More Debugging in Parallel

Wolfgang Högerle
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Germany
w_hoegerle@gmx.de

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Germany
steimann@acm.org

Marcus Frenkel
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Germany
Marcus.Frenkel@feu.de

Abstract—Programs may contain multiple faults, in which case their debugging can be parallelized. However, effective parallelization requires some guarantees that parallel debugging tasks do not address the same fault, an inherent problem of earlier, clustering-based approaches to parallel debugging. In this paper, we identify a number of fundamental trade-offs to be made when selecting algorithms for parallel debugging, and explore these trade-offs using one clustering algorithm and three algorithms from integer linear programming. Results of an evaluation involving a total of 75,000 faulty versions (with up to 32 faults) of 15 subject programs suggest that depending on the number of faults present and the trade-offs one is willing to accept, speed-ups much larger than previously reported can be achieved, even if all derived parallel debugging tasks are handled sequentially.

Keywords—Debugging, Parallelization

I. INTRODUCTION

“Although more research is necessary to determine the best clustering technique, we have demonstrated the promise of parallelizing the debugging effort in such an automated way.” [16, p. 25]

In [16], Jones, Bowring, and Harrold suggested that based on clustering of the set of failed test cases, the task of debugging a program in presence of multiple faults can be divided into parallel subtasks, each focusing on a single fault. A central result of their work is that this can speed up debugging significantly, even if the derived parallel debugging tasks are approached sequentially. From a theoretical perspective, the latter circumstance is of particular interest, since it allows a valuation of the achievement that is independent of the complex scheduling problems that true parallelization involves, which depend on many factors intrinsically difficult to control.

Contributions In this paper, we take a fresh look at the partitioning problem that is at the core of “debugging in parallel”. More specifically, we explore ways to increase the degree of parallelization that can be achieved, and shed light on the trade-offs this involves. Amongst others, we introduce the notion of a bug race, and reflect on the threat it represents to parallelization efforts. We complement our theoretical considerations with evaluation data obtained from partitioning 75,000 multi-fault versions (with up to 32 faults) of 15 Java programs using four different partitioning algorithms, each pursuing a different strategy for increasing parallelism. Our results suggest that the choice of the partitioning algorithm has a significant impact on the qualities of the partitioning; at the same time, our findings let all hopes that obtained clusters focus on single faults appear unrealistic.

NB: We make no claims regarding the absolute utility of any partitioning algorithm for the general problem of debugging in parallel — our assessment is strictly comparative, and meaningful only with all other conditions being equal. In particular, we are well aware of the fact that in practice, assignment of debugging tasks to developers, the time needed to solve each task, and the speed-up that can be obtained by parallelization, depend on a plethora of factors (some of which, arguably including all human factors, are very hard to control). Yet, like so many other development aids, automatically suggesting a parallelization of debugging in presence of multiple faults (as detected, for instance, after a nightly build phase, when the regression test suite is re-run) is an offer developers can accept or ignore — whether or not they appreciate this offer is another question (and one not meant to be answered by this paper).

The remainder of this work is organized as follows. Section II recapitulates the original work by Jones et al. and that of others. Section III lists the specific definitions and assumptions underlying the evaluation of the algorithms described in Section IV. Section V describes the setup of our experiments, whose results are presented in Section VI. A discussion and a brief treatise of the threats to validity conclude.

II. RELATED WORK

In their original work on parallel debugging (which the title of this work alludes to and which, to this day, is considered the reference work on debugging in parallel), Jones, Bowring, and Harrold presented two techniques that partition failing test cases into clusters each targeting a different fault (so-called fault-focussing clusters) [16, 17]. The rationale behind this is that different fault-focussing clusters can be assigned to different developers who can then work in parallel. Also, the number of clusters is said to provide a prediction of the number of (non-obfuscated) faults in a program.

The first clustering technique of Jones et al. clusters failed test cases according to the similarity of their executions’ branching behaviour, summarized as discrete-time Markov chains. Clustering is stopped when the two clusters (of failed test cases) to be merged next appear to target different faults. This is determined by computing the set of most suspicious statements from each cluster (using the failed test cases from a cluster, plus all passed test cases, as input to the Tarantula fault localization technique [15, 17]) and comparing these sets by computing their relative overlap (using the Jaccard similarity of sets). Both the set of most suspicious statements and (binary) set similarity depend on thresholds that can be chosen to tune clustering performance.
The second, considerably simpler, clustering technique directly uses the fault localization results obtained from each single failing and all passing test cases as the basis of computing pairwise similarity of the failed test cases (again using Jaccard similarity and the two thresholds). Fault-focusing clusters are then formed by computing the transitive closure of pairwise similarity.

Each cluster of failed test cases (obtained using either of the above clustering technique) is then complemented with all passing test cases, constituting a fault localization problem that can be handled in parallel with all others. However, even though the clusters are expected to cover different faults, this need not be the case [16, p. 23], so that some of the parallel debugging effort may be lost. As we will see, this is a typical trade-off to be made when trying to increase the number of clusters.

An evaluation on a single subject program with ample (i.e., more than 30-fold) test coverage demonstrated 14% and 28% savings for the first and second clustering technique, resp., in total effort for finding a bug in each cluster (compared to finding the same number of bugs without clustering, re-running the test suite after each found bug had been fixed), as well as 50% and 43% savings when debugging is parallelized [16].

Given that the outcome of studies of this kind tend to depend more on the used subject program than on most other factors [26], the results cannot be generalized. Nevertheless, we could likewise demonstrate reductions in total debugging effort for the second clustering technique, up to eight injected faults (using a slightly different evaluation setting; see Section VI.C).

The idea of clustering failing executions in presence of multiple faults based on a behaviour model was previously explored by Podgurski et al. [22], who used cluster analysis to group execution profiles of user-reported failures ascribed to the same or similar faults [22]. These groups can then be used to predict the frequency of failures caused by a given fault, or to localize that fault. In a variation of this approach, Liu and Han grouped failing traces according to the proximity of the location of the causing fault, as determined by a statistical debugging tool [18]. These two approaches roughly correspond to the two versions of Jones et al.’s clustering technique. However, both works do not address the problem of parallel debugging.

