Concolic Testing of the Multi-sector Read Operation for Flash Memory File System ⋆

Moonzoo Kim[1] and Yunho Kim[1]

CS Dept. KAIST
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr
kimyunho@kaist.ac.kr

Abstract. In today's information society, flash memory has become a virtually indispensable component, particularly for mobile devices. In order for mobile devices to operate successfully, it is essential that flash memory be controlled correctly through file system software. However, as is typical for embedded software, conventional testing methods often fail to detect hidden flaws in the software due to the difficulty of creating effective test cases. As a different approach, model checking techniques guarantee a complete analysis, but only on a limited scale.
In this paper, we describe an empirical study wherein a *concolic testing* method is applied to the multi-sector read operation for a flash memory. This method combines a symbolic static analysis and a concrete dynamic analysis to automatically generate test cases and perform exhaustive path testing accordingly. In addition, we analyze the advantages and weaknesses of the concolic testing approach on the domain of the flash file system compared to model checking techniques.

1 Introduction

Due to attractive characteristics such as low power consumption and strong resistance to physical shock, flash memory has become a crucial component for mobile devices. Accordingly, in order for mobile devices to operate successfully, it is essential that the file system software of the flash memory operates correctly. However, conventional testing methods often fail to detect hidden bugs in the file system software for flash memory, since it is very difficult to create effective test cases that provide a check of all possible execution scenarios generated from the complex file system software. Thus, the current industrial practice of manual testing does not achieve high coverage or provide cost-effective testing. In another testing approach, randomized testing can save human effort for test case generation, but does not achieve high coverage, because random input data does not necessarily guarantee high coverage of a target program.

⋆ This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/Korea Science and Engineering Foundation(KOSEF) (grant number R11-2008-007-03002-0) and the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2009-(C1090-0902-0032)).

These deficiencies of conventional testing incur significant overhead to manufacturers. In particular, ensuring reliability and performance are the two most time-consuming tasks to produce high quality embedded software. For example, a multi-sector read (MSR) function was added to the flash software to improve the reading speed of a Samsung flash memory product [2]. However, this function caused numerous errors in spite of extensive testing and debugging efforts, to the extent that the developers seriously considered removing the feature. Considering that MSR is a core logic used for most flash software with variations, and that improvement of the reading speed through MSR can provide important competitive power to flash memory products, research on the effective analysis of MSR is desirable and practically rewarding.

In spite of the importance of flash memory, however, little research work has been conducted to formally analyze flash file systems. In addition, most of such work [8, 10, 4] focuses on the specification of file system design, not real implementation. In this paper, we describe experiments we carried out to analyze the MSR code of the Samsung flash file system using CREST [12], an open source *concolic testing* [22, 20, 5] tool for C programs. With a given compilable target C code, a concolic (CONCrete + symbOLIC) testing combines both a concrete dynamic analysis and a symbolic static analysis [13, 23] to *automatically* generate test cases that achieve high coverage. However, it is necessary to check the effectiveness of concolic testing on a flash file system through empirical studies, since the success of this testing approach depends on the characteristics of the target program under test. MSR has complex environmental constraints between sector allocation maps and physical units for correct operation (see Section 2.2) and these constraints may cause insufficient coverage and/or high runtime cost for the analysis when concolic testing is applied.

Furthermore, we compare the empirical results obtained from analyzing MSR through concolic testing with those yielded by model checking [9]. As an alternative solution to achieve high reliability, model checking guarantees complete analysis results; the authors reported on the effectiveness of model checking for the verification of MSR in [15]. However, model checking has a limitation with respect to scalability, and thus the analysis results can be applied on a small scale only. Thus, comparison of these two different techniques to analyze MSR can clearly show their relative strengths and weaknesses and will serve as a basis for developing an advanced analysis technique suitable for flash file systems.

The organization of this paper is as follows. Section 2 overviews the file system for the flash memory and describes multi-sector operation in detail. Section 3 briefly explains the concolic testing algorithm. Section 4 describes the experimental results obtained by applying concolic testing to MSR. Section 5 discusses observations from the experiments. Section 6 concludes the paper with directions for future work.

2 Overview of Multi-sector Read Operation

Unified storage platform (USP) is a software solution to operate a Samsung flash memory device [2]. USP allows applications to store and retrieve data on flash memory through a file system. USP contains a flash translation layer (FTL) through which data and programs in the flash memory device are accessed. The FTL consists of three layers

- a sector translation layer (STL), a block management layer (BML), and a low-level device driver layer (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. MSR resides in STL. [1]

2.1 Overview of Sector Translation Layer (STL)

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*.

When new data is written to flash memory, rather than overwriting old data directly, the data is written on empty physical sectors and the physical sectors that contain the old data are marked as invalid. Since the empty physical sectors may reside in separate physical units, one logical unit (LU) containing data is mapped to a linked list of physical units (PU). STL manages this mapping from logical sectors (LS) to physical sectors (PS). This mapping information is stored in a sector allocation map (SAM), which returns the corresponding PS offset from a given LS offset. Each PU has its own SAM.

