# Pre-testing Flash Device Driver through Model Checking Techniques

Moonzoo Kim and Yunho Kim
CS Dept. Korea Advanced Institute of Science and Technology
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr and kimyunho@kaist.ac.kr

Yunja Choi
School of EECS, Kyungpook National University
Daegu, South Korea
yuchoi76@knu.ac.kr

Hotae Kim
Software Laboratories Samsung Electronics
Suwon, South Korea
hotae.kim@samsung.com

## Abstract

*Flash memory has become virtually indispensable in most mobile devices, such as mobile phones, digital cameras, mp3 players, etc. In order for mobile devices to successfully provide services, it is essential that flash memory be controlled correctly through the device driver software. However, as is typical for embedded software, conventional testing methods often fail to detect hidden flaws in the complex device driver software. This deficiency incurs significant development and operation overhead to the manufacturers. As a complementary approach to improve the reliability of embedded software, model checking provides a complete analysis of a target model but the size of the target software is limited due to the state explosion problem.*

*In this project, we have verified the correctness of a multi-sector read operation of Samsung OneNAND*$^{TM}$ *flash device driver by using both model checking and testing. We started the verification task with the model checkers NuSMV and Spin for an exhaustive analysis of a small size flash as a pre-testing step. We then set up a testbed based on a formal model used for model checking and performed testing on a large size flash. Through these verification tasks, we could successfully verify the correctness of the multi-sector read operation with both complete exploration of model checking and scalability of testing.*

## 1 Introduction

Among the various storage platforms, flash memory has become the most popular choice for mobile devices owing to its good characteristics such as low power consumption and strong resistance to physical shock. Thus, in order for mobile devices to successfully provide services to users, it is essential that the device driver of the flash memory operates correctly. However, as is typical of embedded software, conventional testing methods often fail to detect hidden bugs in the device driver software for flash memory since it is infeasible to test all possible scenarios generated from the complex control structure of the device driver. This deficiency incurs significant overhead to the manufacturers. For example, Samsung spent more project time and resources to test flash software than in developing the software.

Limitations of conventional testing were manifest in the development of flash software for Samsung OneNAND$^{TM}$ flash memory [1]. For example, a multi-sector read function was added to the flash software to optimize the reading speed (see Section 3.1). 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. Model checking can be a complementary technique to address the above mentioned weakness of testing through exhaustive analyses. However, due to the state space explosion problem, the size of the model that can be verified using a model checker is limited.

In this project, we have verified the correctness of a multi-sector read operation of a Samsung OneNAND flash device driver by using both model checking and testing. First, we started the verification task with the model checkers NuSMV [5] and Spin [10] for an exhaustive analysis of a small size flash as a pre-testing step. Then, based on the formal model used in model checking, we setup a testbed for both exhaustive testing on a small size flash and randomized testing on a large size flash. Our experience shows that model checking and testing can be effectively used together and this approach efficiently provides more complete analysis results. Through these verification tasks, we successfully verified the correctness of the multi-sector read operation with both complete exploration of model checking and scalability of testing.

## 2 Overview of the OneNAND Verification Project

In this section, we overview the device driver software for OneNAND flash memory.

### 2.1 Overview of the Device Driver Software for OneNAND Flash Memory

OneNAND is a single chip comprising a NOR flash interface, a NAND flash controller logic, NAND flash array, and a small internal RAM. OneNAND provides a NOR interface through its internal RAM. When an application executes a program in OneNAND, the corresponding page of the program is loaded into the RAM in OneNAND by the demand paging manager (DPM) for XIP (execution in place).

Unified storage platform (USP) is a software solution for OneNAND based mobile embedded systems. Figure 1 presents an overview of USP. It manages both code storage and data storage. USP allows applications to store and retrieve data on OneNAND through a file system. USP contains a flash translation layer (FTL) through which data and programs in the OneNAND 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 (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. A prioritized read request for executing a program is made by DPM and this request goes to BML directly. Although USP allows concurrent I/O requests from multiple applications through STL, BML operations must be executed sequentially, not concurrently. For this purpose, BML uses a binary semaphore to coordinate concurrent I/O requests from STL. Furthermore, a prioritized read request from DPM can preempt generic I/O operations requested from STL. Thus, it is important to guarantee the correctness of I/O operations in concurrent settings.

