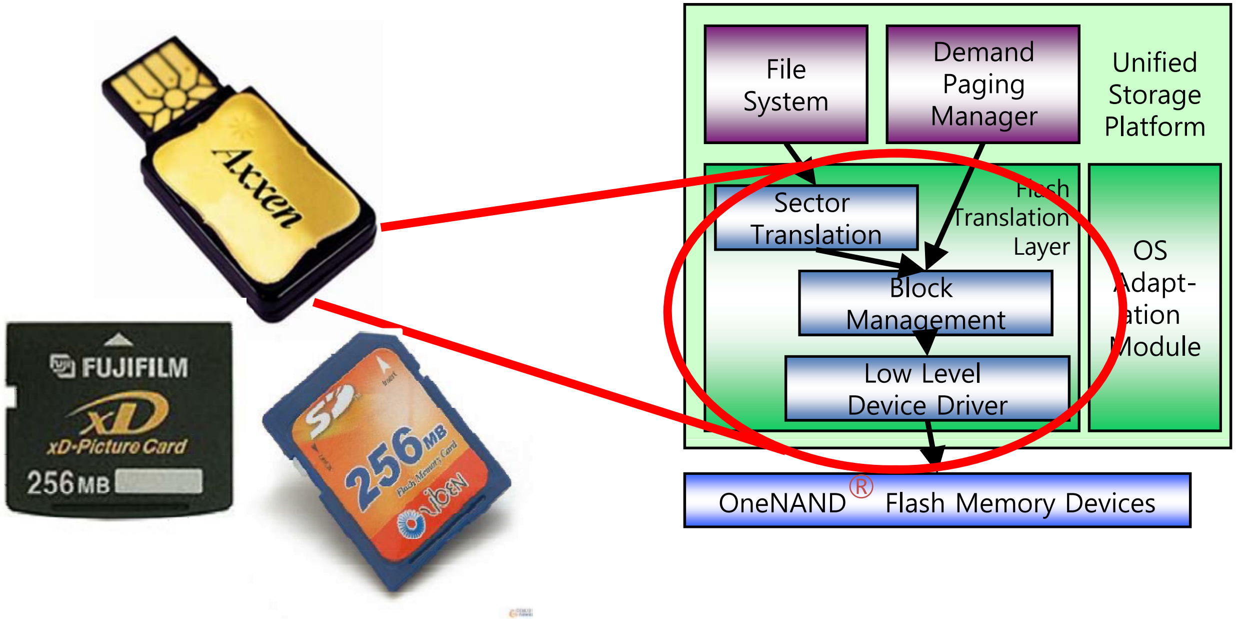
Unit Testing of Flash Memory Device Driver through a SAT-based Model Checker

Moonzoo Kim and Yunho Kim
Provable Software Lab, CS Dept, KAIST

Hotae Kim
Samsung Electronics, South Korea

Summary of the Talk

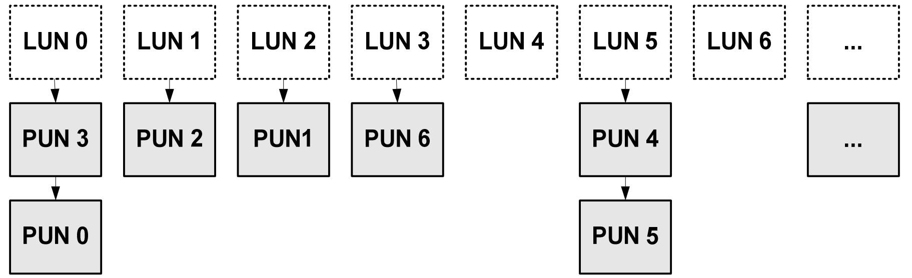


- In 2007, Samsung requested to debug the device driver for the OneNAND™ flash memory
- We reviewed the requirement specifications, the design documents, and C code to identify code-level properties to check.
- Then, we applied CBMC (C Bounded Model Checker) to check the properties
  - Found several bugs
  - Provided high confidence in multi-sector read operation through exhaustive exploration

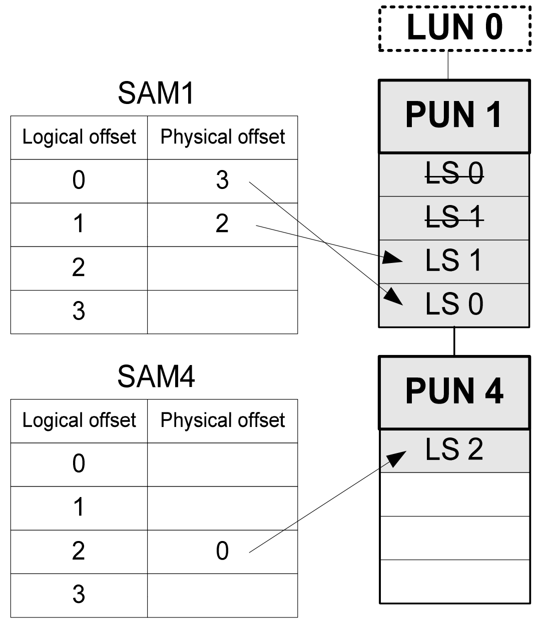
Overview

- Background
  - Logical-to-physical sector translation
  - Overview of the Unified Storage Platform (USP)
  - SAT-based model checking technique
- Identification of properties to check
  - High-level requirements
  - Code-level properties
- Unit analysis result through CBMC
  - Prioritized read operation (PRO)@ Demand Paging Manager (DPM)
  - Semaphore matching (SM)@ Block Management Layer (BML)
  - Semaphore exception handling (SEH)@ STL~BML
  - Multi-sector read operation (MSR) @ Sector Translation Layer (STL)
- Lessons learned and conclusion