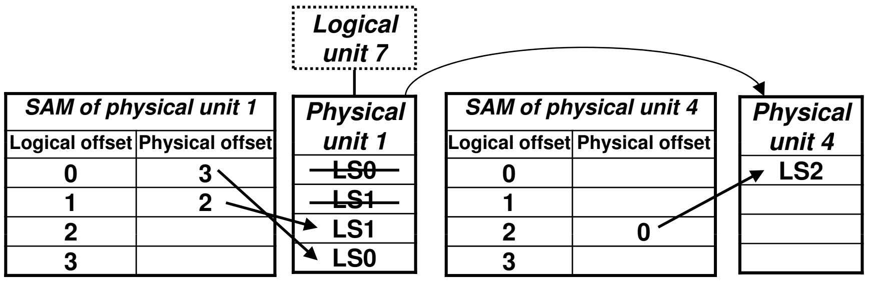


Fig. 1. Mapping from logical sectors to physical sectors

Figure 1 illustrates a mapping from logical sectors to physical sectors where 1 unit consists of 1 block and 1 block contains 4 pages, each of which consists of 1 sector. Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into PS0 of PU1 (SAM1[0]=0). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of PU1 (SAM1[1]=1). The user then updates LS1 and LS0 in order, which results in SAM1[1]=2 and SAM1[0]=3. Finally, the user adds LS2 of LU7, which adds a new physical unit PU4 to LU7 and yields SAM4[2]=0.

2.2 Multi-sector Read Operation

USP provides a mechanism to simultaneously read as many multiple sectors as possible in order to improve the reading speed. The core logic of this mechanism is implemented

[1] This section is taken from [15].

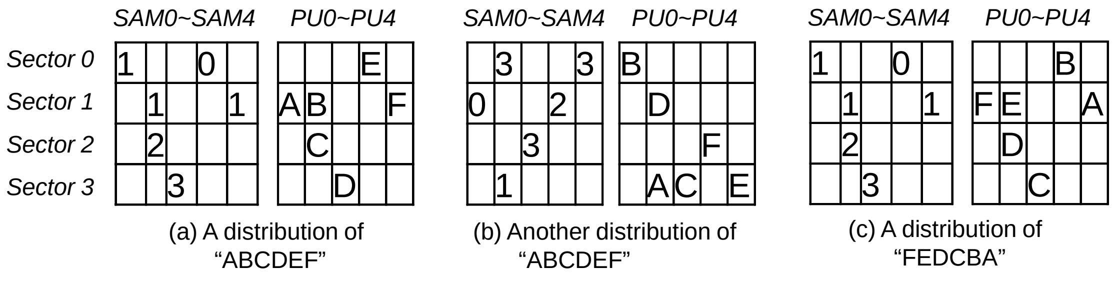
in a single function in STL. Due to the non-trivial traversal of data structures for logical-to-physical sector mapping (see Section 2.1), the function for MSR is 157 lines long and highly complex, having 4-level nested loops. Figure 2 describes simplified pseudo code of these 4-level nested loops. The outermost loop iterates over LUs of data (line 2-18) until the `numScts` amount of the logical sectors are read completely. The second outermost loop iterates until the LS's of the current LU are completely read (line 5-16). The third loop iterates over PUs mapped to the current LU (line 7-15). The innermost loop identifies consecutive PS's that contain consecutive LS's in the current PU (line 8-11). This loop calculates `conScts` and `offset`, which indicate the number of such consecutive PS's and the starting offset of these PS's, respectively. Once `conScts` and `offset` are obtained, `BML_READ` rapidly reads these consecutive PS's as a whole (line 12).

```
01:curLU = LU0;
02:while(numScts > 0) {
03:  readScts = # of sectors to read in the current LU
04:  numScts -= readScts;
05:  while(readScts > 0 ) {
06:    curPU = LU->firstPU;
07:    while(curPU != NULL ) {
08:      while(...) {
09:        conScts = # of consecutive PS's to read in curPU
10:        offset = the starting offset of these consecutive PS's
11:      }
12:      BML_READ(curPU, offset, conScts);
13:      readScts = readScts - conScts;
14:      curPU = curPU->next;
15:    }
16:  }
17:  curLU = curLU->next;
18:}
```

Fig. 2. Loop structures of MSR

For example, suppose that the data is "ABCDEF" and each unit consists of four sectors and PU0, PU1, and PU2 are mapped to LU0 ("ABCD") in order and PU3 and PU4 are mapped to LU1 ("EF") in order, as depicted in Figure 3(a). Initially, MSR accesses SAM0 to find which PS of PU0 contains LS0('A'). It then finds SAM0[0]=1 and reads PS1 of PU0. Since SAM0[1] is empty (i.e., PU0 does not have LS1('B')), MSR moves to the next PU, which is PU1. For PU1, MSR accesses SAM1 and finds that LS1('B') and LS2('C') are stored in PS1 and PS2 of PU1 consecutively. Thus, MSR reads PS1 and PS2 of PU1 altogether through `BML_READ` and continues its reading operation.