Most clustering algorithms, like neural networks and machine learning, are data mining techniques that separate data points in a non-linear fashion. However, applied to the problem of parallel debugging this non-linearity leads to overlaps between the program units covered by the test cases in each cluster. In practice, this means that parallel searches may focus on the same bug, which is arguably counterproductive. Algorithms that perform a strict separation in both dimensions (i.e., neither test cases nor units under test overlap between clusters) therefore seem better suited. Such algorithms can be found in the field of integer linear programming.

In our own prior work [24], we first used conversion to a block diagonal matrix to divide the problem of debugging in the presence of multiple faults into several smaller, strictly independent (i.e., no overlaps between rows or columns) problems. However, the degree of parallelization that could be achieved this way was not always satisfactory (the number of blocks per injected fault was roughly $1.27^{n-1}$ on average, where $n$ was the number of injected faults [25]), especially since further parallelization was often prevented by just a small number of test cases and/or units under test that constituted the overlapping. To deal with this phenomenon, we adopted another method from integer-linear programming (the Weil-Kettler algorithm [27]) that computes block diagonal matrices dismissing impeding rows or columns, and recursively applied it to test coverage matrices that otherwise resisted further division [25]. Our approach was interactive in that at each recursion level, the tested program units preventing further division into independent blocks had to be inspected for faultiness first — only if they were fault-free, debugging the resulting blocks commenced (to keep the inspection effort low, the algorithm was modified to minimize the number of program units that had to be inspected at each partitioning level). This approach increased the parallelization obtained by computing block diagonal matrices by approx. 68% on average. In more recent work, we used another method from integer linear programming, maximum set packing, to compute a minimum number of faults in a program from the coverage of its failed test cases [26]. This allowed us to search more than one fault before re-running the test suite, but worked without partitioning so that no parallelization was supported. In our present work, we will use maximum set packing as a new method of dividing the debugging problem into independent smaller ones, directly comparing its parallelization performance with that of Jones et al.’s second algorithm, block diagonal matrices, and unmodified Weil-Kettler.

Parallel debugging of multiple faults is also enabled by model-based diagnosis. The underlying idea here is to have a logical model for the failure of each test case, and to find an assignment to the health status of each component of the model (the units under test, e.g., the covered statements) that explains all failures. The coverage, or hit spectrum, of the failed test cases is such a model; from it, minimal diagnoses can be computed as minimal hitting sets [2, 10]. Minimal hitting sets usually contain several units under test, all of which need to be faulty to explain the observed failures. Consequently, for each minimal hitting set, the contained units under tests can be inspected in parallel. The problem of this approach, which, unlike Jones et al.’s, does not primarily partition test cases, is that several minimal hitting sets may exist that explain the failure pattern from which they have been derived; therefore, [2] also introduces a ranking of hitting sets based on Bayesian reasoning. This ranking is however somewhat incompatible with the idea of parallel debugging.

Concerning the adverse effects multiple faults may have on fault localization (and thus on localization-based failure clustering), DiGiuseppe and Jones investigated the phenomenon of fault interaction [13]. They distinguished interactions of multiple faults that, compared to the interacting single faults, lead to additional failures, or that obfuscate faults (amounting to fewer failures). We will also observe effects of this kind in our current study, but maintain that their impact on the performance of failure clustering techniques has to be accepted as real.

We do not discuss here other work that targets at fault localization in presence of multiple faults, without addressing the potential for parallelization (such as [7, 28]). It has been discussed in previous work of ours [24, 25] and is not immediately related to our current one.
Parnin and Orso have recently questioned the utility of automated debugging techniques in general [20]. Although their criticism targets at fault localization, rather than partitioning, techniques, it can be transferred to our endeavour also. On the other hand, historically the dismissal of emerging technologies has often been premature, and many theoretical results have later found their use in practice, even if not quite in the way originally anticipated.

III. PRELIMINARIES

A. Definitions

Mathematically, a partition of a set \( X \) is a set of pairwise disjoint, non-empty subsets of \( X \) whose union is \( X \).\(^1\) Partitioning extends from sets to matrices in that partitions of the sets of rows and columns of a matrix divide, or partition, the matrix into non-overlapping submatrices, or blocks, that together cover the original matrix [14]. Note that to make blocks contiguous, the rows and columns of the matrix may have to be permuted.

A test coverage matrix (TCM), or hit spectrum, is a binary matrix whose rows represent the units under test (UUTs) and whose columns represent the test cases (Fig. 1). A “1” in the matrix states that the corresponding UUT is executed by the test case, a “0” states that it is not. The partitioning of the set of test cases into failed and passed ones partitions the TCM (also Fig. 1); we call the submatrix defined by failed test cases a failed test coverage matrix (FCM).

It is important to note that the UUTs that make up a TCM are usually identified at a different level of granularity than the units tested by so-called unit tests. For instance, while the UUTs are commonly recorded at the level of statements or methods, unit tests usually cover larger units, such as classes or packages (and therefore more than one row in a TCM). Furthermore, it is important to note that JUnit test cases, despite the name of their hosting framework, are not necessarily unit tests; integration and system tests can equally be implemented as JUnit tests. Indeed, for the subject programs used in our evaluation (Section V), more than half of the test cases cover more than 13 methods, more than 3 classes, and more than 1 package (or, in case of inner classes, containing class); and more than one fifth of all test cases cover more than 139 methods, more than 27 classes, and more than 5 packages (or containing classes).

For the purpose of this paper, we define the debugging task to be the task of identifying the UUTs that cause the failure of test cases, solely using information found in a FCM (and, possibly, the TCM it is a submatrix of). Identifying faults that do not make test cases fail is not considered part of the debugging task. This means that the test cases are the only specification of the program to be debugged.