Logical to Physical Sector Mapping



1:N mapping from a LUN to PUNs



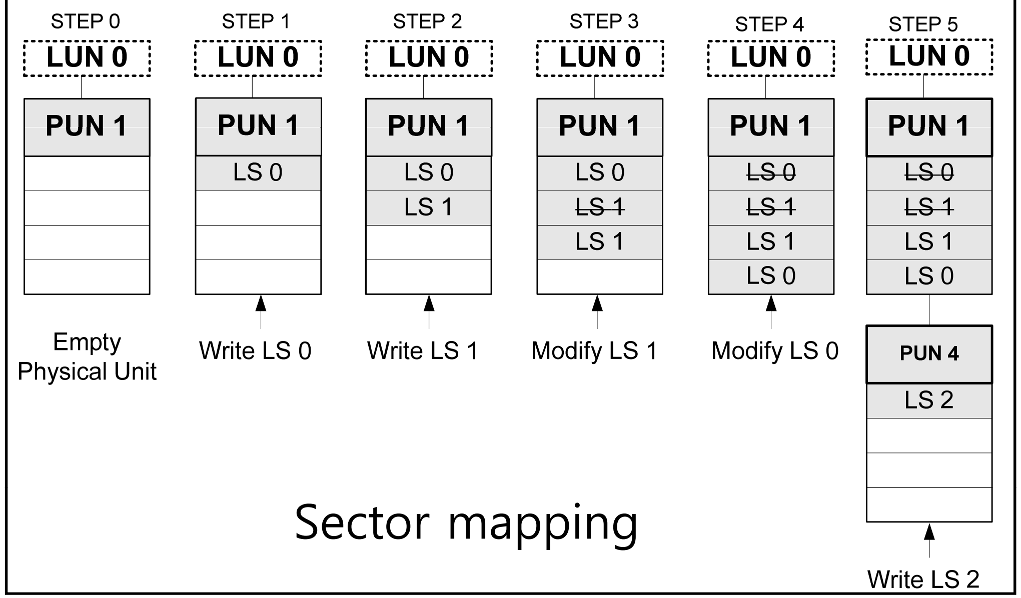
SAM1

Logical offset	Physical offset
0	3
1	2
2	
3	

SAM4

Logical offset	Physical offset
0	
1	
2	0
3	

Sector Allocation Map (SAM)

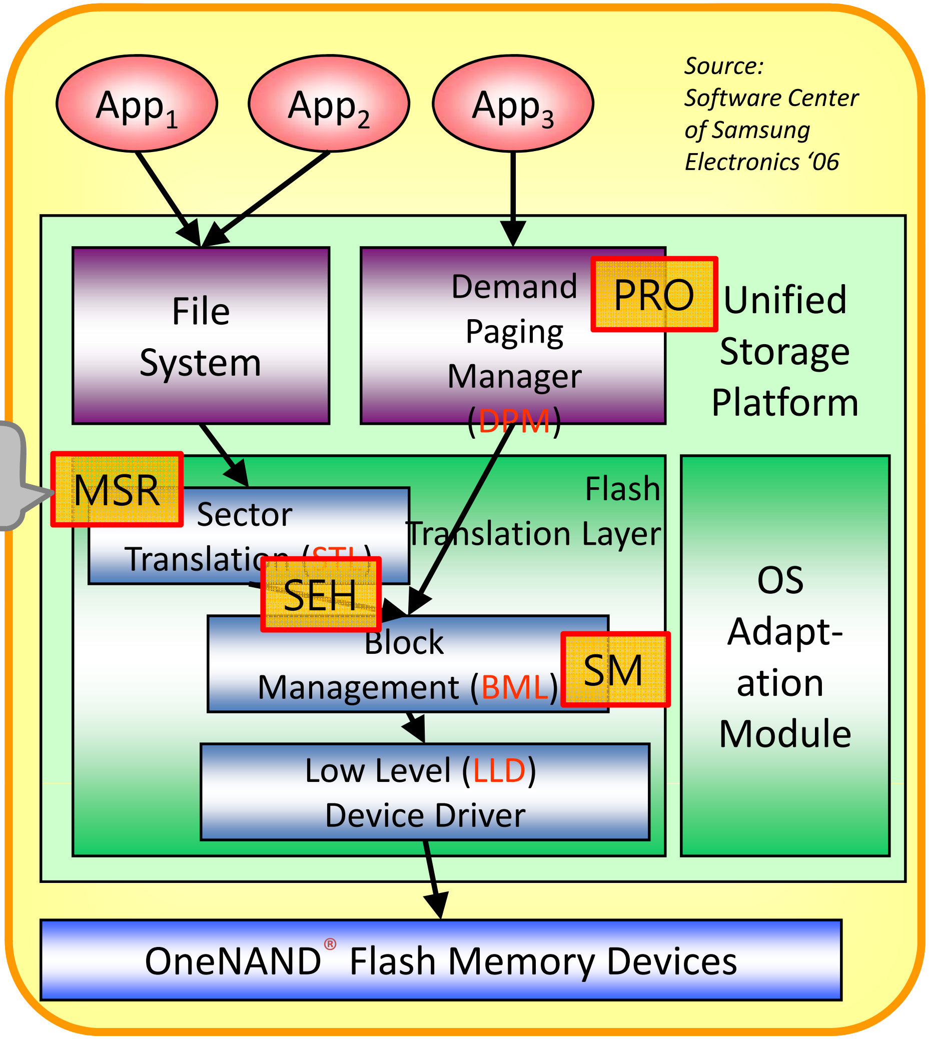


Sector mapping

- In flash memory, logical data are distributed over physical sectors.

Overview of the OneNAND® Flash Memory

- Characteristics of OneNAND® flash
  - Each memory cell can be written limited number of times only
    - Logical-to-physical sector mapping
    - Bad block management
    - Wear-leveling
  - XIP by emulating NOR interface through demand-paging scher
    - Multiple processes access the concurrently
    - Urgent read operation should have a higher priority
    - Synchronization among processes is crucial
  - Performance enhancement
    - Multi-sector read/write
    - Asynchronous operations
    - Deferred operation result check



C Bounded Model Checker (CBMC)

- Handles function calls using inlining
- Unwinds the loops a fixed number of times (bounded MC)
  - A user has to know a upper bound of each loop
    - Loops often have clear upper bounds
    - We can still get debugging result without upper bounds
- Specifies constraints to describe an environment of the target program, which can model non-deterministic user inputs, or multiple scenarios
  - Ex. __CPROVER assume(0<=nDev && nDev<=7)
  - Ex.__CPROVER_assume(SHDC.nPhySctsPerUnit == SHPC.nBlksPerUnit * SHVC.nPgsPerBlk * SHVC.nSctsPerPg)
- Checks properties by assertions