The requirement for MSR is that the content of the read buffer should be equal to the original data in the flash memory when MSR finishes reading, as given by `assert(` $\forall i$`.LS[`i`]==buf[`i`])` inserted at the end of MSR.



SAM0~SAM4
PU0~PU4
Sector 0
Sector 1
Sector 2
Sector 3
(a) A distribution of "ABCDEF"
(b) Another distribution of "ABCDEF"
(c) A distribution of "FEDCBA"

Fig. 3. Possible distributions of data "ABCDEF" and "FEDCBA" to physical sectors

In these analysis tasks, we assume that each sector is 1 byte long and each unit has four sectors. Also, we assume that data is a fixed string of distinct characters (e.g., "ABCDE" if we assume that data is 5 sectors long, and "ABCDEF" if we assume that data is 6 sectors long). We apply this data abstraction, since the values of logical sectors should not affect the reading operations of MSR, but the distribution of logical sectors into physical sectors does. For example, for the same data "ABCDEF", the reading operations of MSR are different for Figure 3(a) and Figure 3(b), since they have different SAM configurations (i.e. different distributions of "ABCDEF"). However, for "FEDCBA" in Figure 3(c), which has the same SAM configuration as the data shown in Figure 3(a), MSR operates in exactly same manner as for Figure 3(a). Thus, if MSR reads "ABCDEF" in Figure 3(a) correctly, MSR reads "FEDCBA" in Figure 3(c) correctly too.

In addition, we assume that data occupies 2 logical units. The number of possible distribution cases for l LS's and n physical units, where $5 \leq l \leq 8$ and $n \geq 2$, increases exponentially in terms of both n and l, and can be obtained by

$$\sum_{i=1}^{n-1}({}_{(4\times i)}C_4 \times 4!) \times ({}_{(4\times(n-i))}C_{(l-4)} \times (l-4)!)$$

For example, if a flash has 1000 physical units with data occupying 6 LS's, there exist a total of 3.9×10^{22} different distributions of the data. Table 1 shows the total number of possible cases for 5 to 8 logical sectors and various numbers of physical units, respectively, according to the above formula.

MSR has characteristics of a control-oriented program (4-level nested loops) and a data-oriented program (large data structure consisting of SAMs and PUs) at the same time, although the values of PS's are not explicitly manipulated. As seen from Figure 3, the execution paths of MSR depend on the values of SAMs and the order of PUs linked to LU. In other words, MSR operates deterministically, once the configuration of the SAMs and PUs is fixed.

PUs	4	5	6	7	8
$l = 5$	61248	290304	9.8×10^5	2.7×10^6	6.4×10^6
$l = 6$	239808	1416960	5.8×10^6	1.9×10^7	5.1×10^7
$l = 7$	8.8×10^5	7.3×10^6	3.9×10^7	1.5×10^8	5.0×10^8
$l = 8$	3.4×10^6	4.2×10^7	2.9×10^8	1.4×10^9	5.6×10^9

Table 1. Total number of the distribution cases

3 Overview of the Concolic Testing Approach

This section presents an overview of the concolic testing algorithm [22, 20, 5]. The concolic testing algorithm executes a target program both concretely and symbolically [13, 23] at the same time. Note that the symbolic path is built following the path that the concrete execution takes. The concolic testing algorithm proceeds in the following five steps:

1. *Instrumentation*
   A target C program is statically instrumented with probes, which record symbolic path constraints from a concrete execution path when the target program is executed.
2. *Concrete execution*
   The instrumented C program is executed with given input values and the concrete execution part of the concolic execution constitutes the normal execution of the program. For the first execution of the target program, initial input values are assigned with random values. For the second execution and onward, input values are obtained from step 5.
3. *Symbolic execution*
   The symbolic execution part of the concolic execution collects symbolic constraints over the symbolic input values at each branch point encountered along the concrete execution path. Whenever each statement S_j of the original target program is executed, a corresponding probe P_j inserted at S_j updates the symbolic map of symbolic variables if S_j is an assignment statement, or collects a corresponding symbolic path constraint C_j if S_j is a branch statement. Thus, a complete symbolic path formula ϕ_i is built at the end of the ith execution by combining all path constraints C_j's.
4. *Deciding the next execution path*
   Given a symbolic path formula ϕ_i obtained in step 3, ϕ_{i+1} (the next execution path to test) is created by negating one path constraint C_j. For example, if depth first search (DFS) is used, ϕ_{i+1} is generated by negating the last symbolic path constraint of ϕ_i. If there is no further new paths to test, the algorithm terminates.
5. *Selecting the next input values*
   A constraint solver such as a Satisfiability Modulo Theory (SMT) solver [3] generates a model that satisfies ϕ_{i+1}. This model assigns concrete values to input values and the whole concolic testing procedure iterates from stage 2 again with these input values.