A block diagonal matrix (BDM) is a partitioned matrix whose blocks off the main diagonal contain only zeroes [14]. A matrix is transformed to a BDM by permuting its rows and columns in a suitable manner (see Fig. 2 a) and b) for an example). The partitioning of a FCM into a BDM is ideally suited for parallel debugging in that each block of interest (i.e., each block on the main diagonal) constitutes its own, independent debugging problem. This is so because the test cases in such a block do not cover any UUTs outside that block (all zeroes above and below that block), and the UUTs in a block are not covered by any test case outside that block (all zeroes left and right to that block; cf. Fig. 2 b). Unfortunately, as we will see, the number of independent debugging task that can be obtained by transforming a FCM into a BDM is usually rather small, so that the parallelization that can be achieved this way is limited. It is therefore useful to investigate alternative divisions of matrices into smaller units.

One way to achieve higher degrees of parallelization is to let the blocks that represent the otherwise independent debugging tasks share rows or columns (corresponding to UUTs or test cases; see Fig. 2 c) and d). Since the set of so-defined blocks does not constitute a partition of the FCM in the mathematical sense (cf. the above definition), we call it a dirty partition (and refer to a partition that is not dirty as a clean partition). Note that the partition defined by a BDM is always clean.

In this work, we limit our studies to dirty partitions of FCMs that result from sharing either rows (Fig. 2 c) or columns (Fig. 2 d), but not both. This allows us to cleanly separate the different consequences of the corresponding overlaps:

- If blocks share UUTs, but no test cases (meaning that the test cases — and only the test cases — are partitioned; Fig. 2 c), a possible consequence is that some UUTs are inspected multiple times. In particular, if a shared UUT is faulty, the effort invested in searching a fault in one block may be completely wasted, namely if the fault was found in another block first. We call such a condition a bug race.
- If blocks share test cases, but no UUTs (meaning that the UUTs — and only the UUTs — are partitioned; Fig. 2 d), bug races are avoided. However, in that case searching the fault causing the failure of a shared test case within a single block may be futile, since the cause of the failure may lie in a different block. Searching the complete coverage of each shared test case for the fault on the other hand extends the

\(^1\) Note that the term partition denotes a parting, not a part.
search into a different block (cf. Fig. 2 d), leading to bug races. Therefore, it is not useful to include shared test cases in a block; instead, they must be ignored. We call such test cases lost test cases. Losing test cases results in an incomplete partition of the FCM, and may mean a loss of diagnostic information in the debugging process, particularly if the coverage of a UUT is lost completely. In that case, we call the UUT a lost UUT.

Intuitively, bug races, lost test cases, and lost UUTs may have an untoward effect on debugging; yet, if they lead to higher degrees of partitioning (and hence parallelization) than clean partitioning, their consideration may be worthwhile.

B. Underlying Assumptions

Our experiments (Section V) and their results (Section VI and VII) rest on a number of assumptions:

1. Faults are intermittent, i.e., faulty UUTs are covered by both failing and passing test cases. However, we assume that every fault makes at least one test case fail — otherwise, it will not be recognized as a fault by a developer (cf. Section A).

2. Conversely, we do not assume that every failed test case executes at least one fault whose execution causes the failure (i.e., we do consider what has been called a spurious failure in [26]). This means that the search of a bug may be in vain. By not excluding these cases, we take into account that failures of this kind do occur in practice: in our experiments described below, they make up for approximately 4.4% of all failures.

3. For the purpose of comparing the quality of partitioning by estimating the effort required to identify a fault in a block, we assume that a programmer always, upon first inspection, recognizes a UUT that causes the failure of a test case as faulty, and recognizes no other UUT — wrongly or rightly — as faulty.

4. Debugging is based on the results of a single run of the regression test suite. When the test suite were re-run after one or more faults have been fixed, we would get a new FCM, a new partitioning, and new parallel debugging tasks, which all depend on the specific fixes performed.

IV. PARTITIONING ALGORITHMS

Permuting a FCM into a BDM is straightforward; [24] describes an algorithm that computes a clean partition that is always maximal (meaning that there is no clean partition with a greater number of blocks) and uniquely determined with effort $O(r^2c)$ (where $r$ and $c$ are the numbers of rows and columns, respectively). Thus, this algorithm gives us the best clean and complete partition that we can expect. To achieve higher degrees of partitioning, we will have to use algorithms that compute dirty or incomplete partitions (cf. Section III.A).

A. The Clustering Algorithm by Jones et al.

While computing the BDM of a FCM partitions both the set of UUTs and the set of test cases, both algorithms by Jones et al. partition only the set of test cases, accepting that UUTs are shared among blocks (cf. [16] and the algorithms’ description in Section II). This means that the obtained partitions are complete, yet dirty; specifically, using either algorithm, bug races may occur.

In this study, we use only the second algorithm, since the first is difficult to reproduce faithfully from the original publications, and since their difference in performance has been shown to be moderate (see [16] and the summary in Section II). We set the thresholds to the values used in the original evaluation [16]; see Appendix for how we tested them for aptness in our own evaluation.

B. The Weil-Kettler Algorithm

The Weil-Kettler algorithm, which was originally designed to solve sparsely populated linear equations [27], identifies rows or columns of a binary matrix that prevent its clean partitioning. Applying suitable permutations, it produces a matrix that, except for some rows or columns singled out at the top or left, has block diagonal form (Fig. 3 b) and c). Depending on whether rows or columns are singled out, we call the algorithm horizontal or vertical Weil-Kettler. Its working has been described in [25] (note that we use the original version [27] of the algorithm here; cf. Sect. II).

