code block analysis. This speed-up is even more distinct when observing the median execution times. For most of the commits, the execution time can be reduced to 10.5 seconds or less when using the artifact change variant, up to 3.20 seconds or less when using the block change variant. This speed-up answers research question **RQ1** and results from the explicit consideration of the changes introduced by commits, the identification of their impact based on the internal processes of the dead variable code analysis, and the execution of only the necessary extent of the re-analysis.

A particular observation from our performance analysis is the impact of the granularity of change identification. While the change identification on the level of artifacts and code blocks

Table 1 Key results of the performance analysis

	Non- Incremental Variant	Artifact Change Variant	Block Change Variant	Conf. Block Change Variant
Average Execution Time	1,012s (σ=122s)	372s (σ=503s)	303s (σ=478s)	304s (σ=477s)
Median Execution Time	1,020s	10.5s	3.20s	3.62s
Found New Dead Variable Code Blocks	100%	100%	100%	100% of blocks referencing conf. option
Advantages	constant execution time, complete	fast, complete	fastest, complete	least number of full analyses, focus on actual bugs
Drawbacks	slow	requires full analysis rather often	requires detailed change analysis	not faster than block change variant, not complete

each provides a significant speed-up in comparison to the non-incremental variant and to each other, this does not hold for the configuration block change variant. Table 1 shows that the configuration block change variant actually takes more execution time on average and in median compared to the block change variant, although it uses the most fine-grained change identification. This is due to the additional analysis effort for identifying blocks that reference a configuration option in combination with a low performance gain (cf. Section 6.4). Hence, the answer to research question **RQ2** is that the granularity of change identification significantly impacts the performance of the analysis up to a certain point, where additional identification efforts counterbalance potential speed-ups.

The overhead of performing our change identification in incremental verification is extremely small, which answers research question **RQ3**. The preparation step responsible for this identification in all incremental variants requires on average only between 0.03% and 0.07% of the overall execution time. Hence, there is no notable negative performance impact, when switching to an incremental verification approach.

From a practitioner's point of view, our incremental variants are most of the time so fast, that they could easily be integrated either within the development environment or as part of a continuous integration cycle. Moreover, the artifact change and block change variants have no drawbacks in relation to the accuracy of the results of the dead variable code block analysis, as shown in Table 1. Both of them always detect whether a commit introduces new dead variable code blocks. However, as the block change variant is the fastest of all three variants, it should be the default choice. A more situation-specific option is the usage of the configuration block change variant. It is comparably fast to the block change variant, but allows for a more specific view on dead variable code blocks. Developers sometimes intentionally introduce dead variable code blocks, for example by using an `#if 0`, as explained in Section 2.1. These statements can be filtered out using the configuration block change variant, to focus on the variable blocks that are dead due to actual bugs.

In summary, our approach to incremental SPL verification provides significant performance improvements resulting from a rather simple change analysis. This simplicity originates from the usage of unmodified change information from commits and a change identification algorithm matching individual file lines against certain expressions. In particular, we do not need any preprocessing of change information to create a specific model of these changes to apply our approach to. However, this simplicity also introduces drawbacks in some situations of our performance analysis. Changes to the variability model or the build model always require a full re-analysis due to their cross-cutting nature. While a deeper analysis and incremental treatment of the variability model might be possible, the overhead would certainly be very significant and might counterbalance a lot of potential improvement. Hence, we did not pursue this further for this paper, but keep it as a subject for future work.

8 Threats to Validity

The performance analysis in this paper considers a sequence of 440 commits of the Linux kernel repository, which is a threat to *internal validity*. It only represents a subset of the available Linux kernel commits (changes) developers typically apply in practice. In Section 5, we discussed in detail why we selected exactly this sequence and explained our actions to mitigate this threat. In particular we compared the results of our comprehensive Linux evolution analysis presented in Kröher et al. (2018c) with the results of the same analysis

for the commit sequence used in this paper (cf. Section 5). While the commits in this sequence introduce 8% to 21% more changes to information relevant for the dead variable code analysis, our incremental approach will perform even better on the large-scale of the commits as fewer relevant changes reduce the analysis time of the incremental variants.

A threat to *construct validity* arises from our definition of relevant changes and their impact on the analysis (cf. Section 3.2): We do not provide a (formal) proof that the effects of those changes are correct or complete. However, the determination of the relevance of these changes relies on our observation in previous work (Kröher et al. 2018c) as well as the general process and the goal of the dead variable code analysis. On this basis, the defined changes and their impact are reasonable and fit the purpose of this paper. A related threat to *construct validity* concerns the correct identification of relevant changes, both, from a conceptual as well as a technical point of view. As we reuse the concepts and implementations for this purpose, as presented in Kröher et al. (2018c), we rely on a conceptually and technically mature foundation.