Note that the above algorithm does *not* raise any false alarms, since it executes a concrete path. However, there is a clear limitation in step 5. A constraint solver cannot solve complex path formulas to compute concrete values; most constraint solvers cannot handle statements containing arrays, pointers, and non-linear arithmetic. To address this difficulty, symbolic constraints are simplified by replacing some of the symbolic values with concrete values, which may result in incomplete coverage.

4 Empirical Study on Concolic Testing MSR

In this section, we describe two series of experiments for concolically testing MSR, both of which target the same MSR code, but with different environment models - a *constraint-based model* and an *explicit model*. Our hypotheses are as follows:

- H_1: Concolic testing is effective for analyzing the MSR code
- H_2: Concolic testing is more efficient than model checking for analyzing the MSR code

Regarding H_1, we expect that concolic testing can detect bugs effectively, since it tries to explore all feasible execution paths. For H_2, considering that model checking analyzes all possible value combinations of variables, concolic testing may analyze the MSR code faster (note that different value combinations of variables may execute a same path).

4.1 Testbed for the Experiments

All experiments were performed on 64 bit Fedora Linux 9 equipped with a 3 GHz Core2Duo processor and 16 gigabytes of memory. We used CREST [1] as a concolic testing tool for our experiments, since it is an open source tool and we could obtain more detailed experimental results by modifying the CREST source code for our purposes. However, since the CREST project is in its early stage, CREST has several limitations such as lack of support for dereferencing of pointers and array index variables in the symbolic analysis. Consequently, the target MSR code was modified to use an array representation of the SAMs and PUs. We used CREST 0.1.1 (with DFS search option), gcc 4.3.0, Yices 1.0.19 [6], which is a SMT solver used as an internal constraint solver by CREST for solving symbolic path formulas.

For model checking experiments, CBMC 2.6 [7] and MiniSAT 1.14 [18] were used. The target MSR codes used for concolic testing and model checking are identical, except nominal modification replacing the assumption statements in CBMC experiments with `if` statements to terminate testing if the assumptions are evaluated false (i.e. invalid test cases (see Section 4.2)). Model checking experiments were performed on the same testbed as that of concolic testing experiments.

To evaluate the effectiveness of concolic testing, we applied *mutation analysis* [14] by injecting the following three types of frequently occuring bugs (i.e. mutation operators), each of which has three instances:

1. *Off-by-1 bugs*

- b_{11}: `while(numScts>`**0**`)` of the outermost loop (line 2 of Figure 2) to `while(numScts>`**1**`)`
- b_{12}: `while(readScts>`**0**`)` of the second outermost loop (line 5 of Figure 2) to `while(readScts>`**1**`)`
- b_{13}: `for(i=0;i<conScts; i++)` of `BML_READ()` (line 12 of Figure 2) to `for(i=0;i<conScts`**-1**`;i++)`

2. *Invalid condition bugs*

- b_{21}: `if(SAM[i].offset[j]`**!=**`0xFF)` in the third outermost loop to `if(SAM[i].offset[j]`**==**`0xFF)`
- b_{22}: `readScts=((4-j)>numScts)?numScts:4-j` in the innermost loop to `readScts=((4-j)<numScts)?numScts:4-j`
- b_{23}: `if((firstOffset+nScts)`**==**`SAM[i].offset[j])` in the innermost loop to `if((firstOffset+nScts)`**!=**`SAM[i].offset[j])`

3. *Missing statement bugs*

- b_{31}: missing `nScts=1` in the second outermost loop
- b_{32}: missing `nReadScts--` in the second outermost loop
- b_{33}: missing `nLun++` corresponding the line 17 of Figure 2

Furthermore, we injected an artificial corner case bug b_c by changing line 13 of Figure 2 as follows:

```
readScts = readScts - conScts -
(PU[1].sect[3]=='A' && PU[0].sect[0]=='B' && PU[2].sect[3]=='C'
&& PU[1].sect[1]=='D' && PU[4].sect[3]=='E' && PU[3].sect[2]=='F')
```

Note that b_c causes an error only when the configuration of the PUs and SAMs satisfies the given condition illustrated in Figure 1.(b). b_c is very hard to detect, since the probability of detecting b_c through testing is extremely low (e.g. $7\times10^{-8} = 1/1416960$ when 6 logical sectors are distributed over 5 PUs (see Table 1)). Therefore, although b_c is not a realistic bug, the effectiveness of concolic testing can be shown more clearly by detecting b_c.

4.2 Experiments with a Constraint-based Environment Model

Constraint-based Environment Model As described in Section 2.2, a test case for MSR is a configuration of SAMs and PUs (see Figure 3). MSR assumes randomly written logical data on PUs and a corresponding SAM records the actual location of each LS. Unfortunately, however, the writing is *not* purely random, but is subject to several constraint rules; the following are some of the representative rules applied to the random writing. For example, the last two rules can be enforced by the constraints in Figure 4.

1. One PU is mapped to at most one LU.
2. If the i_{th} LS is written in the k_{th} sector of the j_{th} PU, then the $(i\ mod\ m)_{th}$ offset of the j_{th} SAM is valid and indicates the PS number k, where m is the number of sectors per unit (4 in our experiments).