When using the Weil-Kettler algorithm, the question is what to do with the singled-out rows (corresponding to UUTs) or columns (test cases). When the UUTs of the FCM singled out by horizontal Weil-Kettler are ignored, debugging of the blocks of the resulting clean partition may be futile, namely when the ignored UUTs are the faulty ones. To avoid this, the UUTs singled out would need to be inspected for faults first. If faults are found, debugging of the remaining clean partition may be in vain (because their test cases may have failed only because of the faults singled out), so that fault fixing and re-running of the tests would be necessary. This however would interfere with parallelization (cf. Section III.B, Item 4), so that in order to be able to compare horizontal Weil-Kettler with the other partitioning algorithms used in our study, we extend
each block formed by the partitioning of the test cases to cover the single out UUTs (so that the partition is dirty; see Fig. 3 b) and observe the hatching, marking non-contiguous blocks). This however may give us bug races.

When the columns (test cases) of a FCM singled out by vertical Weil-Kettler are ignored, we lose their diagnostic information (Fig. 3 c). And yet, this is what has been suggested for shared test cases in Section III.A, arguing that the sharing between blocks may lead to vain searches in all but one sharing block. Therefore we ignore the test cases singled out by the vertical Weil-Kettler algorithm. Note that the resulting partition is clean, but incomplete: it does not cover the original FCM (the coverage from the ignored test cases is lost and, if these test cases were the only ones covering certain UUTs, these are also lost).

C. Maximum Set Packing

The removal of columns (test cases) preventing a clean partitioning can be driven to the extreme, by removing so many of them that the remaining FCM is clearly partitioned into blocks each consisting of a single test case only. A maximum partition of this kind (i.e., a partition that contains as many one-column blocks as possible) is called a maximum set packing (MSP) [21]. The maximum set packing problem is known to be NP-complete; yet, good approximations can be reached using a greedy algorithm, which is cubic in the size of the matrix [21]. For our study, we developed an algorithm that also tries to minimize the size of each “pack”, that is, the coverage of the test cases that constitute the packing (weighted maximum set packing), whose computational effort is $O(r^3 c)$. This is to speed up the debugging process further, by making the resulting partitions as small as possible. Because of its more radical approach, we expect maximum set packing to lose more test cases and UUTs than vertical Weil-Kettler; this loss may however be outweighed by better partitioning (resulting in a larger number of blocks), leading to better results than a partitioning algorithm that loses fewer test cases and UUTs.

Note that unlike for the Weil-Kettler algorithm, it does not make sense to apply maximum set packing to the transposed problem (i.e., to find a maximum packing of UUTs), since this would require the removal of UUTs that are potentially faulty and there is no reason to believe that the remaining, single UUTs constituting the packing are faulty.

Table 1 summarizes the properties of the algorithms used.

<table>
<thead>
<tr>
<th>Property</th>
<th>BDM Jones</th>
<th>Algorithm</th>
<th>ver. W-K</th>
<th>MSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>clean partitioning/bug races</td>
<td>yes/ no/</td>
<td>no/ yes</td>
<td>yes/ no/</td>
<td>yes/</td>
</tr>
<tr>
<td>complete partitioning/lost test</td>
<td>yes/ yes/</td>
<td>no/ no/</td>
<td>no/ yes/</td>
<td>yes/</td>
</tr>
<tr>
<td>cases or UUTs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Subject programs used

<table>
<thead>
<tr>
<th>SUBJECT PROGRAM AND VERSION NUMBER</th>
<th>UNITS</th>
<th>UUTS</th>
<th>TCS</th>
<th>UUTS/TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC Codec 1.3</td>
<td>265</td>
<td>229</td>
<td>188</td>
<td>7/7</td>
</tr>
<tr>
<td>AC Lang 3.0</td>
<td>5373</td>
<td>2075</td>
<td>1666</td>
<td>8/11</td>
</tr>
<tr>
<td>Barbecue Rev. 87</td>
<td>710</td>
<td>297</td>
<td>154</td>
<td>25/27</td>
</tr>
<tr>
<td>Daikon 4.6.4</td>
<td>14387</td>
<td>1936</td>
<td>157</td>
<td>30/84</td>
</tr>
<tr>
<td>Eclipse Draw2d 3.4.2</td>
<td>3231</td>
<td>878</td>
<td>89</td>
<td>82/65</td>
</tr>
<tr>
<td>Eventbus 1.4</td>
<td>859</td>
<td>338</td>
<td>91</td>
<td>32/22</td>
</tr>
<tr>
<td>HTML Parser 1.6</td>
<td>1925</td>
<td>785</td>
<td>600</td>
<td>142/93</td>
</tr>
<tr>
<td>Jaxen 1.1.5</td>
<td>1689</td>
<td>961</td>
<td>695</td>
<td>186/81</td>
</tr>
<tr>
<td>JDepend 2.9</td>
<td>426</td>
<td>171</td>
<td>55</td>
<td>38/37</td>
</tr>
<tr>
<td>Jester 1.37b</td>
<td>378</td>
<td>152</td>
<td>64</td>
<td>15/12</td>
</tr>
<tr>
<td>JExel 1.0.0b13</td>
<td>242</td>
<td>150</td>
<td>335</td>
<td>21/14</td>
</tr>
<tr>
<td>JParsec 2.0</td>
<td>1011</td>
<td>893</td>
<td>510</td>
<td>40/39</td>
</tr>
<tr>
<td>Mince4J 0.5</td>
<td>1663</td>
<td>859</td>
<td>251</td>
<td>49/69</td>
</tr>
<tr>
<td>Time &amp; Money Rev. 207</td>
<td>618</td>
<td>382</td>
<td>233</td>
<td>17/13</td>
</tr>
<tr>
<td>XMLEnc 3.0</td>
<td>1630</td>
<td>366</td>
<td>84</td>
<td>72/80</td>
</tr>
<tr>
<td>mean</td>
<td>2294</td>
<td>698</td>
<td>345</td>
<td>51/44</td>
</tr>
<tr>
<td>standard deviation</td>
<td>3488</td>
<td>589</td>
<td>403</td>
<td>50/31</td>
</tr>
</tbody>
</table>

† here: total number of methods, not counting JUnit test methods

Because of the large absence of mathematical models of the nature of real TCMs and FCMs [12], we assess the appropriateness of our selected partitioning algorithms experimentally. For this, we use 15 subject programs (including their test suites) publically available, and automatically inject multiple faults into them. The experimental setup largely follows that described in [25, 26]; its cornerstones are repeated here to make the paper self-contained.