The definition of the general steps of an (incremental) analysis (cf. Section 4) also threatens *construct validity*. They determine the measuring points for the duration of particular tasks of the analysis. Hence, a different definition will change these detailed timings. Indeed, this does not affect the overall results in terms of the general performance of the individual analysis variants per commit. These results are independent from the steps as they span the entire runtime from start to end. Further, we do not change the core analysis algorithm between incremental and non-incremental variants (cf. Section 4.4), which would be another threat to *construct validity*.

The commit sequence of the performance analysis is not only a threat to internal validity, but also affects *conclusion validity*. In general, the more a commit changes relevant artifacts and information, the slower our incremental analysis variants will be. However, this behavior is natural for any analysis, which investigates changes, and no specific property of our approach. Another threat to *conclusion validity* is the single execution of our performance analysis. The reason for this lack is the time needed to execute the non-incremental dead variable code analysis, which takes multiple days even on our server hardware impeding multiple repetitions in time.

The approach, results, and discussions presented in this paper are limited to the dead variable code analysis, which is a threat to *external validity*. However, we already transferred our approach to another analysis[6] and are confident that also related analyses, like the feature-effect (Nadi et al. 2015) or configuration mismatch analysis (El-Sharkawy et al. 2017), benefit from it. In contrast, analyses relying on detailed code semantics, like type-checking (Kästner et al. 2011), are out of scope.

Finally, the selection of the Linux kernel and its commits for our performance analysis threatens *external validity*. We discussed above that the performance gain significantly depends on the changes introduced by commits. Hence, a different SPL with a potentially different philosophy of committing changes may reveal different results.

9 Related Work

In this paper, we present an approach and evaluate the performance of an incremental SPL verification approach, which considers changes to code, build, and variability model artifacts to reduce the analysis effort. Our approach is based on two key concepts: the ability to decide

[6] https://github.com/KernelHaven/ProblemSolutionSpaceMapperAnalysis

whether a change affects a part of the SPL relevant to an analysis and the ability to trigger only as much of the re-verification process as necessary. The first key concept can be generalized as *change impact analysis* (Bohner 1996). The second key concept belongs to the idea of *incremental* or *regression-based analysis*, with the goal of saving analysis time (Thüm et al. 2019). We structure our discussion of related work along these two key concepts in the remainder of this section. For this purpose, we create several topic clusters and introduce the belonging related work in the respective first paragraph of each cluster. We then proceed to discuss the similarities and differences of these works in the following paragraphs. We focus on those approaches proposed in the SPL domain, as we are specifically interested in SPL analyses. Furthermore, we do not discuss general incremental or regression-based approaches, like regression testing, as these are out of scope for our approach (cf. Section 1).

Change impact analysis for SPLs via slicing: One well-established approach to compute the possible impact of a change is *program slicing* (Weiser 1981). It reduces an artifact to a relevant slice, typically based on data- and control-flow analysis. Recent approaches propose variability-aware program slicing (Angerer et al. 2019; Gerling and Schmid 2019), slicing of component-based SPLs (Yazdanshenas and Moonen 2012), as well as slicing of variability-related models, like feature models (Schröter et al. 2016; Krieter et al. 2016; Acher et al. 2011a, b), delta models (Lity et al. 2015, 2016b), or product family models (Sabouri and Khosravi 2014, 2011a, b).

These approaches are all limited to analyzing one specific type of artifacts, while our approach does not share this limitation. Instead, our example of the dead variable code analysis requires the combined consideration of code, build, and variability model artifacts. Furthermore, none of these approaches is designed for incremental analysis, resulting in the need for a complete re-analysis of all artifacts after one artifact is actually changed. Finally, slicing could only help to investigate the impact of a change, but the discussed approaches do not further utilize their results. In contrast, we use our change impact analysis to derive the necessary re-analysis effort for the dead variable code analysis. As slicing itself is rather costly, for example the variability-aware code slicer by Gerling and Schmid (2019) takes hours to set up the data- and control-flow analysis of the Linux kernel, we decided not to reuse an existing approach, but instead use a much simpler and faster implementation for our change impact analysis.

Change impact analysis for SPLs via other artifact-specific analyses: In addition to slicing, there are further approaches from the SPL domain to calculate the impact of a change based on the analysis of one specific type of artifact, like the architecture (Díaz et al. 2011), the variability model (Dintzner et al. 2015; Lity et al. 2016a; Paskevicius et al. 2012) or the code (Ribeiro et al. 2014).