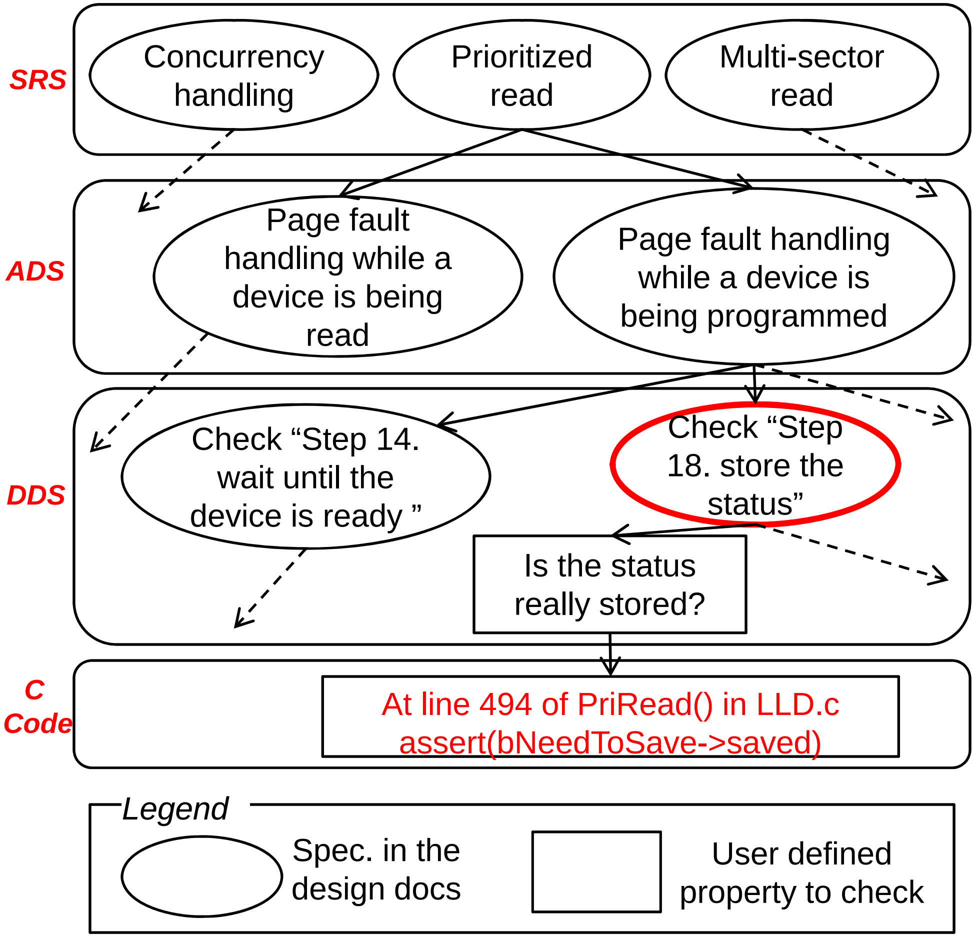
Project Overview

- The goal of the project
  - To check whether USP conforms to the given high-level requirements
    - we needed to identify the code-level properties to check from the given high-level requirements
- A top-down approach to identify the code level properties from high-level requirements
  - USP has a set of elaborated design documents
    - Software requirement specification (SRS)
    - Architecture design specification (ADS)
    - Detailed design specification (DDS)
      - DPM, STL, BML, and LLD

