A Slice-Based Estimation Approach for Maintenance Effort

Hakam W. Alomari  
Department of Computer Science  
Kent State University  
Kent, OH 44242, USA  
halomari@cs.kent.edu

Michael L. Collard  
Department of Computer Science  
The University of Akron  
Akron, OH 44325, USA  
collard@uakron.edu

Jonathan I. Maletic  
Department of Computer Science  
Kent State University  
Kent, OH 44242, USA  
jmaletic@kent.edu

Abstract—Program slicing is used as a basis for an approach to estimate maintenance effort. A case study of the GNU Linux kernel with over 900 revisions spanning 17 years of history is presented. For each revision a system dictionary is built using a lightweight slicing approach and encodes the forward decomposition static slice profiles for all variables in all the files in the system. Changes to the system are then modeled at the behavioral level using the difference between the system dictionaries of two revisions. The three different granularities of slice (i.e., line, function, and file) are analyzed. We use an XML formatted document to represent computed function change information. The retrieved information reflects the fact that additional knowledge of the differences can be automatically derived to help maintainers understand code changes. We consider the hypotheses: (1) that the structured format helps create traceability links between the changes and other software development artifacts. (2) This model is predictive of maintenance effort. The results demonstrate that the approach accurately predicts effort in a scalable manner.

Keywords—effort estimation; program slicing; software metrics; software maintenance

I. INTRODUCTION

Systems must be maintained so as to remain useful [1] and estimating the amount of effort for particular maintenance tasks is a key aspect for any system (closed or open). As systems grow, maintenance typically becomes more complicated and costly. Thus, the maintenance process should be well planned in advance through an accurate effort estimation of the maintenance tasks [2, 3]. Traditionally, maintenance effort is calculated using historical process and coarse-grained system information such as person hours, number of tasks, and system size [4]. The predictor variables used to estimate this value typically compose measures of the system size and complexity, productivity factors, as well as size and number of maintenance tasks [5].

Typically, an estimation process for maintenance effort contains three steps, as follows:

1) Extract maintenance data, such as maintenance effort (person-hours), number of maintenance tasks, system size.

2) Build and validate the maintenance-effort model. Conventionally, this is a mathematical model that represents the maintenance effort as a function of other software measures. The model should be validated against additional maintenance data.

3) Predict future maintenance effort using the maintenance-effort model.

While using maintenance-task information is very attractive for managers of a typical closed-source system, who have to estimate the effort required to maintain the system in terms of the number of developers, this approach is not that useful for larger corrective, adaptive, or perfective tasks during the system evolution of open-source system [3]. In this case, the effort of a maintenance period greatly depends on the amount of source-code changes made to generate a new software revision from an earlier operational revision. For open-source systems, this data is not recorded or documented [6]. Additionally, because of the nature and complexity of the maintenance tasks in open-source systems, there are many negatives to directly using effort-estimation models built on closed-source data. Hence, we cannot follow the same process to estimate maintenance effort. However, the availability of the source code and history allow for other measures that are related to the maintenance effort. To this end we introduce a maintenance effort estimation based directly only on source code. It entails computing the slice for all the variables in a system and modeling how the slice changes over time. Specifically, we identify and validate slice-based software measures and a corresponding process that can represent maintenance effort in open-source systems. We analyze 974 revisions of Linux kernel, and construct, validate, the indirect maintenance-effort model. The estimation approaches of maintenance effort are built and evaluated using residual-analysis statistics. Statistical measures include $R^2$, $\text{adjusted-}R^2$, $\text{PRED}_{25}$, $\text{PRED}_{30}$, $\text{MMRE}$, $\text{MdmRE}$, and $\text{SPR}$ [7, 8]. The prediction results are encouraging and the production of the estimate is very scalable.

The remainder of this paper is organized as follows. Section II discusses indirect maintenance-effort measures. Section III describes program slicing. Section IV introduces the slice-based metrics. Section V applying these metrics to the maintenance activities of the Linux kernel. Section VI presents the process of building the effort estimation models for open-source systems. The approach is evaluated in Section VII. Section VIII reviews related work followed by Section IX with the paper conclusions and some directions for future research.

II. INDIRECT MAINTENANCE EFFORT MEASURES

In this section, we describe two software measures that are related to maintenance effort and could possibly be used to
represent maintenance effort of open-source system. They are lag time and source code change. We need to consider whether lag time and/or source code change are a valid indirect maintenance effort measures.

1) Lag time: Each version of the system has its own release date. The lag time (measured in days) measures the time between the start and the completion of a maintenance task. The lag time includes the duration from the date when a base revision is released, until the date the evolved revision is released. The assumption here that the maintenance requests start when the base revision is released, and the tasks are completed when the evolved revision is released. That is, the lag time is the sum of the individual times for each maintenance task in a revision of a system.