The approach by Díaz et al. (2011) proposes an SPL architecture meta-model that visualizes an explicit mapping among features from a feature model, requirements, and design decisions. For example, if a requirement changes, the meta-model allows to see which design decisions and features are affected by that change.

Paskevicius et al. (2012) describe the idea of change impact analysis for features in a feature model. If one feature is changed, for example, it is removed from the model, their approach conducts three different analyses: check whether the model is still valid, identify other features affected by the change, and calculate some metrics of the model, like the number of contained features.

Dintzner et al. (2015) expand the idea of feature change impact analysis to multi-SPLs, which are SPLs that consists of multiple SPLs and therefore have multiple variability models. They use an enhanced feature model to allow a change impact analysis over those models. For this purpose, they add configuration information to each feature model, so that whenever a

feature changes, the impact of that change to other configurations is visible, allowing to effectively reason about cross feature model relationships.

The approach by Lity et al. (2016a) proposes a similar approach to feature change impact analysis, but for delta models, which are a specific type of variability model used for modeling delta-oriented SPLs. In contrast to annotative SPLs, like the Linux kernel, delta-oriented SPLs explicitly define and model the differences and commonalities between variants as deltas, which describe the required change operations, like additions or removals, to derive a certain variant. Lity et al. enhance the existing delta model with an additional level of delta information, called *higher-order deltas*, that model changes like addition or removal of a delta through evolution over time. This allows to reason about the impact of such a change to the delta model and related variants.

In contrast, Ribeiro et al. (2014) focus on change impact analysis of code changes. They introduce *emergent interfaces*, that allow developers to see the impact of their changes to affected features. Based on data-flow analysis, their tool shows hints regarding affected features directly in the integrated development environment, to make developers aware of the change impact on other features.

All of the approaches in this category discussed above are limited to one specific type of artifact, similar to the approaches of the previous slicing category. They are therefore not applicable to dead variable code analysis, which requires the analysis of the combination of code, build, and variability model artifacts. Furthermore, these change impact analyses could only help to investigate the impact of a change, but do not further utilize their results. In contrast, we use our change impact analysis to derive the necessary re-analysis effort for the dead variable code analysis. Additionally, all of these approaches come with an information overhead, like mapping to requirements or delta information, needed for their specific goal, but not required for our approach. Finally, they provide information only on the level of affected features, but not for individual code blocks. As a consequence, we decided not to reuse an existing approach of this category, but rely on our custom implementation that is tailored to a block-based change impact analysis.

Change impact analysis for SPLs via analysis of multiple artifact types: Michalik and Weyns (2011) propose a variability meta-model that is based on a feature model with additional information like mapping to the implementation artifacts, derived products, and environment properties. They address a specific use case where a change in an artifact needs to be propagated to already deployed and running products which cannot be taken offline for maintenance. Their model therefore needs to analyze the change impact on these products, specifically, if the change is valid for a product in its current configuration. In contrast to the approaches discussed in the previous paragraphs, this approach is the first that allows a change impact analysis over several artifact types.

Their meta-model is partly overlapping with our approach, as we also need information about the mapping from variability model to code artifacts. However, we additionally need information about the build model, which does not exist in Michalik et al.'s approach. Furthermore, their meta-model has several additional entities like environment properties or mapping to products and their configuration, which is not of interest for a dead variable code analysis. Finally, the proposed change impact analysis could only help to investigate the impact of a change, but does not further utilize its results. In contrast, we use our change impact analysis to derive the necessary re-analysis effort for the dead variable code analysis. As there is no existing implementation for Michalik et al.'s approach and the meta-model is not appropriate for our change impact analysis, we decided not to reuse their approach. For the same reason, a comparison of their approach with our approach is not possible in practice.

From a theoretical point of view, we must assume that their approach would not scale to large-scale SPLs, like Linux. For example, if their approach would be applicable to Linux, building the respective meta-model would become practicable infeasible due to the range of products (Linux kernel variants), which is both unknown and extremely large.

Change impact analysis for SPLs via history analysis and heuristics: Another point of view on change impact analysis in the SPL domain is the idea of deriving possible impacts based on the evolution history of a SPL. These approaches can be categorized into two target applications: either detect undocumented relations of artifacts or features (Hamza et al. 2018; Dintzner et al. 2018), or categorize changes into patterns according to their potential impact (Borba et al. 2012; Neves et al. 2015, 2011; Sampaio et al. 2016; Gomes et al. 2019; Teixeira et al. 2020).