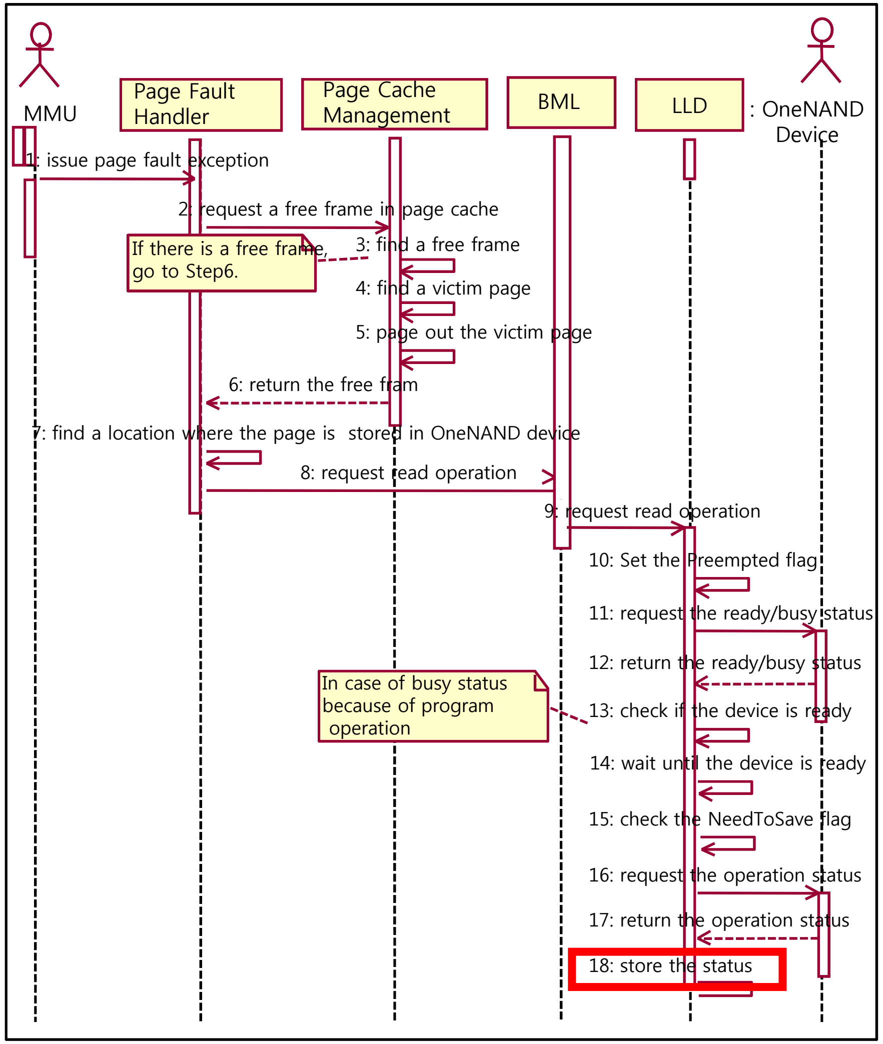
Three High-level Requirements in SRS

- SRS specifies 13 functional requirements, 3 of which have "very high" priorities
  - Support prioritized read operation
    - To minimize the fault latency, USP should serve a read request from DPM prior to generic requests from a file system.
    - This prioritized read request can preempt a generic I/O operation and the preempted operation can be resumed later.
  - Concurrency handling
    - BML and LLD should avoid a race condition or deadlock through synchronization mechanisms such as semaphores and locks.
  - Manage sectors
    - STL provides logical-to-physical mapping, i.e. multiple logical sectors written over the distributed physical sectors should be read back correctly.

Top-down Approach to Identify Code-level Property


SRS
Concurrency handling
Prioritized read
Multi-sector read
ADS
Page fault handling while a device is being read
Page fault handling while a device is being programmed
DDS
Check "Step 14. wait until the device is ready "
Check "Step 18. store the status"
Is the status really stored?
C Code
At line 494 of PriRead() in LLD.c
assert(bNeedToSave->saved)
Legend
Spec. in the design docs
User defined property to check

- Total 43 code-level properties are identified


MMU
Page Fault Handler
Page Cache Management
BML
LLD
: OneNAND Device
1: issue page fault exception
2: request a free frame in page cache
If there is a free frame, go to Step6.
3: find a free frame
4: find a victim page
5: page out the victim page
6: return the free fram
7: find a location where the page is stored in OneNAND device
8: request read operation
9: request read operation
10: Set the Preempted flag
11: request the ready/busy status
12: return the ready/busy status
In case of busy status because of program operation
13: check if the device is ready
14: wait until the device is ready
15: check the NeedToSave flag
16: request the operation status
17: return the operation status
18: store the status

A sequence diagram of page fault handling while a device is being programmed in LLD DDS

Results of Unit Testings

- Prioritized read operation
  - Detected a bug of not saving the status of suspended erase operation
- Concurrency handling
  - Confirmed that the BML semaphore was used correctly
  - Detected a bug of ignoring BML semaphore exceptions
- Multi-sector read operation (MSR)
  - Provided high assurance on the correctness of MSR, since no violation was detected even after exhaustive analysis (at least with a small number of physical units(~10))

A Bug in `PriRead()`

```
374: VOID PriRead(Read(UINT32 nDev, UINT32 nPbn, UINT32 nPgOffset) {
…
416:    if ((bEraseCmd==FALSE32) && (pstInfo->bNeedToSave==TRUE32))  {
417:        pstInfo->nSavedStatus = GET_ONLD_CTRL_STAT(pstReg, ALL_STATE);
418:        pstInfo->bNeedToSave  = FALSE32;
419:        saved=1;  // added for verification purpose   }
…
424:    assert(!(pstInfo->bNeedToSave) || saved);
```

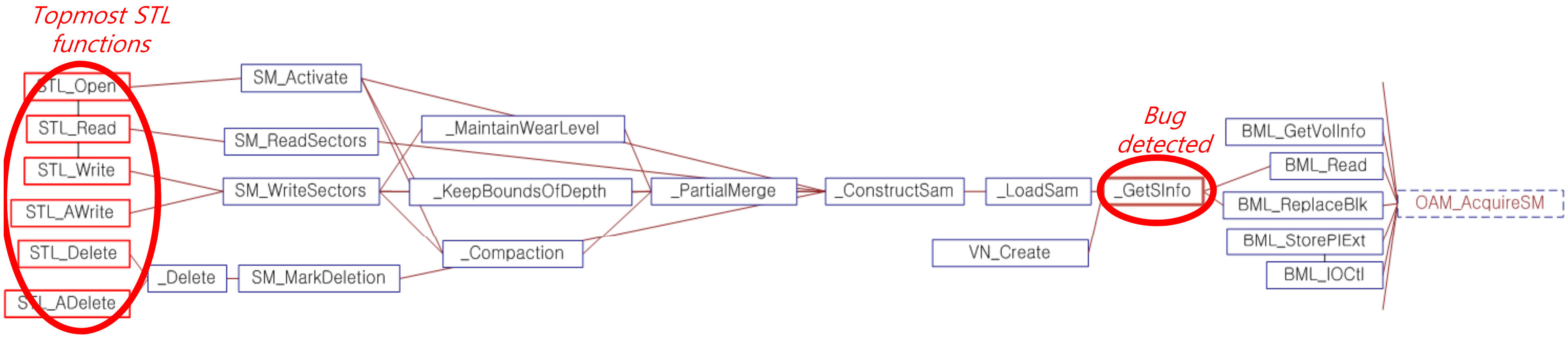
- We added a flag `saved` to denote whether the status of the preempted operation is saved
- CBMC detected the given assertion was violated when an erase operation was preempted
  - It takes 8 seconds and 325 Mb on the 3Ghz Xeon machine
  - CBMC 2.6 with MiniSAT 1.1.4

```
01:…
02:State 14 file LLD.c line 408 function PriRead thread 0
03: LLD::PriRead::1::bEraseCmd=1
04:State 15 file LLD.c line 412 function PriRead thread 0
05: LLD::PriRead::1::1::2::nWaitingTimeOut=…
06:State 17 file LLD.c line 412 function PriRead thread 0
07: LLD::PriRead::1::1::2::nWaitingTimeOut=…
08:…
09:Violated property:
10: file LLD.c line 424 function PriRead
11: assertion !(_Bool)pstInfo->bNeedToSave || (_Bool)saved
12:VERIFICATION FAILED
```

BML Semaphore Usage

- The standard requirements for a binary semaphore
  - Semaphore acquire should be followed by a semaphore release
  - Every function should return with a semaphore released
    - unless the semaphore operation creates an exception error.
- There exist 14 BML functions that use the BML semaphore.
  - We inserted an `smp` to indicate the status of the semaphore
  - and simple codes to decrease/increase `smp` at the corresponding semaphore operation.
- CBMC concluded that all 14 BML functions satisfied the above two properties.
  - Consumes 10 seconds and 300 megabytes of memory on average to analyze each BML function