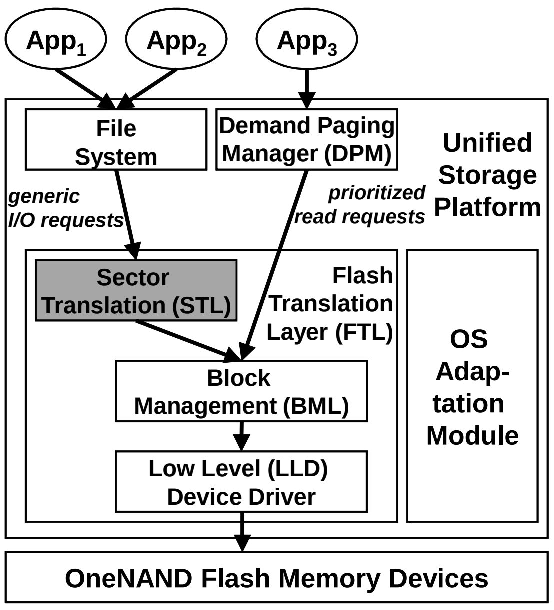

App1
App2
App3
File System
Demand Paging Manager (DPM)
Unified Storage Platform
generic I/O requests
prioritized read requests
Sector Translation (STL)
Flash Translation Layer (FTL)
Block Management (BML)
Low Level (LLD) Device Driver
OS Adap-tation Module
OneNAND Flash Memory Devices

**Figure 1. An overview of USP**

### 2.2 Overview of Sector Translation (STL)

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be multiple of, or equal to a block. Each page contains a set of *sectors*. When new data is written to flash memory, the data is written on empty physical sectors and the physical sectors that contain the old data are marked invalid rather than overwriting old data directly. Since the empty physical sectors may reside in separate physical units (PU), one logical unit (LU) containing data is mapped to a linked list of PUs. Flash file systems must manage mapping from logical sectors (LS) to physical sectors (PS) and perform garbage collection. 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.

Figure 2 illustrates a mapping from logical sectors to physical sectors where 1 unit contains 4 sectors. 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 ($SAM_{PU1}[0] = 0$). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of PU1 ($SAM_{PU1}[1] = 1$). The user then updates the LS1 and LS0, which results in $SAM_{PU1}[1] = 2$ and $SAM_{PU1}[0] = 3$. Finally, the user adds LS2 of LU7, which adds a new physical unit PU4 to LU7/and yields $SAM_{PU4}[2] = 0$.

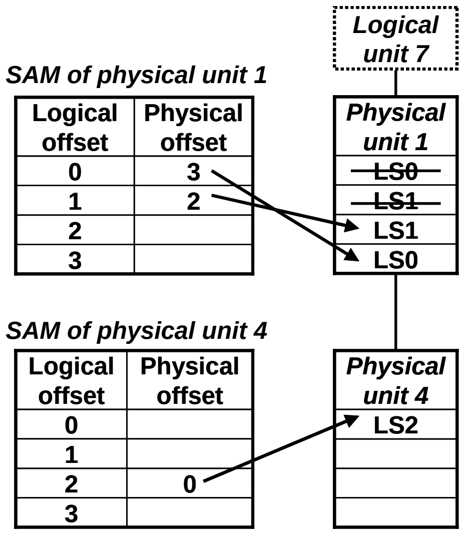


**Figure 2. Mapping from logical sectors to physical sectors**

## 2.3 Project Scope

Our team consists of two professors, one graduate student, and one senior engineer at Samsung Electronics. We worked on this verification project for six months. We spent the first three months reviewing USP design and code to become better familiarized with USP and OneNAND flash. Most parts of USP are written in C ($\sim$30000 lines) and a small portion of USP is written in ARM assembly language.

USP has a set of elaborated design documents that is a total of 259 pages long. It has a set of design documents on each layer of FTL and DPM as well as a software requirement specification (SRS) document. The SRS document specifies 13 functional requirements for USP. Each functional requirement specifies its own priority. There are three functional requirements that have "very high" priority - *concurrency handling, support prioritized read operation*, and *manage sectors*. In this paper, we concentrate on the third property, particularly on the correctness of *multi-sector read operation.* [1]

## 3 Overall Plan for Verification of Multi-sector Read Operation

This section describes an overview of the multi-sector read operation and the overall plan for verification of the operation.

[1]Analyses of the first two properties will be reported in a separate article.

### 3.1 Overview of 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 main procedure of this mechanism is implemented in a single function in STL, called MSR(). Due to non-trivial traversal of complex data structures for logical-to-physical sector mapping (see Section 2.2), MSR() (157 lines long) is highly complex, having 4-level nested loops. The outermost loop iterates until a specified number of sectors are read completely. The second outermost loop iterates over LUs of data and the third loop iterates over physical units mapped to a current LU. The innermost loop reads PS's in the current PU in the order of corresponding LS's in the current LU, i.e., reading the PS containing the first LS of the current LU first, then another PS for the second LS and so on. At this stage of operation, MSR figures out consecutive PS's in the current PU that contain consecutive data and read such multiple PS's as a whole for the performance improvement. Consequently, this function has a notorious bug history. For example, if MSR is implemented incorrectly, MSR may read data incorrectly in the case 3 of Figure 3 because data are not distributed over PS's in order.