Hamza et al. (2018) propose a tool called *CIAHelper* for a combined analysis of code and variability model artifacts as well as the version history of delta-oriented SPLs to identify all products affected by a change request. CIAHelper constructs a dependency graph by first parsing the code files to an abstract syntax tree, which is then annotated with additional information from the delta-model, like delta information or mapping to products. Next, the approach analyzes the whole commit history of the SPL to find connections among the artifacts represented in the graph. The idea is that whenever artifacts are changed together in the same commit, this indicates a possible connection. These possible connections are then added as probabilities to the dependency graph.

The dependency model by Hamza et al. is partly overlapping with our approach as it also contains information about the mapping from variability model to code artifacts. However, necessary information about the build model is not included in their approach. Additionally, their meta-model includes several additional entities, like delta information or mapping to products, which is not of interest for a dead variable code analysis. Furthermore, a key part of Hamza et al.'s approach is the idea of adding heuristics to approximate the possible impact of a change. While this is an interesting approach, their evaluation shows that precision and recall of their change impact analysis is improvable. This is in conflict with the goal of a correct and precise analysis, which our approach is able to conduct.

On top of that, CIAHelper is built for delta-oriented SPLs and currently only works for Java-5-based implementations. This is in contrast to our use case of annotative SPLs in general and, in particular, the Linux kernel. Finally, the proposed change impact analysis could only help to investigate the impact of a change, but does not further utilize its results. In contrast, we use our change impact analysis to derive the necessary re-analysis effort for the dead variable code analysis. As a consequence, an empirical comparison of CIAHelper and our approach, as well as reusing CIAHelper, is not possible. If a variant of the approach would be constructed, which would be applicable to Linux, it would still have the problem of not ensuring 100% correctness as opposed to our approach.

Dintzner et al. (2018) present their *FEVER* tool, which analyzes the change impact on features across code, build, and variability model artifacts by analyzing the commit history of the SPL. Their main research interest is in the co-evolution of different artifacts, where connected artifacts, for example, a variability model artifact defining a feature and a code artifact implementing that feature, are evolved separately from each other. For this purpose, FEVER extracts artifact-specific change information from all commits of the evolution history and stores it in a graph database. Based on several heuristics, these changes are categorized and mapped to features, creating a feature *TimeLine*. This allows to investigate the change history from a feature perspective, where all changes affecting a particular feature are clustered together.

Similar to our approach, FEVER analyzes changes to code, build, and variability model artifacts and is also evaluated on the Linux kernel. However, FEVER uses heuristics to categorize changes into 21 different types, like edits to code blocks or removals of mappings. On the one hand, this allows for semantic reasoning about the changes. On the other hand, it is not as precise and reliable as our approach. Their evaluation shows that only 87.2% out of 810 commits are categorized correctly. In contrast, our approach is able to detect 100% of the relevant changes per analysis variant, while also distinguishing among different change categories. However, we are not interested in the kind of changes, like additions or removals, but whether a change affects an artifact or its content relevant for an analysis. This allows to ignore irrelevant changes, like changed help texts in variability mode artifacts, while guaranteeing correct results of the re-verification process.

Another main difference between FEVER and our approach is the scope of the history analysis. While our approach is regression-based and targets the analysis of one commit at-a-time, FEVER analyzes the whole history at once and is not built for incremental updates. Additionally, FEVER's analyses are also not designed to achieve high performance regarding execution times. This is also reflected in its evaluation, which is focused on accuracy and realism of the results, but does not mention any performance data. Lastly, the proposed feature-oriented change impact analysis could only help to investigate the impact of a change, but does not further utilize its results. In contrast, we use our change impact analysis to derive the necessary re-analysis effort for the dead variable code analysis. As a consequence, an empirical comparison of FEVER and our approach, as well as reusing FEVER, is not possible. If such a comparable approach would be constructed based on their ideas, it would be insufficient to guarantee correctness and completeness due to the heuristic nature of their approach. However, this is what we consider fundamental to our approach.

Neves et al. (2011, 2015) provide a set of *evolution templates* that categorize changes to code, build, and variability model artifacts into different patterns. Based on these patterns, they decide whether a change to a SPL is considered *safe* or *unsafe* with respect to the product line refinement theory defined by Borba et al. (2012). A safe change does not affect the existing products, for example by adding a new optional feature. In contrast, an unsafe change affects existing products, for example by removing a mandatory feature. Sampaio et al. (2016) extend this set by partially safe templates that only affect a part of the existing products, for example by removing a feature that is only present in some of the products and has no effects on other features. Gomes et al. (2019) conducted an empirical study on the Soletta SPL to investigate how often those safe or partially safe changes occur in practice. While earlier approaches of the (partially) safe categorization relied on manual analysis, Gomes et al. used a combination of FEVER (Dintzner et al. 2018) and manual analysis to identify the patterns. Teixeira et al. (2020) proposed the most recent update to evolution templates: a set of algebraic laws that could help to automate the analysis of the build model with regard to safe or unsafe changes.