BML Semaphore Exception Handling (1/2)

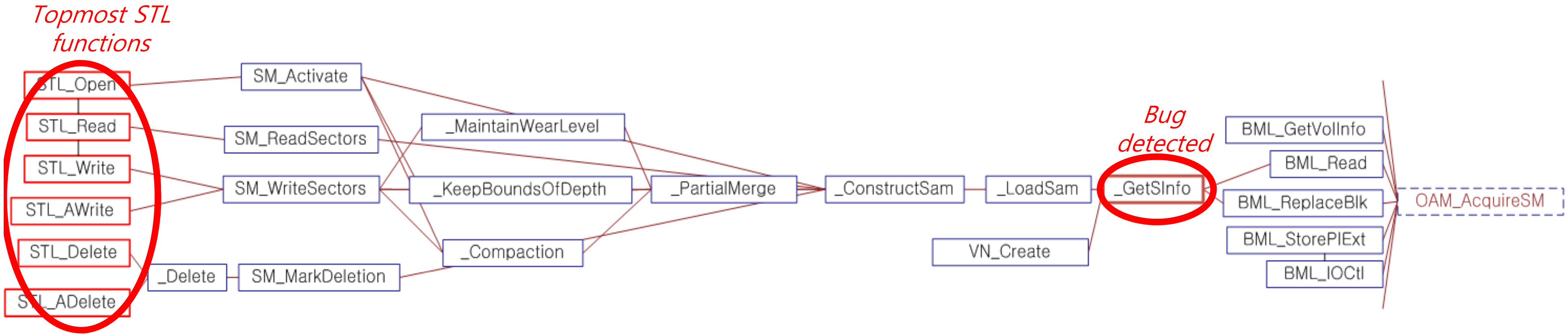


Topmost STL functions

Bug detected

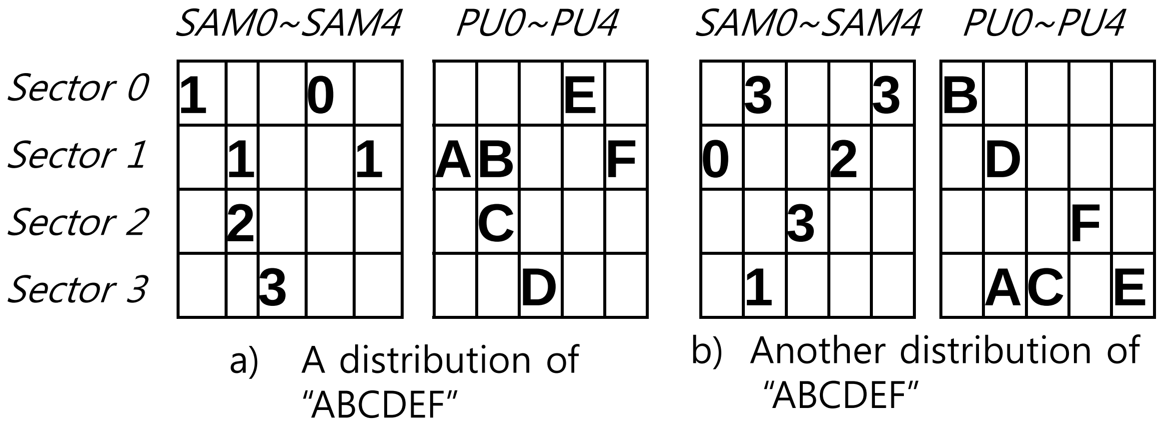
- The BML semaphore operation might cause an exception depending on the hardware status.
- Once such BML semaphore exception occurs, that exception should be propagated to the topmost STL functions to reset the file system
  - We checked this property by the following assert statement inserted before the return statement of the topmost STL functions:
  - assert(!(SMerr==1)||nErr==STL CRITICAL ERR)

BML Semaphore Exception Handling (2/2)



- CBMC analyzed a call graph of each of the topmost STL functions and detected that BML semaphore exception might not propagate due to bug at _GetSInfo()
- The bug was detected when loop bound was set 2 with ignoring loop unwinding assertion.
  - Memory overflow occurred with the loop bound 3
- For STL_Write(), this verification task consumed 616 megabytes of memory in 97 seconds
  - Each call sequence is around 1000 lines long on average.

Multi-sector Read Operation (MSR)



	SAM0~SAM4					PU0~PU4				
Sector 0	1			0					E	
Sector 1		1			1	A	B			F
Sector 2		2					C			
Sector 3			3					D		

a) A distribution of "ABCDEF"

	SAM0~SAM4					PU0~PU4				
Sector 0		3			3	B				
Sector 1	0			2			D			
Sector 2			3						F	
Sector 3		1					A	C		E

b) Another distribution of "ABCDEF"

- MSR reads adjacent multiple physical sectors once in order to improve read speed
  - MSR is 157 lines long, but highly complex due to its 4 level loops
- We built a small test environment for MSR
  - The test environment contains only upto 10 physical units
  - The test environment should follow constraints, which are described by _CPROVER_assume(Boolean exp) statement
    - SAM tables and PUs should correspond each other
    - For each logical sector, at least one physical sector that has the same value exists

Environment Model

- Environment model creation
  - The environment of MSR (i.e., PUs and SAMs configurations) can be described by invariant rules. Some of them are
    1. One PU is mapped to at most one LU
    2. *Valid correspondence between SAMs and PUs:*

       If the *i* th LS is written in the *k* th sector of the *j* th PU, then the *i* th offset of the *j* th SAM is valid and indicates the k'th PS ,

       Ex> 3rd LS ('C') is in the 3rd sector of the 2nd PU, then SAM1[2] ==2

       i=2 k=2 j=1
    3. *For one LS, there exists only one PS that contains the value of the LS:*