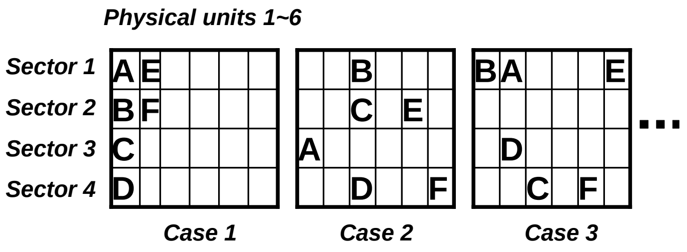


**Figure 3. Possible distributions of data "ABCDEF" to physical sectors**

According to the target requirement property, the MSR() should read multiple sectors correctly, i.e., the content of the read buffer should correspond to the original data written in the flash memory when the function finishes its reading.

We assume that each sector is 1 byte long and each unit has four sectors in our verification tasks. Also, we assume that data is a fixed string (e.g. "ABCDE" if we assume that data is 5 sectors long, and "ABCDEF" if we assume that data is 6 sectors long). In order to check iterations at every loop level, it is necessary to check flashes having data occupying at least 2 logical units and at least one logical unit should be distributed over more than one physical unit. A number of possible distribution cases for $l$ logical sectors and $n$ physical units where $5 \leq l \leq 8$ (i.e., 2 logical units are occupied by data) and $n \geq 2$ can be obtained by the following formula.

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

Table 1 shows the total number of possible cases for 6 logical sectors and 5,10,20, and 1000 physical units, respectively, according to the above formula. For example, if a flash has 1000 physical units with data occupying 6 logical sectors, there exist a total of $3.9 \times 10^{22}$ different distributions of the data.

| PUs | 5 | 10 | 20 | 1000 |
|---|---|---|---|---|
| Cases | $1.4 \times 10^6$ | $2.7 \times 10^8$ | $4.1 \times 10^{10}$ | $3.9 \times 10^{22}$ |

**Table 1. Total number of distribution cases**

### 3.2 A Plan for Verification Tasks

Our ultimate goal is to verify the correctness of MSR() when multiple applications execute MSR() concurrently. It is well known, however, that concurrency increases the complexity of verification exponentially due to the exponential number of interleaving scenarios. Thus, following the basic guideline that verification should start from a simple model and continue to enlarge the model gradually, our first goal is to verify the correctness of MSR() without concurrency, i.e., we assume that only one thread uses MSR() at a time. This article reports experimental results on this goal.

A test process consists of the following three steps.

1. Given data (say "ABCDEF"), the test harness distributes the data into physical sectors and updates SAM accordingly (see Figure 3).
2. MSR() reads the data through traversing physical units.
3. The test harness checks whether the read buffer contains the same original data after MSR() finishes.

In order to maximize the advantages of both model checking and testing, we have verified the multi-sector read operation through a series of verification tasks. First, we start the verification task with model checkers NuSMV and Spin for an exhaustive analysis of a small flash memory consisting of 5$\sim$10 physical units. Then, based on the formal model used in model checking, we setup a testbed for both exhaustive testing on a small size flash and randomized testing on a large flash memory consisting of 1000 physical units. The overall plan is as follows:

1. Model checking with NuSMV and Spin ($\sim$10 PUs)
2. Exhaustive testing on a small flash ($\sim$20 PUs)
3. Randomized testing on a large flash ($\sim$1000 PUs)

There have been various approaches to improve the scalability of model checkers, including the use of different encoding techniques and/or abstractions. For example, symbolic model checking such as NuSMV encodes state space with boolean variables, whereas explicit model checking such as Spin uses bit-vector for the compact representation of state space. Nevertheless, it is well-known that the pre-estimation of their relative scalability on a particular application domain is practically impossible, as a number of case studies have reported earlier [4, 6, 8, 14]. Therefore, we apply both techniques on a series of small-scale MSR() to benchmark their performance.

First, we used the NuSMV model checker for verification of MSR(). NuSMV was our first choice as a model checker, as it is well known for its capability of handling large state space using symbolic manipulation. Details on the verification task are presented in Section 4.

Second, we wrote a Spin model for MSR() for closer connection between the formal model and MSR() since Promela (an input specification language for Spin) has close similarity with C programming language. Details of this verification task are presented in Section 5.