Similar to our approach, the idea of evolution templates allows to analyze changes to code, build, and variability model artifacts and is also applicable to the Linux kernel. However, the categorization of changes with regard to their potential impact on existing products differs from our goal of reducing the re-verification effort after a change. Furthermore, we are not interested in the impact of a change to existing products, but whether a change affects an artifact or its content relevant for an analysis. These diverging goals result in two completely different approaches: While evolution templates are a formalized notion that still requires manual analysis (Teixeira et al. 2020), our approach is focused on fast practical usage and therefore fully automated. Finally, the proposed idea of evolution templates could only help to

investigate whether a change affects existing products, but does not allow to further utilize its results. In contrast, we use our change impact analysis to derive the necessary re-analysis effort for the dead variable code analysis. As a consequence, using the idea of evolution templates for our change impact analysis is not possible.

Incremental SPL analysis: The domain-specific language *IncA* by Szabó et al. (2016) is the only approach that specifically targets the implementation of incremental analysis in general. It enables their definition and execution within an integrated development environment that provides a change-tracking plug-in. Based on graph pattern matching and the program's abstract syntax tree, the approach re-analyzes the relevant parts of the program after a change. To the best of our knowledge, this is the only other approach that explicitly aims at a performance improvement of SPL analyses during evolution.

However, the applicability to SPLs in general is limited, as IncA depends on graph representations of the artifacts being subject to the analysis. For example, IncA is only able to work with software written in mbeddr C, which is a specific C dialect that has its own variability implementation mechanism.[7] An application to software not relying on this dialect, like the Linux kernel, requires a fitting graph representation of the code, build and variability model artifacts, as well as their relations among each other. This is not easily doable, because representing and analyzing un-preprocessed C-code and a Kconfig-model are both challenging endeavors, which are still not possible in a sound and complete way (Lüdemann and Koschke 2015; El-Sharkawy et al. 2015; Gerling and Schmid 2020). Furthermore, the scalability of IncA for large SPLs is still in question, as it was previously only evaluated on smaller software projects and more recent analyses revealed open scalability issues (Szabó et al. 2021). As a consequence, we must regard this approach as being not suitable to the problem we address in this study.

10 Conclusion

We presented an approach for the incremental verification of SPLs using the example of a change-aware dead variable code analysis. The approach is the first one, which solely relies on identifying changes in raw change information (unmodified commit data) to speed-up SPL verifications. It determines a reduced set of elements that have to be re-analyzed by identifying and classifying their changes, while reusing the information of unaffected elements. We applied this approach in three different variants of the dead variable code analysis as well as the original, non-incremental variant to a sample of the Linux kernel history to compare their performances and to evaluate the benefits of our approach.

The results of our performance analysis show that our incremental approach significantly accelerates a dead variable code analysis. Our fastest incremental variant takes only 3.20 seconds or less for most of the changes, while the non-incremental variant takes 1,020 seconds in median. At the same time, the analysis results are 100% accurate and the introduced overhead is neglectable. Furthermore, we identified an unexpected cap regarding the benefits of the granularity of change identification: The more fine-grained configuration block change variant is not able to perform faster than the more coarse-grained block change variant. However, the configuration block change variant could be an interesting option for practitioners to focus on unintentionally introduced dead variable code blocks. In general, we think

[7] http://mbeddr.com/

that our incremental approach could be profitably introduced in continuous integration environments.

We plan to generalize our concepts for incremental SPL verification to develop a generic approach as future work. Our goal is to provide similar benefits in terms of performance and precision to a broader range of analyses, in particular, in the domain of family-based static analysis (Thüm et al. 2014). Further, we identified additional optimization potential in our current technical realization. The investigation and possible integration of other reduction concepts is also a future goal.