The subject programs used are listed in Table 2. Their number of test cases and units under test (methods in our study; see below) totals to 5,172 and 10,473, respectively. All programs are written in Java, and all test cases are JUnit test cases, run with the JUnit 4 test runner. Finding these subject programs (and making their test suites executable) presented a significant effort (many open source projects depend on outdated versions of external libraries which are difficult, if at all possible, to obtain; many open source programs do not have test suites; of those that have, few have other than trivial test cases; note that this includes both the SIR [http://sir.unl.edu] and the Dacapo [http://dacapobench.org] benchmarks, but not our selected programs). These difficulties made drawing a random sample of subject programs, as would be necessary to make our results generalizable (see, e.g., [12]), virtually impossible (cf. Section VIII).

Because in object-oriented programs, methods provide a natural context for searching a fault (they are usually short and most variables are local to a method; also, many debugging tools provide specific assistance at the method level), we used methods, rather than the more popular statements, as the granularity level of determining test coverage (i.e., our UUTs are all methods). Note that, because the algorithms used do not depend on the granularity of the abstraction that a TCM represents, they are applicable to any granularity (including classes, if these are considered the more suitable units of debugging). Faulty versions of the subject programs were obtained using automated fault injection, or mutation. While the use of mutation to generate faulty programs may be questioned in general [6], recent experiments in the area of fault localization have shown that replacing real faults with mutants has had no significant impact on outcome [5, 19]. We used the six fault injectors shown in Table 3 (the first four of which were previously used in the evaluation of mutation analysis itself [6] and in the mutation-based evaluation of fault locators [5, 19]). Each fault injector was used to create as many as possible, but no more than 100 random injections of a single fault into each
TABLE 3. FAULT INJECTORS (MUTATION OPERATORS) USED

<table>
<thead>
<tr>
<th>NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negate Decision (ND)</td>
<td>negate condition in an if or while statement</td>
</tr>
<tr>
<td>Replace Constant (RC)</td>
<td>replace integer constant C by 0, 1, (-1, C+1) or (C-1)</td>
</tr>
<tr>
<td>Delete Statement (DS)</td>
<td>delete a statement</td>
</tr>
<tr>
<td>Replace Operator (RO)</td>
<td>replace an arithmetic, relational, logical, etc. operator by an operator from the same class</td>
</tr>
<tr>
<td>Assign Null (AN)</td>
<td>replace rhs of assignment with null</td>
</tr>
<tr>
<td>Return Null (RN)</td>
<td>replace return expression with null</td>
</tr>
</tbody>
</table>

subject program such that the injection made at least one test case of the subject program fail. This gave us up to 600 detectable fault injections per subject program. In the single fault case (which is used here for reference only), all injections were used to produce one faulty version of the subject program each; in the multiple fault cases, we used the geometric progression 2, 4, 8, 16, 32 for \(n\), the number of faults, and randomly drew 1000 samples of \(n\) injections for each \(n\), making sure that in each sample, the faults were located in different methods. This procedure gave us 5,000 multi-fault versions for each of our 15 subject programs, summing up to 75,000 faulty versions with a total of 930,000 faults. These included 548 spurious failures and 276,279 masked faults (called obfuscated faults in [13]; note that their large number has previously been observed in [13]). Although we could have easily removed these anomalous cases from our sample, we consider their existence as real, and their effect on the evaluation as that of a naturally occurring distortion that has to be accepted.

VI. EVALUATING ALGORITHM PERFORMANCE

Whether one partitioning algorithm performs better than another is a multidimensional question. In fact, any attempt to linearly rank the partitioning algorithms requires so many arbitrary stipulations that it is practically worthless. Instead, we measure the performance of the algorithms in each dimension separately, leaving the global assessment of the results to Section VII.

The first and most obvious criterion for judging the usefulness of a partitioning algorithm for the purpose of parallel debugging is how many non-zero blocks it computes. This will be the subject of Section A. However, this number alone is meaningless: for instance, (dirty) a partitioning of a FCM into its columns (meaning that each test cases constitutes its own block) allows for highly parallelized debugging, but likely wastes a lot of resources, since it leads to many bug races. Likewise, a parallel inspection of all UUTs in a FCM would lead to a near instant discovery of all faults, but much of the invested effort would be wasted, since most blocks (UUTs) will not be faulty. Section B therefore contrasts the number of blocks (parallel debugging tasks) derived with the quality of the obtained partitions. Section C presents the relative speed-up for each approach (based on an estimation of the total wasted effort).

\[ y = ax^b + b \]

A. Partitioning Capability

Fig. 4 shows the average number of blocks derived from the subject programs of Table 2 per number of faults injected for each of the algorithms used. Note that both scales are logarithmic; the (fairly) straight lines suggest a power law (in fact, the untransformed plots fit \(y = ax^b + b\) quite well for all algorithms.) Note that both horizontal Weil-Kettler (the most effective in terms of partitioning capability) and the algorithm of Jones et al. sometimes derive more than one block in presence of a single fault only — inevitably, this causes a bug race.\(^4\)

The dependence of the number of blocks on the number of injected faults observed in Fig. 4 conversely means that the average number of faults per block also follows a power law (it likewise fits \(y = ax^b + b\)). Hence, it is unrealistic to hope that, even when assuming an equal distribution of faults as in our setup, one partition per fault (the so-called fault-focussing clusters of [16]; cf. Section II) can be obtained using any of the investigated algorithms.

B. Quality of Obtained Partitions

As pointed out above, a partitioning algorithm that computes more partitions is not necessarily better than one that computes fewer. Three objective indicators of quality have already been suggested in Section III.A, by identifying three possible problems of the partitioning algorithms used: bug races, lost test cases, and lost UUTs. Intuitively, the less frequent these problems occur, the better for parallel debugging. And yet, these indicators are meaningless without knowing how the algorithms fare with respect to the effort required to find a fault in each derived block.