Third, we abstracted MSR() into a simpler version (called S_MSR()) based on the Spin model. We then tested S_MSR() for all possible cases for a small flash containing 20 PUs. Finally, we generated test cases randomly for a large flash memory containing 1000 PUs and tested S_MSR() on these randomly selected test cases. Details of the third and fourth tasks are presented in Section 6. We performed all verification tasks using a workstation equipped with Xeon Woodcrest 5160 (dual core 3 GHz) and 32 gigabytes memory. The workstation runs 64 bit Fedora Linux 7 and uses Spin 4.3.0 and NuSMV 2.4.3.

## 4 Model Checking MSR() using NuSMV

Model checking techniques [7] are known to be effective for comprehensive verification of small-scale models, but also suffer from a scalability problem – the time and space complexities grow exponentially as the size of the search space grows. This scalability issue is intrinsic to model checking algorithms, which have time complexity bounded by the size of the search space that grows exponentially with the number of state variables.

### 4.1 Symbolic Model Checking

In symbolic model checking [15], a system model is represented with a compact boolean formula. For example, the simple code "if $(x < 2)$ then $x = x + 1$", where $x$ ranges over 0..3, may be encoded with two boolean variables, $x_1, x_0$, representing the current value $x$ and another two boolean variables, $x'_1, x'_0$, representing $x'$ which is the next value of $x$. The relationship among $x$ and $x'$ may be encoded as $x'_0 = \neg x_1 \wedge \neg x_0 \vee x_1 \wedge x_0$ and $x'_1 = \neg x_1 \wedge x_0 \vee x_1$.

This symbolic representation makes it possible to mathematically verify a system property through a series of symbolic computations; for example, if we wish to verify that the property "x is less than or equal to 2" holds for all execution sequences, then the property can be encoded as $\neg(x_1 \wedge x_0)$ and be verified through a satisfiability check of $\neg(x'_0 \wedge x'_1)$ that may require a series of boolean formula manipulations. NuSMV uses OBDDs (Ordered Binary Decision Diagrams) [3] for efficient manipulation of this boolean formula [15].

This approach can be quite effective when a large number of system variables have simple transition relationships that quickly stabilize; for an extreme example, consider a program with an integer input variable $x$ that ranges over 0 to 1000 and suppose the value of $x$ changes inside of the program with the assignment statement $x = x \; mod \; 2$. The symbolic encoding of this program may require 10 boolean variables, e.g., $x_9, x_8, x_7, \ldots x_1, x_0$ to represent the variable $x$, but their simple transition relation ( $x'_i = 0$ for all $i = 1..9$ and $x'_0 = x_0$) makes a boolean manipulation on $x$ quite trivial. On the other hand, exhaustive testing (or explicit model checking) may require 1001 test cases (or iterations) for checking all values of $x$.

## 4.2 Model Translation

We manually specified a NuSMV model for MSR() after reading corresponding design documents and C code. The first challenge in creating a NuSMV model for MSR() arises from the different modeling paradigms used in C and NuSMV; the NuSMV modeling language is dataflow-based, whereas both C and Promela are control-flow based languages. This is a major reason why Spin has been favored in program verification.

The first two columns in Figure 4 show an intuitive translation of a simple C code into Promela; the do-statement in Promela models the repetition where "::" indicates choices and "−>" indicates control-flow. On the other hand, the translation into NuSMV for the same C code (the last column of Figure 4) is considerably more involved since the NuSMV model mainly focuses on how data changes the value from its previous value. The notion of "flow of control" is artificially provided by introducing a boolean variable $after\_do$ indicating whether the current state is before or after performing the loop. Note that the next value of $i$ depends on the current values of $i$ and $after\_do$. Due to the differences in the employed modeling paradigms and the

| C code | Promela code | NuSMV code |
|---|---|---|
| sum=0; | sum=0; | init(sum):=0; |
| | i=0; | init(i):=0; |
| for(i=0; i<10; i++) | do | init(after_do):=0; |
| { | :: i<10->sum=sum+i;i=i+1; | next(sum):= case |
| sum +=i; | :: else->break; | i<10 :sum+i; |
| } | od; | 1 :i; |
| i=0; | i=0; | esac; |
| | | next(i):= case |
| | | i<10 : i+1; |
| | | after_do: 0; |
| | | 1 : i; |
| | | esac; |
| | | next(after_do):= (i=10); |

**Figure 4. Examples: model translation**

lack of variable indexing of arrays, the resulting NuSMV model for the MSR() operation reaches 1000 lines of code.