The lag time data are available for most closed-source systems as well as some open-source systems. For example, the lag time data can be extracted from the defect tracking system, Concurrent Versions System (CVS), or change log [2]. Obviously, lag time is related to maintenance effort. That is, an increase in lag-time is expected to indicate an increase in maintenance effort, and vice versa. However, there could be some problems with using lag time as indirect maintenance effort. For example, using start date and closed date of a maintenance task, there is a risk of over reporting maintenance effort. For example, if a developer is sick for three weeks during the maintenance or it was Christmas and no one bothered to work on the system, the lag time is then over reported. In addition, the granularity of lag time is not accurate; one hour and 8 hours are both considered as one day. Finally, the importance of the bug controls the lag time period. Important bugs are usually handled immediately after the assignment of the task, while less important bugs may be ignored until the next revision. Therefore, using lag time to represent the maintenance effort in not accurate.

2) Source code change: We note that the maintenance effort for open-source systems is not given as the number of person-hours expended as the case in closed-source system [2, 4]. However, it has been argued [2, 9-12] that source-code changes in open-source systems can be used as an indirect measure for estimating maintenance effort. That is, the amount of source-code changes from revision k (base revision) at time t to revision k+1 (evolved revision) at time t+1 indirectly represents the effort spent maintaining the system from revision k to k+1.

A number of researchers have observed that the source-code change can be found using textual, syntactic, or semantic differencing [13]. For example, previous studies [2, 3, 10, 14], determine the source-code change between two consecutive revisions either from CVS logs, using some computer aided software (CASE) tools, or system utilities such as \textit{diff}. When a source code changes are submitted using the Software Configuration Management (SCM) tools (e.g., Subversion, CVS, and ClearCase) best practice is for developers to commit a brief explanation of the change into the change log, which is saved collectively with the source code deltas in the SCM repository. After many source code revisions of software project has been released, the change logs help maintainers to understand the evolution of the source code by providing details on the purpose of the change, and by whom. Unfortunately, the quality of change logs varies greatly. That is, it depends on the developer that submits the changes; how well she understands the source code, and how well she writes the change logs. Imprecise or blank change logs make it hard for system maintainers to understand the source code. Even when change logs are present; they are written in natural language and are typically format free.

For example, Chen et al. [15] discussed the limitations of using the change logs to detect source-code changes in three open-source case studies. He shows that up to 78% of changes made to the source code are omitted from the system’s change logs, and concludes that before using change logs as a research base for development and maintenance of open-source systems, experimenters should check carefully for errors and inaccuracies. Additionally, this tracking data is not always available. For example, the change logs for Linux kernel only started to be released after the major revision 2.4 (revision 2.4.1, January 29, 2001). That’s why Yu [2] in his study of the Linux kernel built two models to estimate the maintenance effort using the change logs for major revisions 2.4 and 2.5 only, with a total of 121 revisions. These facts will make it hard to build effort estimation models and traceability links, based on the information founded in the change logs.

Though the system change information computed at the textual level using SCM tools give us straightforward facts of the textual changes of a system in the new revision, no behavioral change facts are revealed. Textual changes such as changes to comments, variable renaming, code beautification, do not affect program behaviors, while other changes may affect various aspects of program behaviors.

A behavioral aspect of a system can be represented by a system slice that computes a program points that are affected by other program points. By using semantic-differencing approaches that depend on static-program analysis we can extract facts and other information from revisions of the system. For example, static program slicing can be applied to the available source code of the revision, focusing on selected aspects of semantics. This process removes from consideration parts of the program which are determined to have no effect upon the semantics of interest. It is possible to determine the parts with different behaviors by comparing the slices of the base and the evolved revisions with respect to corresponding points.

III. PROGRAM SLICING AND PROGRAM ANALYSIS

Program slicing is a widely used, and well-known, approach for understanding and detecting the impact of changes to software. The idea is simple, given a variable and the location of that variable in a program, a slice produces the other parts of the program that are affected by the variable. Slicing has been used successfully for many years for a wide variety of maintenance tasks [16-27]. The concept of program slicing was originally identified by Weiser [28, 29] as a debugging aid. He defined the slice as an executable program that preserved the behavior of the original program. Weiser’s algorithm traces the data and control dependencies by solving data-flow equations for determining the direct and indirect relevant variables and statements. Since that time, a large variety of different slicing techniques and tools have been
proposed. This large body of literature is well documented in a detailed survey on the vocabulary of program slicing [30].

The calculation of a program slice is, with few exceptions, based on the notion of a Program Dependence Graph (PDG) [31, 32] or one of its variants, e.g., a System Dependence Graph (SDG) [33]. Unfortunately, building the PDG/SDG is quite costly in terms of computational time and space. As such, slicing approaches generally do not scale well, and while there are some (costly) workarounds, generating slices for a very large system can often take days of computing time. Additionally, many tools are strictly limited to an upper bound on the size of the program they can slice because of memory constraints.