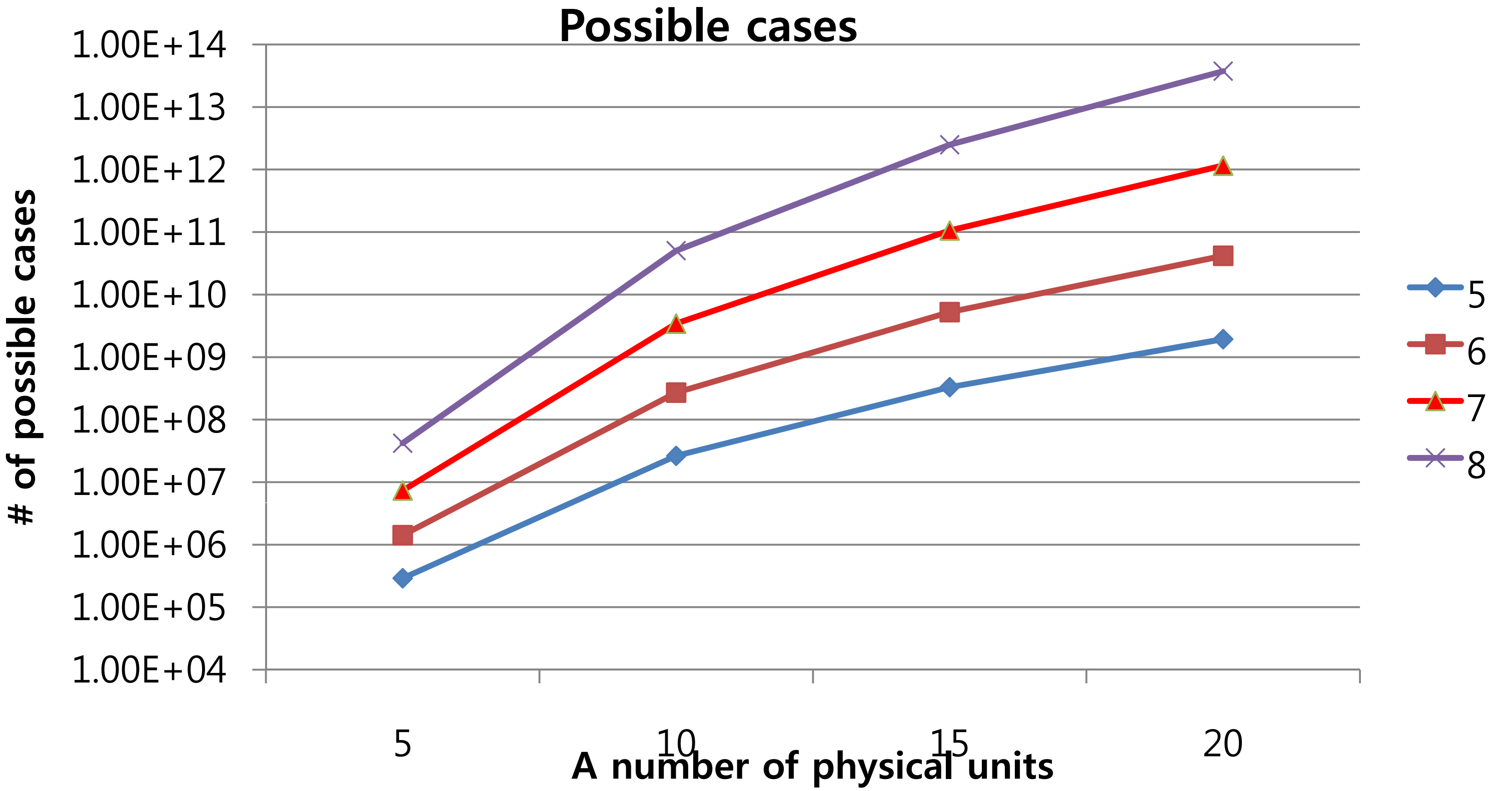
       The PS number of the *i* th LS must be written in only one of the (*i* mod *4*) th offsets of the SAM tables for the PUs mapped to the corresponding LU.

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

	SAM0~SAM4					*PU0~PU4*				
Sector 0	1			0					E	
Sector 1		1			1	A	B			F
Sector 2		2					C			
Sector 3			3					D		

Exponential Increase of Distribution Cases

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



MSR Model Checking Results

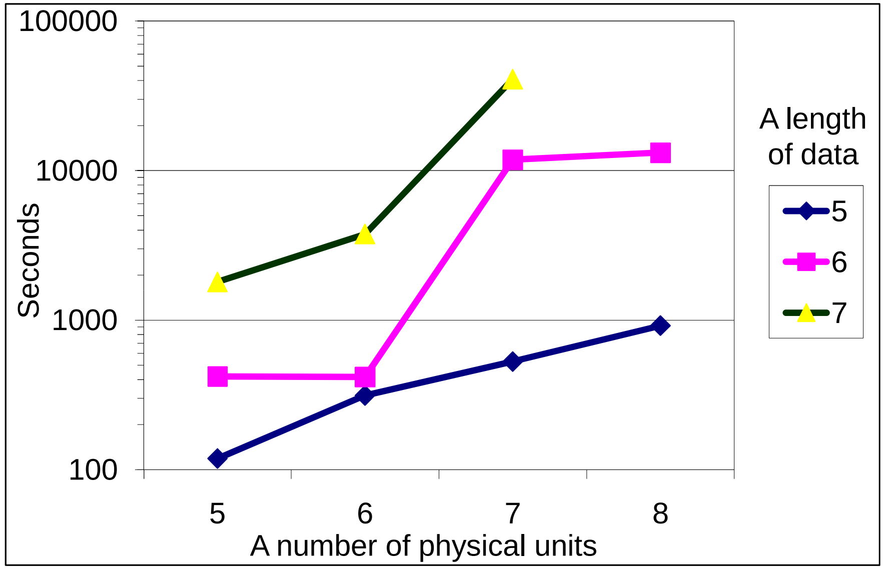
- Verification of MSR by using NuSMV, Spin, and CBMC
  - NuSMV: BDD-based symbolic model checker
  - Spin: Explicit model checker
  - CBMC: C-bounded model checker
- The requirement property is to check
  - after_MSR -> (∀ i. logical_sectors[i] == buf[i])
- We compared these three model checkers empirically