Other issues such as abstraction of data structures and handling of pointers are handled by removing auxiliary information that is not necessary for the verification purpose and by replacing pointers in a linked list as an array index.

## 4.3 Performance Analysis

We have performed a series of experiments in order to assess the efficiency of NuSMV for checking an essential property of MSR(),

$$INV : after\_MSR \rightarrow (\forall i.logical\_sectors[i] = buf[i]),$$

which means that the MSR() routine always maintain the data consistency between logical sectors and the read buffer.

We have restricted the data size of each logical sector to 3, i.e., the data vary over $\{0,1,2\}$, because it is too time consuming to perform the same experiment using full data domain. The resulting performance of NuSMV is quite poor; as shown in Table 2, it takes more than 32 hours for verifying the property when the number of logical sectors is increased to 7 and the number of physical units is 5. We also note that the verification time has worse scalability than that of the memory consumption.

| logical sectors | 5 | 6 | 7 |
|---|---|---|---|
| time (seconds) | 369.73 | $1,502.45$ | 32 hours |
| memory (Mega bytes) | 61.50 | 137.97 | 640.37 |

**Table 2. Time and space complexity of NuSMV model checking with 5 physical units**

The exponential growth of verification time is mainly due to the dynamic reordering of BDD variables to keep the symbolic representation of the state space as compact as possible. OBDD representations for a boolean formula can be quite different in terms of the number of nodes representing the formula. Since finding an optimal BDD variable ordering is an intractable problem (an NP-complete problem [2]), NuSMV periodically attempts to improve orderings by moving each variable through the ordering to find its best location using the sifting algorithm [16]. This ordering process is known to be effective to reduce the state-space, but is time-consuming, as is clearly seen from our experiments.

## 5 Model Checking MSR() using Spin

Due to modeling difficulty and performance limitations of NuSMV, as described in Section 4.2 and Section 4.3, we used Spin as an alternative model checker. Promela is similar to C and as such it is easier to make a formal Promela model from MSR() compared to NuSMV. Furthermore, we can use Modex [12], which is a general purpose semi-automatic translation tool from C to Promela, to create a Promela model with embedded C code from MSR().

### 5.1 Spin Model Checker

Spin is an explicit model checker. In explicit model checking, each system state (called a state vector) consists of variables of a verification model and all system states are stored in a huge hash table explicitly. Thus, it is commonly perceived that Spin cannot handle large state spaces compared to a symbolic model checker such as NuSMV. However, due to its simple state space generation and traversal methods, Spin outperforms symbolic model checkers on some domains. Another advantage of an explicit model checker is that data structures in a model do not incur extra overhead other than increase of the size of state vector because data structures are stored into the state vector as they are, not through complex BDD encoding.

One benefit of using Spin is that it supports the inclusion of embedded C which alleviate the limitations of Promela and provides a means for data abstraction. From a given Promela model, Spin generates a verifier written in C, and compiles/executes the verifier to perform verification. Spin versions 4.0 and later support the inclusion of embedded C code into Promela models through a `c_code{...}` keyword. The main purpose of this extension is to support semi-automatic model extraction from C code [11]. The contents of the embedded C code fragments are blindly copied through from the text of the model into the code of the verifier that Spin generates. Thus, variables in embedded C code are not stored into a state vector and do not contribute to verification. A user can add variables in embedded C code to a state vector using a `c_track` statement.

```
do
:: if
   :: sect < 4 * MAX_PUN ->
      if
      :: used['A']==false ->
         PU[sect/4].sector[sect%4]='A';
         used['A']=true;
         SAM[sect/4].offset[0]=sect%4;
      :: used['B']==false ->
      ...
      :: skip;
      fi;sect=sect+1;
   :: else-> break;
   fi;
od
```

**Figure 5. A pseudo environment model**

Also, a user can track the values of a variable in embedded C code without adding the variables to the state vector, and thus the search traversal of Spin can work correctly. With these embedded C extensions, a user can apply data abstractions in a verification model to increase the scalability of model checking.

### 5.2 Model Translation

In a Promela model, an environment (i.e., logical sectors, physical sectors and SAM) is created in a way similar to the pseudo code in Figure 5. Note that all possible distributions of data into physical sectors are generated *exhaustively* through non-deterministic guarded commands.

For MSR(), Modex [12] creates a single Promela process whose control structures are translated from MSR(). All C control structures such as `if()` and `while()` are translated into corresponding Promela statements. Other C statements are inserted into an embedded C code of the Promela model starting with a keyword `c_code{...}`. Therefore, the Promela model generated by Modex has the same 4-level nested loops as MSR(). Then, we manually modified the generated Promela model to make embedded C code work correctly under the Spin verification environment.