Our slicing approach [34, 35] addresses this limitation by eliminating the time and effort needed to build the entire PDG. In short, it combines a text-based approach, similar to Cordy’s [36], with a lightweight static analysis infrastructure that only computes dependence information as needed (aka on-the-fly) while computing the slice for each variable in the program. The slicing process is performed using the srcML [37, 38] format for source code. Source code is first converted to srcML and then a stream-oriented approach to compute the slice is performed. srcML augments source code with abstract syntactic information. This syntactic information is used to identify program dependencies as needed when computing the slice. srcML (SouRce-Code Markup Language) is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. The srcML format is supported with a toolkit, including src2srcml and srcml2src, which supports conversion between source code and the format.

We implemented our approach in a tool called srcSlice\(^1\). The approach was first introduced in [34], and there we conducted a small comparison study to the CodeSurfer tool from GammaTech Inc.\(^2\) After that in [35], we extended this evaluation to a total of 18 open source systems, along with making a number of enhancements to our algorithm and tool. The results of the comparison showed that the slices produced by our approach are very reasonable with respect to accuracy. It is also shown to be very efficient with regard to computational time. For example, it takes approximately 20 minutes (using a desktop machine) to convert the Linux kernel (~14 MLOC) into srcML and compute the slice for every variable in the entire system. To further demonstrate the scalability, we applied the tool to 17 years of revisions of the Linux kernel and present the results.

In our work, a program slice can be represented by a system dictionary instead of PDG/SDG. To detect system changes of two program slices, we compare their corresponding dictionaries. To make the comparison process efficient and make the comparison results reusable, we developed a system slice encoding (SSE) algorithm that encodes a program slice to a hash value, thereby allowing the slice hashes of a system to be used to detect behavioral changes across revisions. The slice hashes for revisions are stored in the system change information to identify behavioral changes across revisions. This information serves as complement to the change logs to help maintainers understand changes better. The representation, srcML, is an XML format that we used to represent the system change information, so the change information can be identified at different granularities (i.e., line, function, and file). The XML format also permits traceability links to be built between the change information and other software artifacts.

### A. Slice Profile and System Dictionary Construction

To compute a slice, certain dependence information is required. Unlike other slicing techniques, our algorithm does not rely fully on pre-computed data and control dependencies since they can require costly analysis, e.g., constructing the def-use chains in the existence of pointers. Instead, this is calculated as needed on the fly for the slicing variable while constructing the slice. The approach computes a slice profile that contains all the relevant statements, from all possible slices, over a given slicing variable \(v\). After the algorithm is applied, the slice profile associated with a variable \(v\) consists of the lines of code transitively affected by the value of \(v\) along control and data dependencies. We define our slicing criterion to consist of a file name, a function name, and a variable name. This slicing criterion is the triple \((f, m, v)\) where \(f\) is a file in the system, \(m\) is a function/method in the file \(f\), and \(v\) is a variable in the given function \(m\). This definition of a slicing criterion does not require a precise reference to a statement number. This concept of slicing is used by Gallagher et al. [39] and is referred to as a decomposition slice.

Our approach allows decomposition slices to be constructed with respect to a set of slicing criteria. Rather than just a single variable of interest within the original program, our definition can retrieve the slices for all the variables inside a given function by modifying the slicing criterion to \((f, m)\). Moreover, the slicing criterion \((f)\) can be used to find all the slices of all variables in all functions in a given file. A system dictionary is built, referred to as \((F, M, V)\), and includes all files in the system, all functions in each file, all variables in each function, and all global variables in the system. Each entry of the system dictionary is a slice profile with the following structure:

- file, function, and variable names;
- @index, an index of each variable as declared in order in the function;
- slines, a list of lines that comprise the slice;
- cfunctions, a list of functions called using the slicing variable;
- dvariables, a list of variables that are data dependent on the slice variable;
- pointers, a list of aliases of the slicing variable; and
- controledges, a list of all possible control-flow edges of the slicing variable.

We now present a definition of our slicing criterion and how a slice is computed using the criterion.

**Definition 1 (Forward Decomposition Slice)** A forward decomposition slice \(ds\) of a program \(p\) is constructed with

\(^1\) Available for download at www.srcML.org under General Public License.

respect to a given file $f$, a given function $m$ in $f$, and a given variable $v$ in $m$. It consists of the union of the static forward slices (denoted by $sfs$) constructed for the criteria $\{(v_1, s_1), \ldots, (v_k, s_k)\}$, where $\{s_1, \ldots, s_k\}$ is the set of statements in $p$ that assign to $v$. It is defined as:

$$ds(f, m, v) = \bigcup_{s \in \{v_1, \ldots, s_k\}} sfs(v, s)(p).$$

This definition can be generalized to cater to a set of variables, functions, and files. This yields a definition for the general forward decomposition slice.

**Definition 2 (General Forward Decomposition Slice)** A general forward decomposition slice of a program $p$ is constructed with respect to the following slicing criteria $(f, m)$, $(f)$, and $(F, M, V)$, where $F = \{f_1, f_2, \ldots, f_l\}$ is the finite set of files in $p$, $M = \{m_1, m_2, \ldots, m_k\}$ is the finite set of methods for each $f \in F$, and $V = \{v_1, v_2, \ldots, v_k\}$ is the finite set of variables for each $m \in M$. The general decomposition slice for all variables (i.e., set $V$) inside a given function $m$ is formed by:

$$mds(f, m) = \bigcup_{i=1}^d ds(f, m, v_i),$$

where $\{v_1, \ldots, v_k\}$ is the set of variables assigned to by $m$.