1) Expected Wasted Effort

As noted in the introduction, in practice the effort required to debug a partition depends on many factors, most of which are outside the control of this study. To get an impression of the relative suitability of the partitions obtained by each partitioning algorithm nevertheless, we compute the average wast-

\(^4\) In the case of horizontal Weil-Kettler, this is caused by singling out the faulty UUT as preventing partitioning (which by itself makes it a hot candidate for debugging; see [25]). As detailed in Section IV.B, this UUT is then added to the resulting blocks, leading to the bug race.
ed effort required to localize a UUT causing a failure in each block. The average wasted effort is the number of UUTs that — on average — need to be inspected in vain in each block before one causing a failure (subsequently simply called a fault) in that block is found. (Cf. the argumentation of Section III.B for why faults not causing a failure in a block are ignored.)

In the rich literature on fault localization (see, e.g., [1, 2, 5, 7, 8, 11, 15, 23]), the order in which UUTs are inspected is usually suggested by some heuristic. So called coverage, or hit spectra, based fault localization algorithms utilize solely information contained in a TCM and its FCM for obtaining such an order, which would make them ideally suited for assessing the quality of partitioning in our setting. However, this would make our results dependent on the quality of the fault localization technique used, opening up yet another dimension of our evaluation. Therefore, with the exception of Fig. 7, we use a lower bound of fault localization accuracy that we have recently put forward [26].

Intuitively, any deterministic algorithm that suggests an inspection order that performs worse on average than a strictly random inspection order provides a lower bound for the fault localization accuracy that we have recently put forward [26].

Mathematically, our lower bound of the wasted effort for finding the first fault in each block depends on the number of faults injected. The overall decrease of average wasted effort per block with increasing numbers of faults is explained by Eq. (1) and the fact that the number of faults per block, $m$, increases faster with the number of faults injected (cf. Section A) than the number of UUTs, $n$, in each block (not shown). The slight increase that can be observed when moving from one to two faults is due to the fact that in a significant number of cases, the number of blocks computed is 1 in both cases; then, $m$ increases less than $n$ in each block. The reason why maximum set packing consistently performs better than all others is that its blocks (each consisting of a single test case only) have the smallest coverage, and thus the smallest set of candidate UUTs to pick from (the smallest $n$ in Eq. (1)).

The ranking of algorithms suggested by Fig. 5 may be unfair — it is easy for an algorithm that creates fewer, smaller blocks (“cherry-picking” by computing an incomplete partition) to perform better than one that computes a complete partition with more, but on average larger blocks. However, even though in our experiments (in which all faults are injected) we always know how many faults are contained in each block, in practice (where the partitioning algorithms are ultimately to be applied) expecting debugging to search for more faults than there is evidence of is unjustified. Although there are ways to assess a minimum number of faults in a FCM (which equally apply to blocks; see, e.g., [4, 26]), they rest on various assumptions (e.g., that spurious failures do not occur) to which we do not wish to subject our results. Therefore, we search only one fault in each block. Note that if a block contains no faults (i.e., $m = 0$), the expected value of the wasted effort for finding the first is $n$, i.e., the number of UUTs in the block (meaning that all effort is wasted). This accords with intuition.

Fig. 5 shows how the average expected wasted effort for finding the first fault in each block depends on the number of faults injected. The overall decrease of average wasted effort per block with increasing numbers of faults is explained by Eq. (1) and the fact that the number of faults per block, $m$, increases faster with the number of faults injected (cf. Section A) than the number of UUTs, $n$, in each block (not shown). The slight increase that can be observed when moving from one to two faults is due to the fact that in a significant number of cases, the number of blocks computed is 1 in both cases; then, $m$ increases less than $n$, the joint coverage of all failed test cases caused by two, rather than one, faults. The reason why maximum set packing consistently performs better than all others is that its blocks (each consisting of a single test case only) have the smallest coverage, and thus the smallest set of candidate UUTs to pick from (the smallest $n$ in Eq. (1)).

The ranking of algorithms suggested by Fig. 5 may be unfair — it is easy for an algorithm that creates fewer, smaller blocks (“cherry-picking” by computing an incomplete partition) to perform better than one that computes a complete partition with more, but on average larger blocks. Fig. 6 therefore shows how the average expected wasted effort depends on the number of blocks computed. Since the average number of blocks created by each algorithm increases monotonically with the number of faults injected (cf. Fig. 4), the left-to-right ordering of the points associated with each algorithm corresponds to the sequence of the number of faults injected (i.e.,
the leftmost point represents the 1-fault case, and the rightmost the 32-fault case). An algorithm whose curve corresponds to the scaled down curve of another algorithm (i.e., whose points are closer to the origin) is therefore more successful, but arguably only in cherry-picking: For instance, while it appears that the vertical Weil-Kettler algorithm performs better than horizontal Weil-Kettler (it drops further and faster), it does so only because it computes fewer blocks for each number of present faults, which require on average less wasted effort than the same number of blocks computed by horizontal Weil-Kettler. It would be fallacious to conclude from this, however, that vertical Weil-Kettler is the better choice if the capacity for parallelization is low; this need not be the case since given the same number of present faults, from the many blocks horizontal Weil-Kettler computes, those could be chosen that are the easiest to debug (which, using random inspection, would be those with the least coverage). Indeed, as we will see, the debugging speed-up that can be achieved by vertical and horizontal Weil-Kettler is roughly the same on the number of blocks computed by vertical Weil-Kettler (Section C).

To prepare for the argument that a certain partitioning algorithm may be especially suited for a specific fault locator, and therefore leads to better results if used in combination with that locator, we also used the Tarantula suspiciousness metric (which was used in the original experiments of Jones et al. [16]) to assess the wasted effort. The results are shown in Fig. 7; as can be seen, even though the overall performance using the Tarantula suspiciousness ranking for computing the wasted effort is consistently better across all algorithms, the relative performance of Jones et al.’s partitioning algorithm remains roughly the same.