For data structures used by MSR(), we replace a linked list of physical units and SAMs in MSR() with an array of physical units and SAMs, since Spin cannot directly track a dynamic linked list in embedded C code. Thus, we modified the data structure of physical units and SAMs and corresponding statements to use the modified data structure. This modification was made through a manual translation script, which is given as an additional input to Modex. The translation script for Modex is 63 lines long.

The requirement property is checked by

`assert(logical_sectors[0] == buf[0] && ... )` statement located at the end of MSR(). The translated Promela model is 250 lines long, including embedded C code.

### 5.3 Performance Analysis

We have performed a series of experiments with different lengths of data as well as different numbers of physical units. These experiments were performed without lossy compression such as bitstate hashing [10]. Figure 6 illustrates performance data for checking the requirement property. In all of the experiments, Spin shows that the requirement property is satisfied.

Spin verified a flash containing 10 physical units and 6 logical sectors in 50 minutes, consuming 20 gigabytes of memory. Considering that Spin exhaustively analyzed all $2.7 \times 10^8$ cases (see Table 1), at a rate of 11 microseconds per case, the verification performance of Spin is not obstructive. As can be seen in Figure 6, the memory consumption and verification time increases exponentially in relation to the number of physical units. The bottleneck in this verification task is its memory consumption. Spin handles states explicitly, and thus the exponentially increasing number of possible distribution cases accordingly causes an exponential increase of memory.

Compared to NuSMV, however, Spin shows significantly better performance for verification tasks of this type. For example, for a test case of 7 logical sectors and 5 physical units, Spin takes 62 seconds with 797 megabytes, 268 megabytes of which is consumed for a hash table setup, while NuSMV takes more than 32 hours with 640 megabytes. [2] This performance advantage is mainly achieved by the fact that Spin stores state space without BDD encoding, which was the performance bottleneck of NuSMV in verification tasks of this type (see Section 4.3).

## 6 Testing MSR()

Based on the Promela model for MSR(), we developed a testing environment as well as an abstracted version of MSR(), called S_MSR(). Test input to S_MSR() is a configuration of a flash memory consisting of PUs and corresponding SAMs as depicted in Figure 3 and Figure 2. Given a configuration, we checked whether or not S_MSR() read the content of a flash memory correctly by comparing the read buffer with the content of the flash memory. These test inputs were generated by the similar algorithm to the environment model in the Promela model (see Figure 5) which generates all possible configurations of a flash memory exhaustively. [3] We tested S_MSR() instead of MSR() for the following reasons.

First, we could not compile and execute MSR() directly, because, for security reasons, Samsung did not provide us a testbed that included OneNAND device and low-level codes necessary for compilation. This type of situation occurs frequently in software development projects, particularly when the target software has yet to be completely developed, and other parts necessary for compiling and executing the target code are not available.

Second, MSR() calls numerous sub-functions to access OneNAND devices. Figure 7 shows a partial dependency graph of MSR(). A full dependency graph of MSR() has total 56 distinct sub-functions (we do not have code for 8 out of 56 functions) in 9-level depths. Since our focus is to analyze MSR(), not its underlying functions, it is desirable to use abstracted versions of such sub-functions, similar to those used in the Promela model.

For example, BML_MRead() (located at the center of Figure 7) reads consecutive physical sectors that contain consecutive data in one physical unit. BML_MRead() performs various other detailed tasks such as setting device registers and handling a read error of a physical sector among consecutive ones, etc. The Promela model as well as the NuSMV model do not contain such details, but contain essential sector traversing operations only. We removed such non-essential details, but the core control structure of MSR() is maintained.

The reuse of abstract formal models reduces the testbed setup time since we do not have to re-analyze MSR() to decide what to include for testing. In this way, S_MSR() contains only the essential core of MSR() and all auxiliary environments are omitted. Furthermore, because S_MSR() contains less code and abstract operations, testing time is reduced. Consequently, we could concentrate on MSR() without having to consider details of those sub-functions.