The general decomposition slice for all variables in a given file $f$ is given by:

$$fds(f) = \bigcup_{i=1}^y mds(f, m_i).$$

The general decomposition slice for all variables in all the files $F$, and all global variables in the system is given by:

$$gds(F, M, V) = \bigcup_{i=1}^j fds(f_i).$$

---

**Figure 1.** (a) Sample source code, (b) system dictionary with two slice profiles for the source code in (a). The final slice for $sum = [2, 5, 8]$ and the final slice for $i = [3, 4, 5, 6, 8, 9]$ after considering dependencies.

Let us now look at a simple example. The approach works much like a programmer would compute a slice in their head. Figure 1 presents a small program (a) along with the final system dictionary (b). The dictionary includes two slice profiles, one for each of the variables $sum$ and $i$. The $\@index$ represents the position of variables as declared in the function. In this way, we can deal with variables of the same name within the same scope. The slice profiles are computed by examining each line starting from the beginning (line 1) and determining the forward slice. Definition-use chains are followed along with forward control dependencies. The profile for $sum$ is created first as it is encountered in line 2 ($slines(sum) = \{2\}$). Then the profile for $i$ is created in line 3 ($slines(i) = \{3\}$). The two profiles are updated as follows for the given line number:

4: $slines(sum) = \{2\}$; $slines(i) = [3, 4]$, controledges($i$) = {(3, 4)}
5: $slines(sum) = [2, 5]$, controledges(sum) = {(2, 5)}; $slines(i) = [3, 4, 5]$, dvariables(i) = {sum}, controledges(i) = {(3, 4, 5)}
6: $slines(sum) = [2, 5]$, controledges(sum) = {(2, 5)}; $slines(i) = [3, 4, 5, 6]$, dvariables(i) = {sum}, controledges(i) = {(3, 4, 5, 6)}

7: $slines(sum) = [2, 5, 8]$, controledges(sum) = {(2, 5), 8); $slines(i) = [3, 4, 5, 6]$, dvariables(i) = {sum}, controledges(i) = {(3, 4, 5, 6)}
8: $slines(sum) = [2, 5, 8]$, controledges(sum) = {(2, 5), (2, 8), (3, 8), (3, 4, 5, 6)}

9: $slines(sum) = [2, 5, 8]$, controledges(sum) = {(2, 5), (2, 8), (3, 8), (3, 4, 5, 6)}

These are the slice profiles for each variable, and the complete slice is then computed by finding the control-flow edges and then taking the union of the $slines$ with the slice profiles of the $dvariables$, $cfunctions$, and $pointers$, minus any lines that are before the initial definition of the slice variable (i.e., the set $\{1, \ldots, \text{def}(v) - 1\}$). Thus, because $sum$ is data dependent on $i$, the complete slice for $i = slines(i \cup slines(sum))$ – {1, 2}. This comes out to $\{3, 4, 5, 6, 8, 9\}$. This final computation can be carried out for all variables via a single pass through the dictionary. Due to space limitations, we only explained the slicing algorithm for a simplified language that supports just assignment, while statements, and scalar variables. In our implementation [35], we developed a full revision of the slicing algorithm that works on a variety of C/C++ programs and language features.

**B. Encoding slicing information**

Our system slice encoding (SSE) algorithm works on slice profiles represented by the system dictionary defined by Alomari et al. [34]. The basic process of the SSE algorithm starts with a single pass through the system dictionary, encoding each slice profile to a string value, which is then fed to a hash algorithm to produce the final results, the hashed slice encoding. There are two steps in the SSE algorithm.

**Step 1 of the SSE algorithm: Encode the slice profiles**

The complete slice for each slicing variable after taking the union of all related slice profiles will have the following format structure:

- **variableName**: @index; **slices**: [2, 5, 8]; **SliceProfile(sum)** = @index(1), **slices**: [2, 5, 8]; **SliceProfile(i)** = @index(2), **slices**: [3, 4, 5, 6, 9]; **dvars** = [sum]

For the example program in Figure 1, the **variable encoding string** for each slicing variable is as follows: $sum; \@1[2, 5, 8]$ and $i; \@2[3, 4, 5, 6, 8, 9]$. And the **function encoding string** = **main**; [2, 3, 4, 5, 6, 8, 9]. The file encoding string = **fileName**; {fds(f)i}. Finally, the **System encoding string** = **systemName**; {gds(F, M, V)}. 