2) Bug Races

As noted in Section IV, due to the sharing of UUTs between blocks the algorithms of Jones et al. and horizontal Weil-Kettler may cause bug races, which are detrimental to the parallel debugging process. However, sharing of UUTs, or even sharing of faulty UUTs, alone is not sufficient for a bug race — instead, the first fault found in two or more blocks must be a shared one. Because of the complex nature of the problem, we were not able to derive a formula for an expected value similar to Eq. (1); instead, we simulated random fault localization, counting the number of collisions and averaging it over 200 repeats.

The results of this are shown in Fig. 8. As can be seen, the number of bug races increases roughly linearly, giving us .07 bug races per block (with an offset of .16) for Jones et al.’s algorithm, and .15 bug races per block (with an offset of ~.1) for horizontal Weil-Kettler. Clearly, the greater number of blocks that horizontal Weil-Kettler computes has its price: on average, almost every 6th debugging task runs into a bug race.

3) Lost Test Cases

Vertical Weil-Kettler and maximum set packing do not cause bug races, but may lose failed test cases, and with them valuable debugging information (cf. Section IV). Fig. 9 has the numbers: as was to be expected, the number of test cases lost by maximum set packing is dramatic (dropping test cases is its very nature). On the contrary, the ones dropped by vertical Weil-Kettler are much fewer which, given that the number of blocks it computes is not dramatically lower, makes it appear the better choice if availability of more failed test case is expected to make debugging easier. Using expected wasted effort as the measure of ease, however, this is not indicated (cf. Fig. 6); in this regard, MSP consistently performs better.
faulty UUTs to be detrimental. The loss has to be weighed against the number and quality of blocks obtained: if both are high compared to algorithms that do not lose UUTs, losing the parallel detection of further faults is likely not an issue. Also, with increasing numbers of faults, the faults per block increase as well (cf. Section A), so that as long as search is limited to one fault in each block, not all faults will be found, anyhow.

4) Lost UUTs

When test cases are dropped, it may ultimately be the case that UUTs lose their coverage, in which case they drop out of the partitioning. This is not a problem if they are not faulty. Fig. 10 therefore only shows the number of faulty lost UUTs. As can be seen, the losses of vertical Weil-Kettler are much smaller than the losses of maximum set packing throughout.

While one might consider lost faulty UUTs to be detrimental, the loss has to be weighed against the number and quality of blocks obtained: if both are high compared to algorithms that do not lose UUTs, losing the parallel detection of further faults is likely not an issue. Also, with increasing numbers of faults, the faults per block increase as well (cf. Section A), so that as long as search is limited to one fault in each block, not all faults will be found, anyhow.

C. Partitioning Gain

Most immediately related to the goal of parallel debugging are the savings achieved when debugging a partitioned FCM, compared to debugging the non-partitioned FCM.

However, as argued in the introduction, the actual debugging speed-up obtained by debugging blocks in parallel depends on the available resources and is generally a scheduling problem that is beyond the scope of this paper. Given that parallelization does not necessarily cut costs (the total effort could be greater for parallel debugging than for sequential debugging), a more independent assessment of the gain is to check whether the total debugging effort is actually reduced by partitioning (see [16, 25] for some first evidence that this may indeed be the case). Intuitively, this is the case if the summed up (wasted) effort for finding the first fault in each of $k$ blocks $B_i$,

$$
\sum_{i=1}^{k} w(e_i, B_i)
$$

obtained by partitioning is less than the (wasted) effort required to find the first $k$ faults in the original FCM without partitioning (and without fixing each fault after it has been found, and re-running the test suite), computed as in Eq. (2).

Fig. 11 shows this speed-up, calculated as the fraction of Eq. (3) and Eq. (2): a speed-up of 2 means that the total wasted effort using partitioning into $k$ blocks is half that of finding $k$

faults without partitioning. It is interesting to note that while BDM, horizontal, and vertical Weil-Kettler perform nearly the same (reaching a speed-up of nearly 2), both Jones et al.’s algorithm and maximum set packing show extreme behaviours (with the former dropping to an average slow-down for more than 3 blocks, corresponding to 8 injected faults). The dominance of maximum set packing over its competitors is impressive; it is the result of lower wasted effort in a greater number of blocks (where the latter leads to more summations in Eq. (3), but also to a rise in the value of Eq. (2) by the factor of $k$).

D. Computational Effort

Application of all partitioning algorithms to the 75,000 FCMs used in our experiments took approx. 3 hours to run on a contemporary PC. By contrast, running the test cases that produced the FCMs took several days. Complexity of the partitioning algorithms is therefore not an issue.

E. Significance of Findings

Because we have no stochastic model of our data (e.g., we cannot assume that they are normally distributed), we must use a non-parametric testing procedure to test the hypothesis that the five algorithms, or rather the data sets they produce, are mutually different. To do this, we use a two-stage procedure described by Conover [9], which first uses Quade’s test to check whether the hypothesis that each sample comes from the same distribution can be rejected with an error probability of $p = .05$. If it is rejected, we know that at least two algorithms perform differently. In this case, Conover proposes a scheme of posthoc-tests to assess for which groups (algorithms) distributional equality cannot be rejected [9]. These tests are two connected sample asymptotic $t$-tests for equality of means of the weighted rank sums Quade’s test uses. If the hypothesis of equality is rejected, one can again consider the two samples to be different.

Using this procedure, and assuming $p = .05$, we find that the hypotheses that all algorithms produce different numbers of blocks, and lead to different expected wasted effort are confirmed in most cases, meaning that with an error probability less than 5%, our results are not the product of chance.