Excerpts of the SMV Model

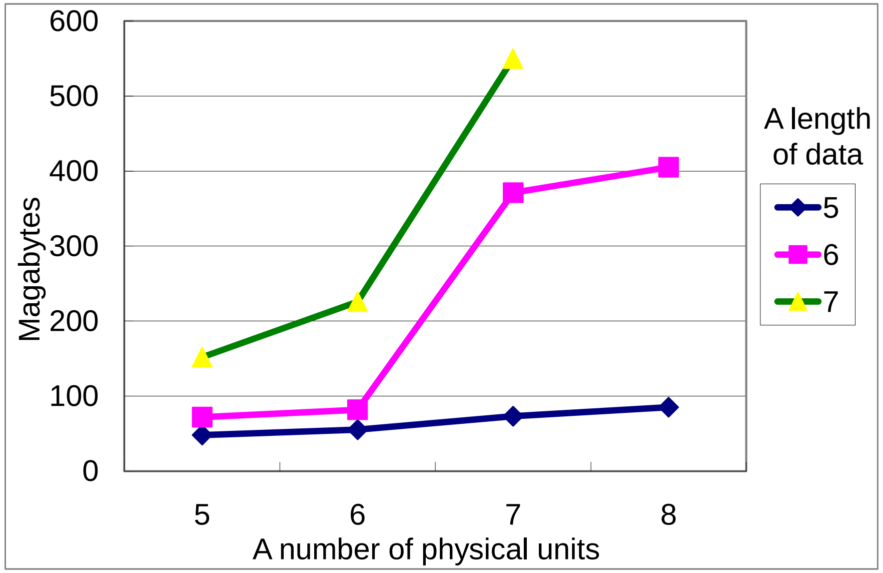
```
MODULE main

-- Variable declaration
VAR
   SAM      : array 0..4 of sam_type;
   PU       : array 0..4 of PU_type;
   buf      : array 0..4 of 0..5;
   nScts    : 0..5;

-- SPEC
INVARSPEC (after_first_do ->
PU[0].sect[0]=1 &
PU[0].sect[1]=2 &
PU[0].sect[2]=3 &
PU[0].sect[3]=4 &
PU[3].sect[0]=5)
```

```
init(buf[0]):=0;
--  if( pBuf==0 && 0 < nScts )
--        buf[0]= PU[PU_id].sect[nFirstOffset]
next(buf[0]):
  case after_fourth_do :
       case pBuf = 0 & 0 < nScts:   -- i=0
      case
          PU_id=0 & nFirstOffset=0: PU[0].sect[0];
          PU_id=0 & nFirstOffset=1: PU[0].sect[1];
          PU_id=0 & nFirstOffset=2: PU[0].sect[2];
          PU_id=0 & nFirstOffset=3: PU[0].sect[3];
          ...
          PU_id=4 & nFirstOffset=3 : PU[4].sect[3];
      esac;
   esac;

init(buf[1]):=0;
next(buf[1]):= ...
```

Verification Performance of NuSMV



(a) Time consumption



(b) Memory consumption

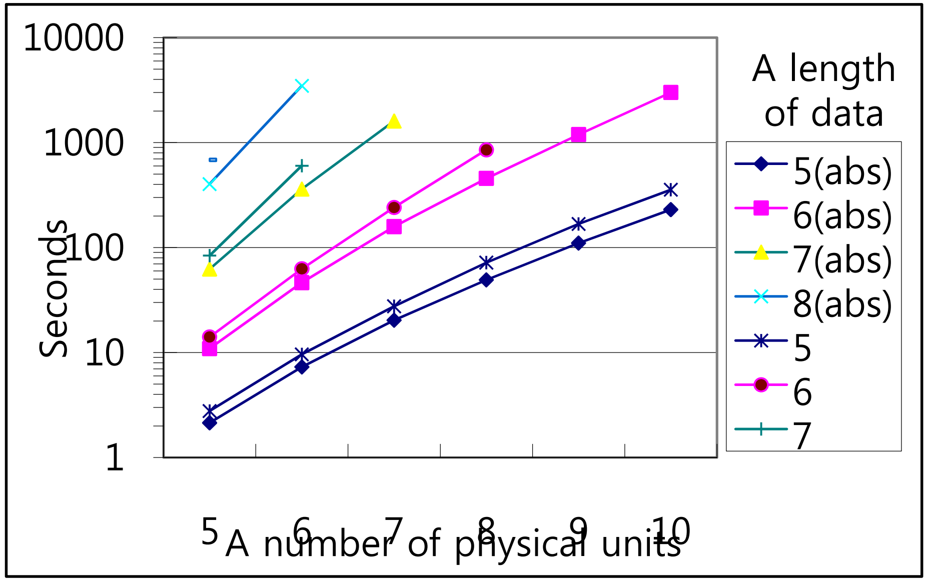
- Verification was performed on the machine equipped with Xeon5160 (3Ghz, 32Gbyte Memory), 64 bit Fedora Linux 7, NuSMV 2.4.3
- The requirement property was proved correct for all the experiments (i.e., MSR is correct in this small model)
    - For 7 sectors long data that are distributed over 7 PUs consumes more than 11 hours while consuming only 550 mb memory