---

Alomari, Collard, Maletic

---

4
**Step 2 of the SSE algorithm: Hash the string value**

This step maps the encoding string from Step 1 to a hash value using the MD5 hash algorithm [40]. The MD5 hash for the resulting encoding string from Step 1 is:

The variable encoding string for the variable `sum` = 6e9ed3c2a88b623c05347ae687d289. The variable encoding string for the variable `i` = e20426ade165seaac2a9c09d261. The function encoding string for the function `main` = f65571d34742bf9a65e53e9a6640d2b.

**IV. SLICE-BASED METRICS**

Many existing software metrics are computed only using syntactic information of the code and use that to model semantic information. For example, cyclomatic complexity is computed by counting the number of branch (i.e., conditionals) to infer semantic complexity. Semantic information is much more difficult to derive and model. For example, a semantic change in one function might create a ripple effect among other functions. In a maintenance context, the effort estimation is a function of the code that is to be (was) changed. To help identify such problems, program slicers are often applied and are a valuable tool in determining size effects.

In the context of effort prediction, Ramil et al. [41] stated that one may start the investigation of building an effort model by obtaining empirical data and by estimating from such data a productivity function $f()$. The final empirical data involved in the estimation of $f()$ is represented in the following equation:

$$\Delta \text{effort} (t, t+1) = f(\text{activity} (t, t+1)) + \text{error} (t, t+1),$$

where, $\Delta \text{effort} (t, t+1)$ represents the estimated effort. That is, the effort required evolving the system from interval $t$ to $t+1$. The activity $(t, t+1)$ represents the amount of work accomplished over the time interval. Finally, $\text{error} (t, t+1)$ is the modeling error. In addition, Ramil mentioned that the appropriate way to measure the activity $(t, t+1)$ in the continuing evolution context is by measuring some indicators of source-code change, e.g., lines of source code (LOC) or function points (FP) [42]. However, other metrics can also be extracted from source code with different degrees of granularity. Once the productivity function $f()$ is determined, the resultant model may be used to predict future maintenance effort requirements.

**A. Measuring Slice-Based Metrics**

To compute system behavioral change information across the entire revision history of a project, we check out every pair of consecutive revisions of the project from its subversion repository, use src2srcml to convert the source code into srcML format, use srcSlice to obtain the slice profiles with respect to the slicing variables in each revision, and apply the SSE algorithm on them. We compare the slice hashes of each variable, function, and file in the later revision with the corresponding hashes in the prior revision to find the behavioral changes. Finally, we save the system behavioral change information for each revision in a MySQL database. Figure 2 shows the project architecture and the flow of data among components.

In order to build the slice-based maintenance-effort model, for each of the 974 versions of the Linux kernel, we extract ten measures from the source-code repository and the changes between slice profiles. These measures are described in Table I. In these measures, $\Delta \text{sliceSize}$, $\Delta \text{function}$, and $\Delta \text{file}$ are the extent of change between the two versions, and hence could be used to indirectly represent maintenance effort. We explain each of the slice-based metric items as follows.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sliceSize</td>
<td>Total slice size measured in LOC</td>
</tr>
<tr>
<td>$\Delta \text{sliceSize}$</td>
<td>Indirect maintenance effort at the system level, measured as the difference of slice sizes</td>
</tr>
<tr>
<td>$\Delta \text{function}$</td>
<td>Indirect maintenance effort at the function level, measured as the number of functions which contain modified slices</td>
</tr>
<tr>
<td>$\Delta \text{file}$</td>
<td>Indirect maintenance effort on the file level, measured as the number of files which contain modified slices</td>
</tr>
<tr>
<td>LOC</td>
<td>Total size of the system measured in LOC</td>
</tr>
<tr>
<td>files</td>
<td>Total size of the system measured in number of files</td>
</tr>
<tr>
<td>LOC-g</td>
<td>Difference between LOCs for two consecutive versions</td>
</tr>
<tr>
<td>files-g</td>
<td>Difference between files for two consecutive versions</td>
</tr>
<tr>
<td>lag-time</td>
<td>Time duration between two versions in days</td>
</tr>
<tr>
<td>Scoverage</td>
<td>The slice coverage, the slice size relative to LOC</td>
</tr>
</tbody>
</table>

The first metric that we introduce is sliceSize, the slice size measured in LOC. For an individual slice this is just the $d_s$ value measured at Section III. For a function and file, we summed the sliceSize of all slices of the variables inside the function ($mds$) and file ($fds$), respectively. For a system, we sum the sliceSize for all slices in the system ($gds$). For two versions of the system, we can then measure the difference between the sliceSizes. This forms a new metric $\Delta \text{sliceSize}$. This gives us some idea of the growth of the system in terms of complexity, i.e., the $\Delta \text{sliceSize}$ metric represents the increase or decrease in the number of impacted statements. Additionally, the number of modified slices between two versions is used to introduce two more metrics, $\Delta \text{function}$ and $\Delta \text{file}$. The metric $\Delta \text{function}$ is the number of functions which contain modified slices, and the metric $\Delta \text{file}$ is the number of files which contain modified slices. These two metrics indicate how much the changed statements in a slice profile depend on each other by intra-procedural or inter-procedural control or data dependencies. A high $\Delta \text{function}$ value indicates more logically
complex code, and a high $\Delta file$ value may indicate that the changes in the system were very broad.