3. The PS number of the i_{th} LS must be written in *only* one of the $(i\,mod\,m)_{th}$ offsets of the SAM tables for the PUs mapped to the $\lfloor \frac{i}{m} \rfloor_{th}$ LU.

$$\begin{aligned}\forall i, j, k\ (LS[i] = PU[j].sect[k] \rightarrow (&SAM[j].valid[i\,mod\,m] = true\\ &\&\ SAM[j].offset[i\,mod\,m] = k\\ &\&\ \forall p.(SAM[p].valid[i\,mod\,m] = false)\\ &\text{ where } p \neq j \ \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th}\ LU))\end{aligned}$$

Fig. 4. Environment constraints for MSR

If a given configuration of SAMs and PUs satisfies the constraints, this configuration is *valid*; invalid, otherwise. It is important to check whether a given test case is valid or not, since an invalid test case may produce an incorrect testing result. Therefore, for accurate unit testing, it is essential to provide a precise environment model to feed valid test cases only.

To enforce the constraint-based environment model on the test cases, all elements of the SAM tables and PUs are declared to be analyzed symbolically through `CREST_unsigned_char(PU[i].sect[j])` and `CREST_unsigned_char(SAM[i].offset[j])` statements for all valid `i` and `j`. Then, a test driver/environment model checks whether concrete values assigned by CREST to those variables satisfy the constraints in Figure 4. If not, the execution terminates immediately without testing MSR. Note that these constraints are encoded as `if` statements in nested loops handling universally quantified i, j, k, and p, which results in a complex environment model.

Experimental Results Due to a time limitation, we could perform 4 experiments with 4 to 5 PUs with 5 to 6 LSes. The total numbers of test cases generated and the ratios of the valid test cases over the total test cases are depicted in Figure 5. For example, CREST generated a total of 5.6×10^5 test cases for 4 PUs with 5 LSes, and only 61248 test cases (around 11% of the total test cases) among them were valid. Note that the numbers of the valid test cases for these 4 experiments are equal to the numbers of all possible configurations of the SAMs and PUs (see Table 1). This means that the concolic testing covers all possible execution scenarios of MSR. [2] Consequently, all injected bugs b_{11} to b_{33} as well as b_c were detected; most of them were detected in a few seconds through the first few hundred test cases.

The performance of the concolic testing is shown in Figure 6. For example, CREST took 2594 seconds for the experiments with 4 PUs and 5 LSes. The amount of time to

[2] We tried to perform the same experiments with CUTE (32 bit binary) [22] but failed; CUTE crashed after consuming 4 gigabytes of memory at the constraint solving step at the third iteration. We could not continue the experiments with CUTE, since neither the source code nor user support was available.

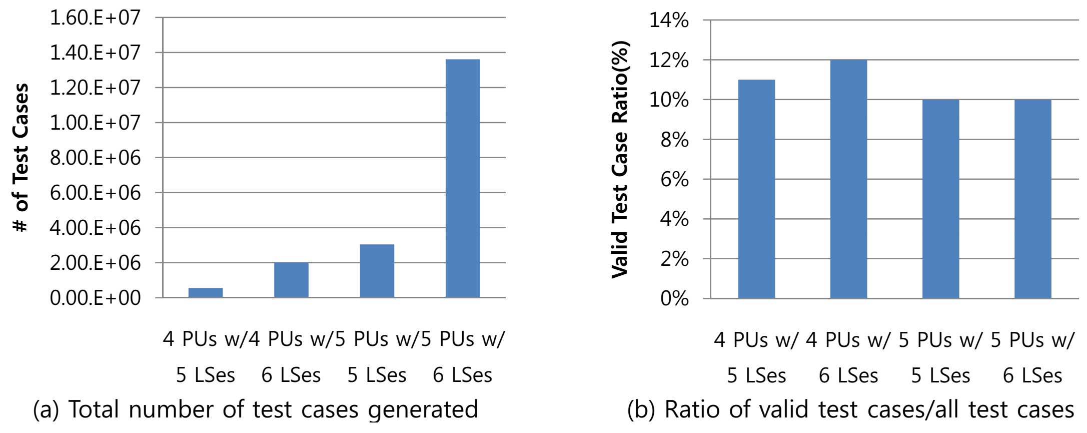


Fig. 5. Generated test cases with constraint-based environment model

analyze MSR increases exponentially in terms of the number of PUs and LSes. Figure 6(a) shows that CREST is several hundred times slower than CBMC. Figure 6.(b) shows that symbolic execution, Yices, and system execution (e.g. launching a target program) take around 40%, 40%, and 20% of the total execution time. However, all experiments use around 10 megabytes of memory only, since the DFS search in CREST needs only a small amount of information regarding the previous execution path, not the whole execution tree. In comparison, CBMC consumed 40 megabytes and 89 megabytes for 4 PUs with 5 LSes and 5 PUs with 6 LSes, respectively. Therefore, the memory bottleneck problem associated with model checking does not exist for concolic testing.