Acknowledgements This work is partially supported by the ITEA3 project REVaMP^2, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not of the BMBF. Moritz Flöter (second author) contributed to this paper as part of his former employment at the University of Hildesheim.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability https://github.com/SSE-LinuxAnalysis/IncrementalAnalysesEvaluation

Code Availability https://github.com/KernelHaven/IncrementalAnalysesInfrastructure and https://github.com/KernelHaven/IncrementalDeadCodeAnalysis

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

Acher M, Collet P, Lahire P, France R (2011) Decomposing feature models: Language, environment, and applications. In: 26th International conference on automated software engineering. IEEE Computer Society, Washington, pp 600–603

Acher M, Collet P, Lahire P, France R (2011) Slicing feature models. In: 26th international conference on automated software engineering. IEEE Computer Society, Washington, pp 424–427

Adams B, De Schutter K, Tromp H, De Meuter W (2007) The evolution of the linux build system. Electronic Communication of the European Association of Software Science and Technology 8:1–16

Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston

Angerer F, Grimmer A, Prähofer H, Grünbacher P (2019) Change impact analysis for maintenance and evolution of variable software systems. Automated Software Engineering 26:1–45

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6):615–636

Berger T, Kästner C (2016) KBuildMiner. https://github.com/ckaestne/KBuildMiner. Accessed 18 Aug 2021

Bohner S (1996) Impact analysis in the software change process: a year 2000 perspective. In: International conference on software maintenance. IEEE Computer Society, Los Alamitos, pp 42–51
Borba P, Teixeira L, Gheyi R (2012) A theory of software product line refinement. Theoretical Computer Science 455:2–30
Chastek G, Donohoe P, Kang KC, Thiel S (2001) Product line analysis: a practical introduction. Tech. Rep. CMU/SEI-2001-TR-001, ESC-TR-2001-001, Carnegie Mellon Software Engineering Institute
ComAn Team (2018) Commit Analysis (ComAn). https://github.com/SSE-LinuxAnalysis/ComAn. Accessed 14 May 2020
ComAnI Team (2019) Commit Analysis Infrastructure (ComAnI). https://github.com/CommitAnalysisInfrastructure. Accessed 14 May 2020
Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wąsowski A (2012) Cool features and tough decisions: a comparison of variability modeling approaches. In: 6th international workshop on variability modeling of software-intensive systems. ACM, New York, pp 173–182
Díaz J, Pérez J, Garbajosa J, Wolf AL (2011) Change Impact analysis in product-line architectures. In: 5th European conference on software architecture. Springer, Berlin, pp 114–129
Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D (2012) A robust approach for variability extraction from the Linux build system. In: 16th international software product line conference, vol 1. ACM, New York, pp 21–30
Dintzner N, Kulesza U, van Deursen A, Pinzger M (2015) Evaluating feature change impact on multi-product line configurations using partial information. In: 14th International conference on software reuse. Springer International Publishing, Cham, pp 1–16
Dintzner N, van Deursen A, Pinzger M (2018) Fever: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. Empirical Software Engineering 23(2):905–952
Eichelberger H, Schmid K (2013) A systematic analysis of textual variability modeling languages. In: 17th International software product line conference. ACM, New York, pp 12–21
Eichelberger H, Kröher C, Schmid K (2013) An analysis of variability modeling concepts: expressiveness vs. analyzability. In: 13th international conference on software reuse, Springer, Berlin Heidelberg, pp 32–48
El-Sharkawy S, Krafczyk A, Schmid K (2015) Analysing the Kconfig semantics and related analysis tools. In: 14th international conference on generative programming: concepts and experiences, ACM, New York, pp 45–54.
El-Sharkawy S, Krafczyk A, Schmid K (2017) An empirical study of configuration mismatches in Linux. In: 21st international systems and software product line conference, vol A. ACM, New York, pp 19–28
Epanechnikov VA (1969) Non-parametric estimation of a multivariate probability density. Theory of Probability & Its Applications 14(1):153–158
Gerling L, Schmid K (2019) Variability-aware semantic slicing using code property graphs. In: 23rd international systems and software product line conference, vol A. ACM, New York, pp 1–7
Gerling L, Schmid K (2020) Syntax-preserving slicing of c-based software product lines: an experience report. In: Proceedings of the 14th international working conference on variability modelling of software-intensive systems, pp 1–5
Git (2020) Git version control system. https://git-scm.com/. Accessed 14 May 2020
Git (2021) Git documentation reference - git-apply. https://git-scm.com/docs/git-apply, version 2.33.0. Accessed 18 Aug 2021
Godfrey M, Tu Q (2000) Evolution in open source software: a case study. In: International conference on software maintenance. IEEE Computer Society, Washington, pp 131–142
Gomes K, Teixeira L, Alves T, Ribeiro M, Gheyi R (2019) Characterizing safe and partially safe evolution scenarios in product lines: an empirical study. In: 13th international workshop on variability modelling of software-intensive systems. Article 15, ACM, New York, pp 1–9
Hamza M, Walker RJ, Elaasar M (2018) CIAhelper: Towards change impact analysis in delta-oriented software product lines. In: 22nd international conference on systems and software product line. ACM Press, Gothenburg, pp 31–42
Hellebrand R, Silva A, Becker M, Zhang B, Sierszecki K, Savolainen J (2014) Coevolution of variability models and code: an industrial case study. In: 18th international software product line conference. vol 1, ACM, New York, pp 274–283
Hunsen C, Zhang B, Siegmund J, Kästner C, Leßenich O, Becker M, Apel S (2016) Preprocessor-based variability in open-source and industrial software systems: an empirical study. Empirical Software Engineering 21(2):449–482
Israeli A, Feitelson D (2010) The linux kernel as a case study in software evolution. Journal of Systems and Software 83(3):485–501
Kästner C (2016) kconfigreader. https://github.com/ckaestne/kconfigreader. Accessed 27 Nov 2020