By comparing the slice size ($sliceSize$) to the system size (LOC), we can measure the slice coverage using the $\overline{coverage}$ metric [43]. This metric represents the active portion of the system and is included as a factor of maintenance activity. It is related to different types of maintenance because each type has a different effect on the maintenance effort [3, 14, 44]. For example, corrective maintenance requires more effort than the other types of maintenance. Finally, since the size of the system is shown in the literature to be related to maintenance effort, we also extract the LOC and number of files. For two versions of the system, we can then measure the difference. This represents the system growth and forms the metrics LOC-g and files-g.

B. Maintenance Effort Metrics

As previously said, since we are looking at open-source system, the developers did not log hours for their maintenance effort. Instead of measuring maintenance effort using person-days/hours, our proposed model measures maintenance effort by measuring the amount of source-code changes using source-code slicing. To approximate maintenance effort as change-days, we scrutinized our slicing data-set to see which proxy measures for maintenance effort we could find. We settled on three slice-based proxy measures.

$\Delta sliceSize(i, i - 1)$, change in lines of code of slice: we define change here as the total number of lines of code that were changed in the slice of revision $i$ with respect to previous revision $i - 1$ of a system.

$\Delta function(i, i - 1)$ and $\Delta file(i, i - 1)$, functions/files containing modified slices, respectively: this counts the total number of slice changes that were needed to resolve an issue affecting a function/file. The motivation here is that as function/file complexity grows, so does the number of slices that will be impacted for a given source code change. We define the maintenance time $t(k)$ of revision $k$ as the number of days elapsed between the release of revision $k$ and previous revision $k - 1$. Maintenance effort $Effort(k)$ measured in change-days is obtained by multiplying maintenance time $t(k)$ by the above proxy measures as a correction factors, leading to the following complete expression for maintenance effort: $Effort(k) = t(k) \times [\Delta sliceSize + \Delta function + \Delta file]$.

V. SLICE-BASED METRICS ON THE LINUX KERNEL

As a way of showing the application of our indirect maintenance-effort metrics on a real system, we have applied the metrics to the Linux kernel. These metrics are then compared to traditional measures of code effort, e.g., LOC.

The Linux revisions are classified as stable and development revisions. Each major revision includes several releases identified with either a three or four digit numbering scheme. The first digit represents the generation, i.e., Linux has three generations, initially with generation 1 released in 1994, generation 2 released in 1996, and generation 3 started in 2011 (not part of the dataset). The second digit represents the major kernel revisions either even or odd. Up until major revision 2.4 even digits (e.g., 1.0, 1.2, 2.0, etc.) corresponded to stable revisions, whereas odd numbers (e.g., 1.1, 1.3, 2.1, etc.) corresponded to development revisions. The third digit is the minor kernel revision. However, in August 2004 this numbering scheme was changed affecting all the revisions released after this date. A fourth digit number was added starting with revision 2.6.8.1, after that the third number in a revision indicates the development of new functionality, and the presence of a fourth number represents bug fixes [45].

We analyzed 11 major revisions containing 974 separate releases, covering a period that exceeds 17 years of software evolution. Figure 3 presents $\Delta Effort(t)$ applied to Linux evolution as estimated from the slice data. The $\Delta t(k)$ is measured in days and multiplied by the summation of effort’s proxy measures (e.g., $\Delta sliceSize$). As an example, let us consider two revisions of the Linux kernel, v.1.1.88 and v.1.1.89, released on 31 January 1995 and 5 February 1995, respectively. From the release dates, it is clear that the maintenance activities of revision v.1.1.89 lasted 5 days. The revision’s $\Delta sliceSize$ = 1000, $\Delta function$ = 112, and $\Delta file$ = 31. Based on these data, the total maintenance effort for revision v.1.1.89 is: $Effort(v.1.1.89) = 5 \times [1000 + 112 + 31] = 5715$ change-days.

![Figure 3. $\Delta Effort$ applied to Linux kernel's evolution in approximation number of change-days.](image)

For each release the size of each revision in measured in LOC as reported by the utility wc, and the size of the revision as both file and release counts. The slice size represents the average slice size for forward slicing. Averages were computed from all slices taken with respect to all the variables in the system (e.g., the total $sliceSize$ for each major revision, $gds$). The full sources of the kernel (.gz) files were downloaded from the official Linux kernel archives (www.kernel.org).

We use the data extracted from the 11 major revisions to indirectly build estimation models for maintenance effort. The models were built on data from 783 revisions, and then validated on the maintenance data of 191 revisions from major revision 2.6. Table II summarizes the descriptive statistics of the collected measures. As we can see for some measures, such as $\Delta sliceSize$, $\Delta function$, $\Delta file$, LOC-g, files-g, and lag-time, the standard deviation is greater than the corresponding mean, indicating that the data are widely spread. For example, for $\Delta sliceSize$ the minimum value is 5 and the maximum value is 899,045 representing an extremely wide range. It is worth noting that the wide range of data does not indicate a fault in the measurements. In other words, these measurements do not necessarily need to be a certain value or within a certain range.
It is common for historical datasets to contain a considerable number of missing values, and several techniques have been developed to deal with this [3]. The main advantage of our dataset is that it does not contain missing values. This is because our data is the source code, and no external metadata or other records are used.