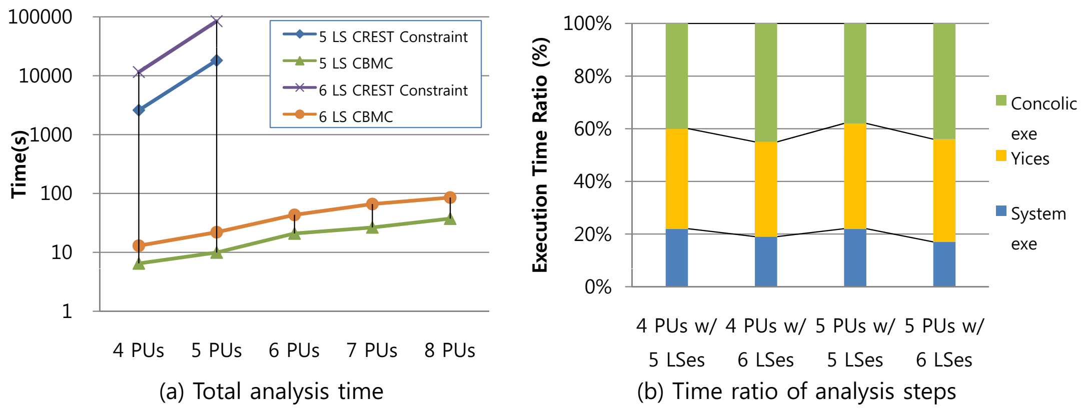


Fig. 6. Analysis time with constraint-based environment model

4.3 Experiments with an Explicit Environment Model

Explicit Environment Model As we have seen from Figure 5(b), the constraint-based environment model generated too many invalid test cases. Thus, we decided to use an explicit environment model that generates test cases *explicitly* by selecting a PU and its sector to contain the l th logical sector (`PU[i].sect[j]=LS[l]`) and setting the corresponding SAM accordingly (`SAM[i].offset[l]=j`). Therefore, most of the generated test cases satisfy the constraints between SAMs and PUs.

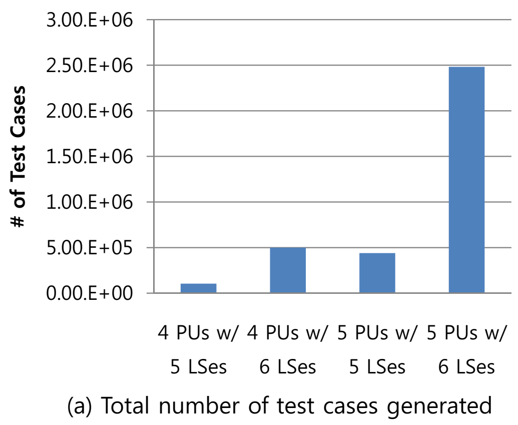
However, since CREST cannot support accessing array elements through a symbolic array index variable, we have to modify assignments of SAMs and PUs in the environment model so that these assignments access array elements through constants, not index variables. This workaround solution is depicted in Figure 7. `idxPU` and `idxSect`, which indicate the physical location of the ith logical sector data (`LS[i]`), are declared to be handled symbolically (lines 3 and 4). In the explicit environment model, the `switch` statements starting at line 9 and line 10/17 respectively handle `idxPU` and `idxSect` case by case. Note that, although this explicit environment model does not generate many invalid test cases, it increases the total number of execution paths due to these additional `switch` statements.

```
01:for (i=0; i< NUM_LS; i++){
02:  unsigned char idxPU, idxSect;
03:  CREST_unsigned_char(idxPU);
04:  CREST_unsigned_char(idxSect);
05:  ...
06:  //The switch statements encode the following two statements:
07:  //  PU[idxPu].sect[idxSect]= LS[i];
08:  //  SAM[idxPu].sect[i]= idxSect;
09:  switch(idxPU){
10:    case 0: switch(idxSect) {
11:            case 0: PU[0].sect[0] = LS[i];
12:                    SAM[0].offset[i] = idxSect; break;
13:            case 1: PU[idxPU].sect[1] = LS[i];
14:                    SAM[0].offset[i] = idxSect; break;
15:            ...  }
16:            break;
17:    case 1: switch(idxSect) {
18:             ...
```

Fig. 7. Explicit environment model for MSR

Experimental Results Due to a time limitation, we could perform only 4 experiments with 4 to 5 PUs with 5 to 6 LSes with the explicit environment model. The total numbers of test cases generated and the ratios of the valid test cases over the total test cases are depicted in Figure 8. For example, CREST generated a total of 10^5 test cases for 4

PUs with 5 LSes, 61248 test cases (around 60% of the total test cases) among them being valid. Thus, the explicit environment model generates test cases more efficiently compared to the constraint-based model. Similar to the experiments with the constraint-based model, the numbers of valid test cases for these 4 set of experiments are equal to the numbers of all possible configurations of the SAMs and PUs (see Table 1). All injected bugs b_{11} to b_{33} and b_c were detected, but within fewer test cases; most of them were detected in 3 seconds through the first 50 test cases.