Kästner C, Giarrusso G P, Rendel T, Erdweg S, Ostermann K, Berger T, (2011) Variability-aware parsing in the presence of lexical macros and conditional compilation. In: 2011 ACM international conference on object-oriented programming systems languages and applications. ACM, New York, pp 805–824

KernelHaven Team (2019a) KernelHaven plug-in: incremental analyses infrastructure. https://github.com/KernelHaven/IncrementalAnalysesInfrastructure. Accessed 14 May 2020

KernelHaven Team (2019b) KernelHaven plug-in: incrementalDeadCodeAnalysis. https://github.com/KernelHaven/IncrementalDeadCodeAnalysis. Accessed 14 May 2020

KernelHaven Team (2020) KernelHaven. https://github.com/KernelHaven. Accessed 14 May 2020

Krieter S, Schröter R, Thüm T, Fenske W, Saake G (2016) Comparing algorithms for efficient feature-model slicing. In: 20th international systems and software product line conference. ACM, New York, pp 60–64

Kröher C, Schmid K (2017a) A commit-based analysis of software product line evolution: two case studies. Technical Report SSE 2/17/E, University of Hildesheim

Kröher C, Schmid K (2017b) Towards a better understanding of software product line evolution. In: Softwaretechnik-trends, gesellschaft für informatik e.V., Fachgruppe PARS, Berlin, Germany, vol 37, no 2, pp 40–41

Kröher C, El-Sharkawy S, Schmid K (2018) KernelHaven: An experimentation workbench for analyzing software product lines. In: 40th international conference on software engineering: companion proceeedings. ACM, New York, pp 73–76

Kröher C, El-Sharkawy S, Schmid K (2018b) KernelHaven: An open infrastructure for product line analysis. In: 22nd international systems and software product line conference, vol 2. ACM, New York, pp 5–10

Kröher C, Gerling L, Schmid K (2018c) Identifying the intensity of variability changes in software product line evolution. In: 22nd international systems and software product line conference, vol 1. ACM, New York, pp 54–64

Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in fourty preprocessor-based software product lines. In: 32nd ACM/IEEE international conference on software engineering, vol 1. ACM, New York, pp 105–114

Linux (2018) Kconfig language. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt. Accessed 14 May 2020

Linux (2019) Linux kernel makefiles. https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt. Accessed 14 May 2020

Lity S, Baller H, Schaefer I (2015) Towards incremental model slicing for delta-oriented software product lines. In: 22nd international conference on software analysis, evolution, and reengineering. IEEE Computer Society, Washington, pp 530–534

Lity S, Kowal M, Schaefer I (2016) Higher-order delta modeling for software product line evolution. In: 7th international workshop on feature-oriented software development. ACM, New York, pp 39–48

Lity S, Morbach T, Thüm T, Schaefer I (2016) Applying incremental model slicing to product-line regression testing. In: 15th international conference on software reuse: bridging with social-awareness. Springer, New York Inc, New York, pp 3–19

Liu YA, Stoller SD (2003) Eliminating dead code on recursive data. Science of Computer Programming 47(2): 221–242

Livengood S (2011) Issues in software product line evolution: complex changes in variability models. In: 2nd international workshop on product line approaches in software engineering. ACM, New York, pp 6–9

Lüdemann D, Koschke R (2015) From preprocessor-constrained parse graphs to preprocessor-constrained control flow. In: 2015 IEEE 15th international working conference on source code analysis and manipulation (SCAM), pp 211–220

Meinicke J, Thüm T, Schröter R, Benduhn F, Saake G (2014) An overview on analysis tools for software product lines. In: 18th international software product line conference, vol 2. ACM, New York, pp 94–101

Mendonca M, Wasowski A, Czarnecki K (2009) SAT-based analysis of feature models is easy. In: 13th international software product line conference. Carnegie Mellon University, Pittsburgh, pp 231–240

Michalik B, Weyns D (2011) Towards a solution for change impact analysis of software product line products. In: 9th working IEEE/IFIP conference on software architecture. IEEE Computer Society, Washington, pp 290–293