VI. SLICE-BASED MAINTENANCE EFFORT MODELS

As discussed in Section I, building an accurate maintenance-effort estimation model should be derived from accurate maintenance-effort data, which is rarely recorded for open-source, and many closed-source, systems [2, 3]. Therefore, we cannot apply an effort-estimation model built from a closed-source system directly to an open-source system because the absence of maintenance-effort data prevents validation. Alternatively, we take the following approach:

Phase 1: Identify measures that are theoretically related to and can indirectly represent maintenance effort. The candidate measures should be available for most systems, both closed and open source. If such measures can be found and validated, we can construct an indirect model for maintenance effort and use it to predict the indirect effort of open-source systems.

Phase 2: Extract the maintenance data. The data include indirect maintenance-effort identified and validated in previous phase (aka dependent variables) and the data of other related measures that can be used to predict the indirect maintenance effort (aka independent variables). For example, if we identify source-code changes from revision k to revision k+1 as the indirect maintenance effort, then LOC change between both revisions is a measure of source-code change. In this paper, we use slice-based changes to measure the indirect maintenance effort, so sliceSize change is a measure of source-code change.

Phase 3: Validate the correlation between the dependent variables and independent variables. We used Spearman’s rank-correlation coefficient since there are no assumptions regarding the underlying distribution of the data, and its use is recommended for hypothesis testing when the number of data points exceeds 30 [5]. Strong correlation means that the independent variables can be used to indirectly represent maintenance effort; weak correlation indicates the measure is not eligible to represent maintenance effort.

Phase 4: Multiple linear regression analysis is used to build the effort-prediction approach. Specifically, the indirect maintenance-effort is represented as a function of other related measures. We validate this approach against collected maintenance data from the Linux kernel. In addition, we show how we can improve this approach by considering three different granularities of slice sizes.

Phase 5: Predict the indirect maintenance effort based on the models built in the previous phase. The indirect maintenance effort can be represented at three levels, namely line level, function level, and file level. Therefore, the dependent variables could be one of the ASize, Afunction, and Afile multiplied by the maintenance time measured in days, or simply we use the effort equation \( E(k) \). The independent variables are LOC, files, sliceSize, files-g, LOC-g, and Scoverage. Table III shows Spearman’s rank correlations between the dependent variable and independent variables based on the maintenance data of 783 revisions of Linux. The correlation coefficients that are statistically significant at the 0.01 level (2-tailed) are shown in bold. The strong linear relations are not necessarily significant, since the significance is specified by the p-value.

From Table III, we can distinguish multiple significant linear correlations between the dependent variable and some of the independent variables. The \( \Delta E \) is significantly correlated with files-g, LOC-g and Scoverage. Based on this observation, we built the indirect maintenance-effort estimation model. This model considers only those independent variables that have significant correlations with the dependent variable at the 0.01 level. The model is:

\[
\Delta E = c_1 + c_2 \text{ (files-g)} + c_3 \text{ (LOC-g)} + c_4 \text{ (Scoverage)}
\]

The \( c_1 \) variable represents the constant factor or the intercept, which characterizes the height of the regression line when it crosses the y-axis where the dependent variable is plotted, or we can say that the \( c_1 \) represent the predicted value of the dependent variable when all the independent variables are equal to zero. The \( c_i \) (where \( i = 2 \) to \( 4 \)) represents the slope of the line regression which indicates the sensitivity of the dependent values to the changes in the independent values. That is \( c_i \) represent the increase or decrease in \( y \) for each unit change in \( x \). For example, in \( \Delta E \)'s model the coefficient for the LOC-g is equal \( c_3 \), so for every unit decrease in the LOC-g a c3 unit increase in \( \Delta E \) is predicted, when holding all other variables constant. The effort-estimation model is linear, and linear regression is used to estimate the coefficient. Table IV shows the linear regression analysis of the model. The p-value demonstrates the ability of the independent variable to have a significant predictive capability in the presence of other variables. If the independent variable has a non-significant p-value, then we can remove this variable and refit the model again, since this variable does not have predictive capability in the presence of other independent variables. If adding a new independent variable can improve the accuracy of the model, then this variable is said to have the predictive capability.
The R² coefficient of determination value is important to determine whether or not the regression model was helpful. If the regression line provides an estimate of the predictable values that closely match the observed values, then the R² value will be close to one (the better the data fits the model), and with zero indicating no relation between independent and dependent variables. The adjusted-R² that adjusts for the number of independent variables in a model is also calculated. The value of adjusted-R² only increases if a new independent variable improves the model more than would be expected by chance.