A more detailed presentation of our statistical analyses can be found in the web appendix (see Section IX).
VII. DISCUSSION

Clearly, in terms of speeding up the sequential debugging process, maximum set packing performs best: while the other algorithms cannot demonstrate a larger than twofold speed-up, that of maximum set packing is at least twice as high for up to 32 errors injected (Fig. 11). But also with respect to the degree of parallelism achieved (roughly, i.e., except for spurious failures, corresponding to the number of faults detected by the algorithm; cf. Section III.A) it performs rather well: for 4 and more injected faults, it computes the second highest number of blocks throughout (Fig. 4). The good performance in terms of speed-up comes mostly from cherry picking: since it selects single non-overlapping test cases with small coverage, the wasted effort may be expected to be lower than for the other algorithms, which cluster several test cases into one block, thus arriving at a higher coverage per block. Fig. 5 and Fig. 6 clearly show this for the expected wasted effort (which is based on a random order of inspection of UUTs; see Section VI.B.1); Fig. 7 suggests that the advantage may carry over to fault localization strategies using a more sophisticated (than random) order of inspection. In practice, this superior performance may come at too high a cost, however: since maximum set packing loses a large number of failed test cases (Fig. 9), the diagnostic information that is conveyed in these failures is also lost. In practice, this may turn out to be fatal, since a longer, lost test case may provide a better clue to the error than an included, shorter one. Note that the concomitant high number of lost UUTs (Fig. 10) is less of an issue: it is outweighed by the high number of blocks (which, although they contain fewer faults in total, still have a higher fault density). Note that maximum set packing cannot lead to bug races.

If the debugger to whom a debugging task has been assigned is to be given maximum diagnostic information without introducing bug races, BDM must be chosen. As was to be expected (cf. Section III.A), it achieves the lowest degree of parallelization on average (Fig. 4); yet, should one accept this low degree (perhaps because the capacity available for true parallelization is similarly limited), the wasted effort per block is actually better than for the remaining algorithms (Fig. 6 and Fig. 7). The speed-up in sequential debugging (Fig. 11) can keep up with the other algorithms; yet, one must be aware that this speed-up can be enjoyed only for a comparatively small number of faults (viz. blocks; see above).

Somewhat surprisingly, horizontal Weil-Kettler achieves the highest degree of parallelization, higher than maximum set packing (Fig. 4). The price is clearly the number of bug races this entails (Fig. 8); however, given that the number of bug races per block averages at roughly .15 (meaning that the frequency of a single debugger running into a bug race while debugging a block is 1 in 6), this seems like a low price to pay: even if every sixth block is debugged in vain, the number of remaining blocks is still higher than that obtained for maximum set packing. The speed-up for sequential debugging is lower though than for maximum set packing: it is roughly the same as that for BDM (Fig. 11), yet sustained for a larger number of blocks. Like for BDM, no diagnostic information is lost, which may let horizontal Weil-Kettler appear more attractive than maximum set packing in practical settings.

Somewhat to our disappointment, vertical Weil-Kettler could not demonstrate any particular strengths: although it clearly loses fewer test cases and UUTs than maximum set packing (thus preserving more diagnostic information; Fig. 9 and Fig. 10), it does not perform significantly better than the alternative algorithms in any of the other dimensions.

VIII. THREATS TO VALIDITY

The threats to the validity of our work are basically those described in [26]. In particular, we must concede that due to the extreme dependence of the results on the subject programs chosen (see project data on the web appendix referred to below), our results cannot be generalized to arbitrary programs. However, we have tried to minimize the dependence of our results on other assumptions.

A general threat to the validity of work on automated debugging aids is the large unavailability of real programs in significant numbers with documented faults exposed by failing test cases. Evaluations like ours that do not rely on the eternal Siemens suite therefore usually resort to automated fault injection, or mutation, which, although it may be questioned [6], seems to be acceptable [5, 19].

To counter the argument that in practice, regression tests are divided between unit tests and integration (or system) tests, and that both the outcome of our evaluation and the utility of automated partitioning depend on the kind of tests partitioning is applied to, we maintain that at least for JUnit and the open source domain, the strict division of test cases into unit tests (which should be easier to debug) and integration tests that a separate evaluation would require does not seem to be made.

IX. AVAILABILITY OF ARTEFACTS

All implementations and data sets underlying this work (including diagrams per subject program) can be downloaded from the project website at www.feu.de/ps/prjs/PD.

X. CONCLUSION

If a program contains multiple faults, it may be possible to separate the faults and to debug the program in parallel. Revisiting earlier work of Jones et al. suggesting that debugging speed-ups can be achieved by clustering the set of failed test cases, we have explored various alternatives to increase parallelism, and have shed light on the trade-offs this requires. Results of the large-scale experiments that we have conducted suggest that parallelism can be significantly increased by using algorithms from integer linear programming; yet, hoping for clusters each focusing on a single fault does not seem realistic.

ACKNOWLEDGEMENTS

The authors thank Martin Monperrus for his valuable comments on an earlier version of this paper.

APPENDIX: TUNING JONES ET AL.’S ALGORITHM

The algorithm by Jones, Bowring, and Harrold depends on two thresholds, MostSusp and Sim, which can be tuned for performance (cf. Section II). Since the target of the algorithm is to obtain fault-focusing clusters (ideally: one per fault), an obvious magnitude to minimize is the distance between the number of faults present and the number of clusters derived. Because of the inherent discreteness of the partitioning algorithm, optimization either requires a complete search over the (discretized) two-dimensional threshold space, or a sto-
stochastic approach. The size of the problem (75,000 FCMs to tune) precludes the former, so that we resorted to the (stochastic) Covariance Matrix Adaptation Evolution Strategy, as implemented in the Apache Commons Math library. This suggested the values .51 for Sim (whose original value was .5) and .12 for MostSusp (original value .2).

Using the optimized thresholds in our evaluation of Jones et al.’s algorithm, the number of blocks derived increases by 61% on average. At the same time, however, the number of bug races increases by 103% (suggesting a greater overlap between blocks), so that the average expected wasted effort remains roughly the same (it decreases by 6% on average).

Given that using our optimization, the thresholds have been fitted to our specific data set, we decided to leave them as suggested in [16] for the main evaluation.

References