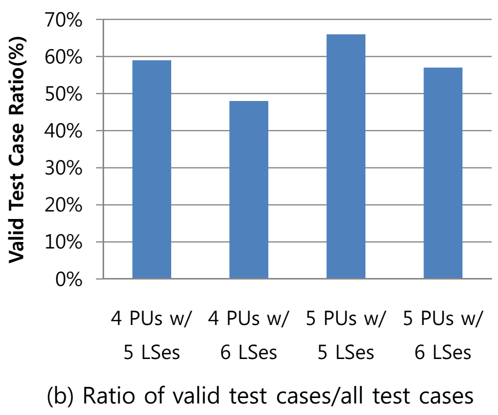


Fig. 8. Statistics on the generated test cases with explicit environment model

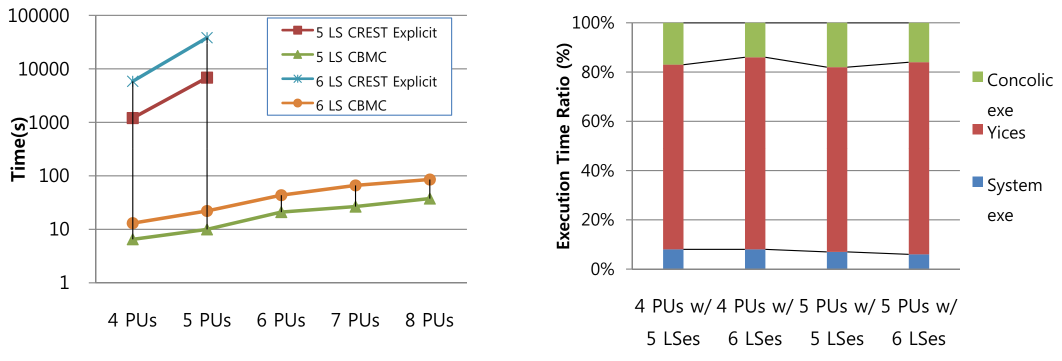
The performance of the concolic testing approach with the explicit environment model is depicted in Figure 9. For example, CREST took 1203 seconds for the experiments with 4 PUs and 5 LSes. Although the concolic testing with the explicit model is twofold faster than the testing with the constraint-based model, it is still a hundred times slower compared to CBMC (see Figure 6). Yices takes around 75% of the total execution time, since invalid test cases are significantly reduced, which thus decreases the portion of symbolic execution time and system execution time. Note that the symbolic execution path formulas for invalid test cases are very short and are solved quickly. Therefore, improvement of the SMT solver is an important issue with regard to the success of concolic testing.

5 Discussion

In this section, several issues are discussed on the basis of our experience of applying concolic testing to MSR.

5.1 Weaknesses of Concolic Testing

Although our hypothesis H_1 is accepted through the empirical study (i.e. the concolic testing method demonstrates capability of detecting bugs through high coverage), H_2 is



(a) Total analysis time (b) Time ratio of analysis steps

Fig. 9. Analysis time with explicit environment model

rejected (i.e. its performance on MSR is worse than the performance of model checking MSR by CBMC (see Figure 6 and Figure 9)). This poor performance was caused by several steps of the concolic testing algorithm (see Section 3).

First, for a target program with a complex environmental model such as MSR, the concolic testing wastes a large amount of time to generate invalid test cases. In the experiments with the constraint-based environment model and the explicit environment model, around 90% and 45% of the total test cases generated were invalid respectively (see Figure 5 and Figure 8). Considering a unit under testing often has preconditions or constraints enforced by its interacting components, the concolic testing framework should provide an efficient way to control the generation of concrete input values so as to generate only valid test cases. Second, concolic execution (see steps 2 and 3 of the algorithm in Section 3) causes high overhead, since each original C statement is supplemented with a probe reflecting a concrete execution in a symbolic manner; around 40% and 15% of the total execution times were spent for the concolic executions with the constraint-based model and the explicit model, respectively (see Figure 6 and Figure 9). Note that the original MSR code takes less than 0.1% of the concolic execution time. Lastly, the performance of the constraint solver Yices was slow, although the path formulas of MSR are conjunctions of only linear arithmetic conditions and can be solved rapidly by many efficient algorithms [21]. Therefore, from our experiments, we can conclude that CREST needs to be improved for practical usage.

5.2 Importance of an Environment Model

Through the various experiments carried out to analyze MSR, including conventional testing [16], concolic testing, and model checking [15], we found that it is important to build an accurate and efficient environment model for the analysis of a flash file system. Also, it was found that different analysis techniques can commonly use the same environment model. For example, the constraint-based environmental model (see Section 4.2) was originally designed for model checking through CBMC and used as is with only nominal modification. Similarly, the explicit environmental model was originally designed for model checking through SPIN [11]. We used this environmental model for SPIN with slight modification due to the limitation of CREST (i.e., array index variables are not symbolically handled). Furthermore, the design of the environment model substantially affects the analysis performance (see Section 4.2 and Section 4.3).