From Table IV, we can see that this model has a moderate both R² and adjusted-R² values, which means, based on the data of 783 revisions, the model is by some means accurate in predicting the indirect maintenance effort. Unfortunately, a high value of R² does not guarantee the goodness of the model and does not indicate whether the appropriate independent variables have been used in the model or not [3, 14]. In addition, the high values of R² and adjusted-R² do not assess the quality of future prediction, but only the capability of fitting the sample data. That is, if an effort estimation model is developed using a particular dataset and the accuracy of this model is evaluated using the same dataset, then the value obtained will be optimistic. The error will be low and will not represent the performances of the model on future datasets.

### VII. Evaluating Model Performance

To study the quality of the proposed model for future predictions, we apply the model to predict the indirect maintenance effort of 191 revisions from major revision 2.6. These revisions range from revision 2.6.25.3 released May, 10 2008 to revision 2.6.37.1 released Feb, 17 2011. The predicted results and the actual observed measurements are compared to study the accuracy of predictions. Model validation is the most important step in the model building process. The validation of a model often consists of the analysis of residuals [2, 3, 10]. The residual represents the difference between the predicted value estimated by the model and the observed value of the dependent variable. Our residual analysis includes the following.

SPR statistics: is the sum of absolute value of the residuals (e.g., prediction errors). That is, the SPR = \sum k | Observed k – Predicted k |. MRE statistics: the magnitude relative error, which include the MMRE (mean magnitude relative error), and MdMRE (median magnitude relative error). The MRE is defined as: MRE k = (| Observed k – Predicted k |) / Observed k. The MdMRE is calculated, since the MMRE is known to be very sensitive to the extreme values, such as a few very high relative error MRE values could influence the overall result. Other indicators commonly used to evaluate the prediction model based on MRE are the percentage of prediction at specific level PRED, which measures the percentage of predicted values within X% of the observed values. The value of X is suggested in [46] to be at least 25% and a good prediction model should predict 75% of the observed values. The two variants of the measure PRED we calculated are:

PRED25: the number of predicted values for which MRE was less than or equal to 25%.

PRED50: the number of predicted values for which MRE was less than or equal to 50%.

The predicted results and the measurements are compared to study the accuracy of the predictions. Figure 4 illustrates the comparisons of the predictions and measurements for ΔEffort. In this figure we plotted the ΔEffort values (observed and predicted) on the y-axis with the revision date on the x-axis.

### VIII. Related Work

Many approaches to the effort-estimation problem have been derived using different assumptions, data sources, and methods to process the data to estimate the effort in the context...
of maintaining strictly managed and closed-source systems [2, 10]. These models can be categorized into three main categories namely Analogy, Delphi, and Parametric [47]. The first two categories derive the estimation models based on the past experience of similar systems, or using expert opinions. In contrast, Parametric effort estimation models involve the construction of statistical models from empirical data, e.g., using regression analysis on available data. Moreover, the Parametric models mathematically relate the effort and duration (e.g., days) to the variables that influence them.

Boehm et al. [48] was the first to presents an algorithmic software cost estimation model named the constructive cost model COCOMO. In [49] the same author extended the COCOMO model to estimate the maintenance effort by using a size-change factor to estimate the development effort. This factor represents the estimation of the size of changes expressed as the fraction from the total size of the system measured in LOC, this factor is change over a year period. De Lucia et al. [3] called this factor the “annual change of traffic” since this metric estimates the total software LOC changes during the year. Another work based on the size of changes is presented by Hayes et al. [44] who built a model for adaptive-maintenance effort using the changed LOC and the number of operators changed.

Belady and Lehman [50] suggest a model to approximate the cost and effort of releasing a new revision from an old one. The suggested model estimates the efforts that are related to both the functionality updating and anti-regressive activities. The maintenance-effort estimation that involves the convention of linear regression analysis was introduced by De Lucia et al. [14]. In this research, the authors claimed that the types of the different maintenance tasks should be considered to improve the outcomes of the estimation model being used. Coarse granularity measures have an impact on predicting required changes during the maintenance activities of the software project. For example, Lindvall [51] demonstrates that the number of classes outperform the finer grained metrics in change prediction. In contrast, non-linear cost estimation models were proposed by several researches. For example, in [52] a code decay and a related number of measurements were illustrated to construct a non-linear changes prediction model.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a large-scale empirical study aimed at building indirect maintenance-effort estimation models for open-source systems. The dataset was obtained from the Linux kernel and used as a case study to build and validate the models performance using multivariate linear regression. Our proposed maintenance-effort estimation models are able to accurately determine the source-code changes based only on the source code, and estimate the maintenance effort based at the amount of changes made maintaining the system. It is worth noting that we did not construct a direct maintenance-effort model (person-hours) for open-source systems. However, we decided to use the available source code, because: (1) there is limited direct maintenance-effort data available for open-source systems and we therefore cannot validate the correctness of such a model; and (2) maintenance effort represented as person-hours is less meaningful for open-source systems.

REFERENCES