Mukelabai M, Nešić D, Maro S, Berger T, Steghöfer JP (2018) Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In: 33rd ACM/IEEE international conference on automated software engineering. ACM, New York, pp 155–166

Nadi S, Holt R (2012) Mining kbuild to detect variability anomalies in Linux. In: 16th European conference on software maintenance and reengineering. IEEE Computer Society, Washington, pp 107–116

Nadi S, Holt R (2014) The Linux kernel: A case study of build system variability. Journal of Software: Evolution and Process 26(8):730–746

Nadi S, Berger T, Kästner C, Czarnecki K (2015) Where do configuration constraints stem from? An extraction approach and an empirical study. IEEE Transactions on Software Engineering 41:820–841

Neves L, Teixeira L, Sena D, Alves V, Kulesza U, Borba P (2011) Investigating the safe evolution of software product lines. In: 10th ACM international conference on generative programming and component engineering. ACM, New York, pp 33–42

Neves L, Borba P, Alves V, Turnes L, Teixeira L, Sena D, Kulesza U (2015) Safe evolution templates for software product lines. Journal of Systems and Software 106(C):42–58

Paskevicius P, Damasevicius R, Štuikys V (2012) Change impact analysis of feature models. In: 18th international conference on information and software technologies. Springer, Berlin, Heidelberg, pp 108–122

Passos L, Teixeira L, Dintzner N, Apel S, Wasowski A, Czarnecki K, Borba P, Guo J (2016) Coevolution of variability models and related software artifacts - a fresh look at evolution patterns in the Linux kernel. Empirical Software Engineering 21(4):1744–1793

Ribeiro M, Borba P, Kästner C (2014) Feature maintenance with emergent interfaces. In: 36th International conference on software engineering. ACM, New York, pp 989–1000

Sabouri H, Khosravi R (2011) Efficient verification of evolving software product lines. In: 4th IPM international conference on fundamentals of software engineering. Springer, Berlin, Heidelberg, pp 351–358

Sabouri H, Khosravi R (2011) Reducing the model checking cost of product lines using static analysis techniques. In: 8th international symposium on formal aspects of component software. Springer, Berlin, Heidelberg, pp 296–312

Sabouri H, Khosravi R (2014) Reducing the verification cost of evolving product families using static analysis techniques. Science of Computer Programming 83:35–55

Sampaio G, Borba P, Teixeira L (2016) Partially safe evolution of software product lines. In: 20th International systems and software product line conference. ACM, New York, pp 124–133

Sat4j Team (2021) Sat4j. https://www.sat4j.org/. Accessed 18 Aug 2021

Schmid K, de Almeida ES (2013) Product line engineering. IEEE Software 30(4):24–30

Schröter R, Krieter S, Thüm T, Benduhn F, Saake G (2016) Feature-model interfaces: the highway to compositional analyses of highly-configurable systems. In: 38th international conference on software engineering. ACM, New York, pp 667–678

Silverman B (1986) Density estimation for statistics and data analysis. Monog Stat Appl Probab 26

Szabó T, Erdweg S, Voelter M (2016) IncA: a DSL for the definition of incremental program analyses. In: 31st IEEE/ACM international conference on automated software engineering. ACM, New York, pp 320–331

Szabó T, Erdweg S, Bergmann G (2021) Incremental whole-program analysis in Datalog with lattices. In: Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation, ACM, pp 1–15

Tartler R, Lohmann D, Sincero J, Schröder-Preikschat W (2011) Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In: 6th conference on computer systems. ACM, New York, pp 47–60

Teixeira L, Gheyi R, Borba P (2020) Safe evolution of product lines using configuration knowledge laws. In: Brazilian symposium on formal methods, Springer, pp 210–227

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

Thüm T, Teixeira L, Schmid K, Walkingshaw E, Mukelabai M, Varshosaz M, Botterweck G, Schaefer I, Kehrer T (2019) Towards efficient analysis of variation in time and space. In: 23rd International systems and software product line conference, vol B. ACM, New York, pp 57–64

Torvalds L (2020) Linux kernel source tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. Accessed 14 May 2020

VAMOS/CADOS Team (2015) Undertaker. https://vamos.informatik.uni-erlangen.de/trac/undertaker. Accessed 14 May 2020

Weiser M (1981) Program slicing. In: 5th International conference on software engineering. IEEE Press, Piscataway, pp 439–449

Yazdanshenas AR, Moonen L (2012) Fine-grained change impact analysis for component-based product families. In: 28th IEEE international conference on software maintenance. IEEE Computer Society, Los Alamitos, pp 119–128