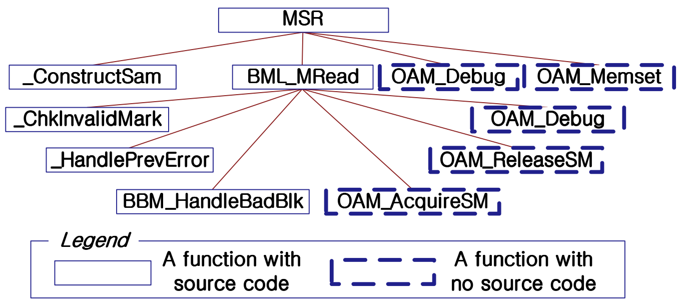
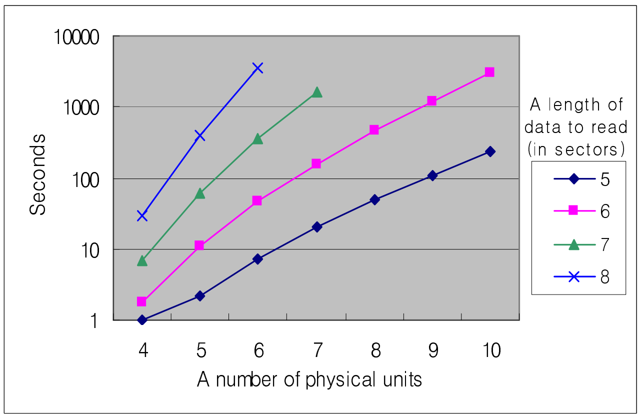


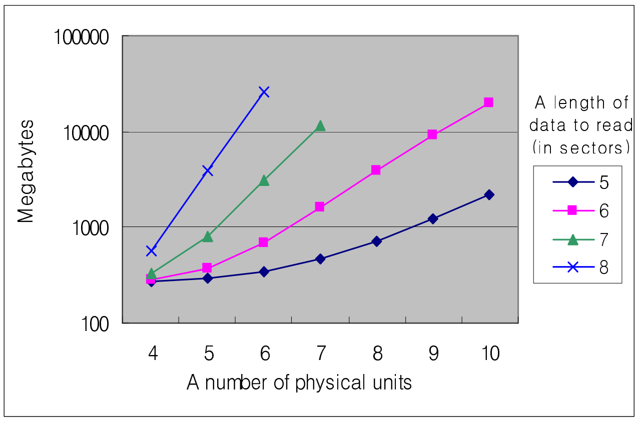
**Figure 7. A partial dependency graph of MSR()**

[2]Note that the data size of each logical sector in NuSMV experiments was 3 while Spin used a full data size, i.e., 256.

[3]Although the environment model in the Promela model uses non-determinism, we can implement the same effect using iterations in C.



**a) Time complexity**



**b) Space complexity**

**Figure 6. Time and space complexity of Spin model checking**

Table 3 shows the time taken to perform exhaustive testings with 6 sectors long data. For all of these exhaustive testings, no violation of the property was detected. Exhaustive testing could analyze a larger flash than Spin could analyze since memory was not a bottleneck. Also, the performance of exhaustive testing is roughly 50 times faster compared to Spin for the case of 10 PUs, mainly because Spin bookkeeps all states and transitions in memory. The use of specially designed testing environment for the given property also plays an important role for the high performance testing; Spin is a general purpose model checker which is designed to handle temporal logic properties in general, and, thus, cannot win over a specialized tool. However, if a violation occurs, Spin can provide step-by-step counter examples, which are invaluable in helping to find a bug, at the cost of slower execution.

| # of PUs | 5 | 10 | 15 | 20 |
|---|---|---|---|---|
| Time (seconds) | 1 | 62 | 1496 | 14277 |

**Table 3. Time complexity for exhaustive testing**

Finally, we have performed randomized testing on $10^{11}$ randomly chosen cases over 1000 physical units: this takes 8 hours 20 minutes. For this large size flash, we cannot perform exhaustive testing since even $10^{11}$ chosen cases covered only $\frac{1}{3.9 \times 10^{11}}$ among all possible cases. Considering that a 1 gigabyte flash memory has more than a half million physical units (= $2^{19}$ units $\times$ 4 sector/unit $\times$ 512 byte/sector), randomized testings cannot provide sufficient coverage ever.

## 7 Lessons Learned

This section presents lessons learned from this project.

### 7.1 Model Checking as Pre-testing

Through this project, we found that an integrated analysis framework using both model checking and testing is effective and also efficient. In particular, we found that a formal model used for model checking can provide a good base for setting up a testbed and abstraction in the target program under testing. This is because a formal model contains an explicit environment model that is equivalent to the testing environment and the target program is abstractly modeled to minimize unnecessary details and the computational overhead of the verification process. Thus, efforts to make a formal model can enhance the reliability of the target program through model checking and, at the same time, reduce testing costs. Therefore, we believe that model checking can be used as an effective pre-testing task.