Considering the importance of an environment model in unit testing, the claim of automated test case generation by concolic testing is only partially true, since an experienced user has to build an environment model.

5.3 Comparison with Model Checking

Concolic testing can be considered as a light-weight model checking method, since it generates all test cases corresponding to all possible execution paths. However, these two different analysis techniques have as many different characteristics as common characteristics. Table 2 compares these techniques briefly based on our experience, although this comparison result might not be applicable to other target programs.

In general, model checking provides better accuracy, since the coverage of concolic testing may not be complete if a target program contains complex statements that cannot be solved by a constraint solver (note that this was not the case for MSR). Also, constraint solvers used for concolic testing are not sufficiently advanced to manipulate symbolic execution path formulas efficiently. However, in terms of applicability, concolic testing has notable advantages, since it can analyze a target program with underlying binary libraries as it is, without manual abstraction, which is necessary for model checking.

	Accuracy	Analysis speed	Memory usage	User effort	Applicability
Concolic testing	High	Slower	Low	Middle	High
Model checking	Highest	Slow	High	High	Low

Table 2. Comparison of concolic testing and model checking

5.4 Hard Characteristics of MSR for Concolic Testing

It was found that MSR is a hard instance for concolic testing. Concolic testing can efficiently analyze programs whose data domain can be significantly abstracted. For example, concolic testing can analyze binary search programs or sort programs quickly. The data domain of MSR (especially SAMs), however, cannot be abstracted, since every different value in every single element of SAMs leads to a unique execution path. Thus, as shown in Section 4.2 and Section 4.3, the total number of valid test cases generated is exactly the same as the number of all possible configurations of the PUs and SAMs (see Table 1). In other words, in the analysis of MSR, concolic testing is burdened by as much complexity as model checking. The same difficulty in analysis of MSR applies to model checking and a scalability issue remains.

6 Conclusion and Future Work

We reported our experience of applying a concolic testing method to analyze the MSR code, a complex unit of a flash file system, and analyzed the strengths and weaknesses of the approach empirically. Although several goals of the concolic testing method could be achieved through the experiments (e.g., automated test case generation, high coverage, and detection of bugs), CREST suffered from a few limitations including slow analysis speed and lack of support for array index variables. We expect that CREST will be able to overcome these limitations in the near future.

As future study, we plan to build a flash file system model that can be used by file-system-dependent applications in a concolic testing framework. One inspiring related work was carried out by Microsoft [17], where an intelligent mock object (an environment model in our terminology) for a file system was developed to test target applications in the PEX framework [19]. The mock file system automatically generates various possible test cases necessary to test applications, which can save significant effort to test file-system-dependent applications.

Acknowledgments

We would like to thank Hotae Kim at Samsung Electronics and Prof. Yunja Choi at Kyungpook National University for their valuable discussion on the environment models for flash file systems.

References

1. CREST - automatic test generation tool for C. `http://code.google.com/p/crest/`.
2. Samsung OneNAND fusion memory. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html`.
3. SMT-LIB: The satisfiability module theories library. http://combination.cs.uiowa.edu/smtlib/.
4. A.Butterfield, L.Freitas, and J.Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming*, 74(4), Feb 2009.
5. C.Cadar, D.Dunbar, and D.Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation (OSDI)*, 2008.
6. B. Dutertre and L. Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification (CAV)*, 2006.
7. E.Clarke, , D.Kroening, and F.Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
8. E.Kang and D.Jackson. Formal modeling and analysis of a flash filesystem in alloy. In *Abstract state machines, B and Z*, 2008.
9. E.M.Clarke, O.Grumberg, and D.A.Peled. *Model Checking*. MIT Press, January 2000.
10. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying intel flash file system core specification. In *4th VDM-Overture Workshop*, 2008.

11. G. J. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.
12. J.Burnim and K.Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.
13. J.C.King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.
14. J.H.Andrews, L.C.Briand, and Y.Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, 2005.
15. M. Kim, Y. Choi, Y. Kim, and H. Kim. Formal verification of a flash memory device driver - an experience report. In *SPIN Workshop*, 2008.
16. M.Kim, Y.Kim, Y.Choi, and H.Kim. Pre-testing flash device driver through model checking techniques. In *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2008.
17. M.Marri, T.Xie, N.Tillmann, J.de Halleux, and W.Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test*, 2009.
18. N.Een and N.Sorensson. An extensible sat-solver. In *SAT 2003*, 2003.
19. N.Tillmann and W.Schulte. Parameterized unit tests. In *European Software Engineering Conference/Foundations of Software Engineering*, 2005.
20. P.Godefroid, N.Klarlund, and K.Sen. Dart: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, 2005.
21. S.Berezin, V.Ganesh, and D.L.Dill. An online proof-producing decision procedure for mixed integer linear arithmetic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.
22. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference/Foundations of Software Engineering*, 2005.
23. W.Visser, C.S.Pasareanu, and S.Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis*, 2004.