Our approach is opposite to that of [9] which performs randomized testing as a prelude to formal verification in order to establish a quicker path to finding many bugs at the early stage of development. Our case is different since we analyzed a stable code that had previously been tested extensively by Samsung. Thus, our aim was to detect subtle hidden bugs, if any, not to make a quicker path to find many initial bugs. However, the two studies similarly imply that both model checking and testing are necessary for ensuring high reliability of the target program and that a well-designed abstract model can provide much more confidence in correctness than can random exploration.

### 7.2 Benefits of C-like Modeling Language

In this project, NuSMV and Spin show major differences in both modeling effort and verification performance. Promela uses a C-like syntax and semantics, and thus translation from C to Promela is not difficult. A close relationship between the target program and the formal model is important, because there is always a possibility of mistranslation. If a mapping between the target program and the formal model is not clear, it will be difficult to check the soundness of the formal model and thus an incorrect verification result could be obtained.

Another benefit of C-like modeling language is that a formal model can be conveniently used to build a testbed as well as an abstracted version of the target program under testing. Finally, the embedded C feature of Promela can be used as a powerful data abstraction technique (see Section 5.2). Therefore, for verification of C programs, especially those using a large data structure, Spin has a clear advantage over NuSMV.

### 7.3 Inadequacy of Randomized Testing

We could observe that randomized testing alone does *not* provide enough confidence on the correctness of multi-sector read operation since even hundred billions of testings for 8 hours covers only $\frac{1}{3.9\times10^{11}}$ of possible cases. Considering that Samsung relied on randomized testings only, it is no wonder that multi-sector read operation has a notorious history of bugs. Therefore, we need to perform exhaustive analysis on a small model through model checking and randomized testing on a large model together.

The scalability issue brought about by a large scale data structure can be addressed using an alternative approach, *theorem proving*, when it is required. Theorem proving is based on modeling a system in mathematical logic and performing logical computations (called a decision procedure), such as mathematical induction, skolemization, and resolution, to verify a system property. This approach has a clear benefit: the complexity of verification depends on the number of logical rules to be resolved, which is independent of the size of data. In other words, the time and space complexity for verifying MSR() using theorem proving will be independent of the number of physical units.

### 7.4 Why Use Model Checking

Samsung had performed the majority of testing for OneNAND software randomly at a file system level. Testing through a file system does not provide direct control over logical-to-physical mapping, and thus opportunities to detect bugs are missed. Also, even a huge number of randomized testings does not provide sufficient coverage for detecting bugs, as can be seen in the present experiments. Although results of these exhaustive analyses confirm the correctness of the multi-sector read operation for a small flash only, exhaustive exploration through model checking and testing can provide high confidence in the correctness of the multi-sector read function. Therefore, we believe that an exhaustive analysis of sector mappings for a small number of units through model checking is needed. This should then be followed by the establishment of a testbed using a formal model and randomized testing on a large number of units to simplify the testbed setup and increase debugging efficiency.

## 8 Conclusion and Future Work

In this project, we have successfully verified the correctness of a multi-sector read operation of Samsung OneNAND$^{\mathrm{TM}}$ flash device driver with exhaustive exploration of model checking and scalability of testing. Through the project, we found that exhaustive analysis on a small flash by model checking can be helpful for the verification task and observed that a formal model can be effectively used to set up a testbed and test an abstract version of target software. We will extend our analyses of MSR() by augmenting the current Spin model to allow multiple threads to execute MSR() concurrently and verify the concurrency properties such as absence of deadlock/livelock as well as the correctness property we have verified.

Samsung highly valued the verification results and, as a next project, we plan to analyze a flash file system to check data consistency at the events of random power-off. This issue is very important because flash memory is widely used in mobile devices, which frequently experience unexpected power-off in daily usage. For this topic, we will also apply runtime verification techniques similar to [13] to analyze subtle details of a C program which are hard to model.

## References

[1] Samsung OneNAND fusion memory. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html`.

[2] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45(9), September 1996.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] Y. Choi. From NuSMV to SPIN: Experiences with model checking flight guidance systems. *Formal Methods in System Design*, pages 199–216, June 2007.

[5] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceeding of International Conference on Computer-Aided Verification*, 2002.

[6] Y. Dong, X. Du, G. J. Holzmann, and S.A. Smolka. Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking. *International Journal on Software Tools for Technology Transfer*, (4), 2003.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, January 2000.

[8] C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In *SPIN Workshop*, 2002.

[9] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, May 2007.

[10] G. J. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.

[11] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Spin Workshop*, 2004.

[12] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4), 2002.

[13] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 2004.

[14] G. Luettgen and V. Carreno. Analyzing mode confusion via model checking. In *SPIN workshop*, 1999.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[16] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design(ICCAD)*, November 1993.