Excerpts of the Spin Model

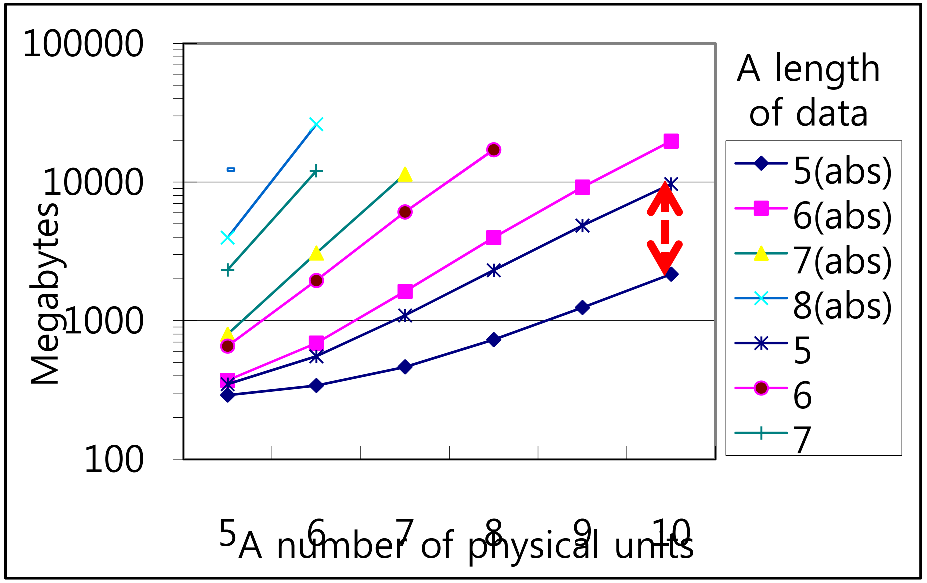
```
active proctype SM_ReadSectors() {

  byte buf[NUM_LS_USED];
  byte nScts;
  byte nFirstOffset;
  byte nNumOfScts=NUM_LS_USED;
  byte nReadScts=nNumOfScts;
  byte nSamIdx;

  do  /* 1047: while (nNumOfScts >0) { */
  ::  nNumOfScts > 0 ->
     PU_id = lui[nLun];
     if          /* nReadScts = ... */
     :: (SECT_PER_U-nSamIdx)> nNumOfScts ->
         nReadScts = nNumOfScts;
      :: else->nReadScts =SECT_PER_U- nSamIdx;
      fi;
      nNumOfScts = nNumOfScts - nReadScts;

      do       /* line 1068: while (nReadScts > 0) */
      ::  (nReadScts > 0) ->  PU_id = lui[nLun];
          nFirstOffset=255;
          nScts=1; nReadScts--;
```

```
do  /* line 1075:  do {…  */
::  true;
    if /* line 1077: if(pstCurrent->pSam[nSamIdx]…*/
    :: SAM[PU_id].valid[nSamIdx]->  nFirstOffset =
      SAM[PU_id].offset[nSamIdx];nSamIdx++;
      do  /* line 1084:while (nReadScts > 0) { …} */
      ::  (nReadScts > 0) ->
           if
           ::FirstOffset+nScts==
                SAM[PU_id].offset[nSamIdx] ->
              nScts++;nReadScts--;nSamIdx++;
           :: else-> break;
           fi;
      :: else->break;
      od;

      BML_MRead(PU_id,nFirstOffset,nScts,pBuf);
      break;
    :: else;
    fi;
    if /*line 1112: } while ( PU[PU_id].nil != true) */
    :: PU[PU_id].nil -> break;
    :: else;
    fi;
    PU_id++;
od;
}
```

Verification Performance of Spin

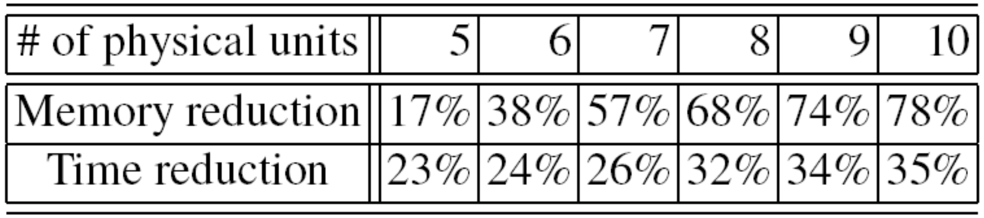


(a) Time consumption



(b) Memory consumption

- The requirement property was satisfied
- The data abstraction technique shows significant performance improvement upto 78% of memory reduction and 35% time reduction (for 5 logical sectors data)



# of physical units	5	6	7	8	9	10
Memory reduction	17%	38%	57%	68%	74%	78%
Time reduction	23%	24%	26%	32%	34%	35%

Modeling by CBMC

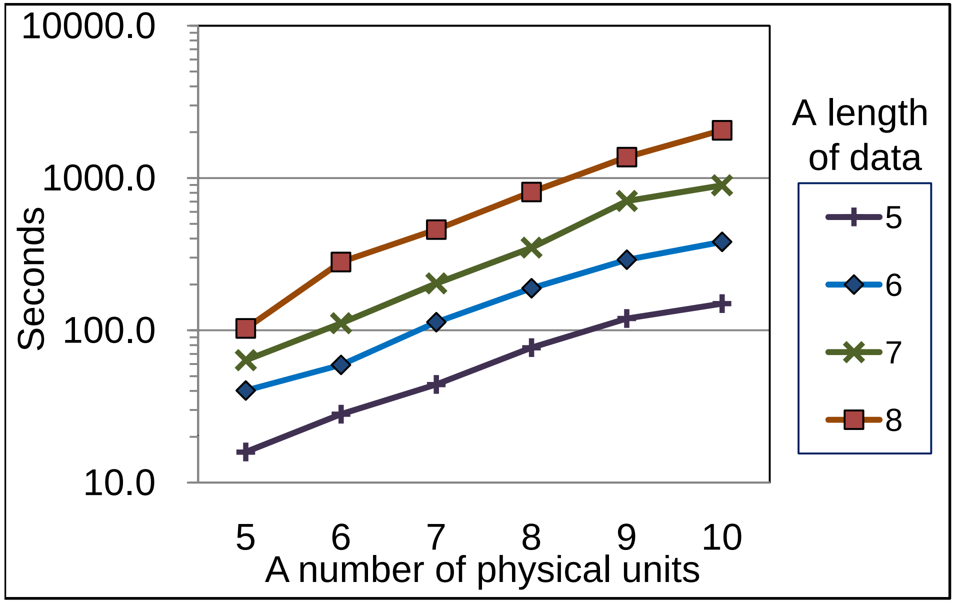
- CBMC does not require an explicit target model creation
- An environment for MSR was specified using assume statements and the environment model was similar to the environment model in NuSMV
- For the loop bounds, we can get valid upper bounds from the loop structure and the environment setting
  - The outermost loop: L times (L is a # of LUs)
  - The 2nd outermost loop: 4 times (one LU contains 4 LS's)
  - The 3rd outermost loop: M times (M is a # of PUs)
  - The innermost loop: 4 times (one PU contains 4 PS's)

L=2, M=5

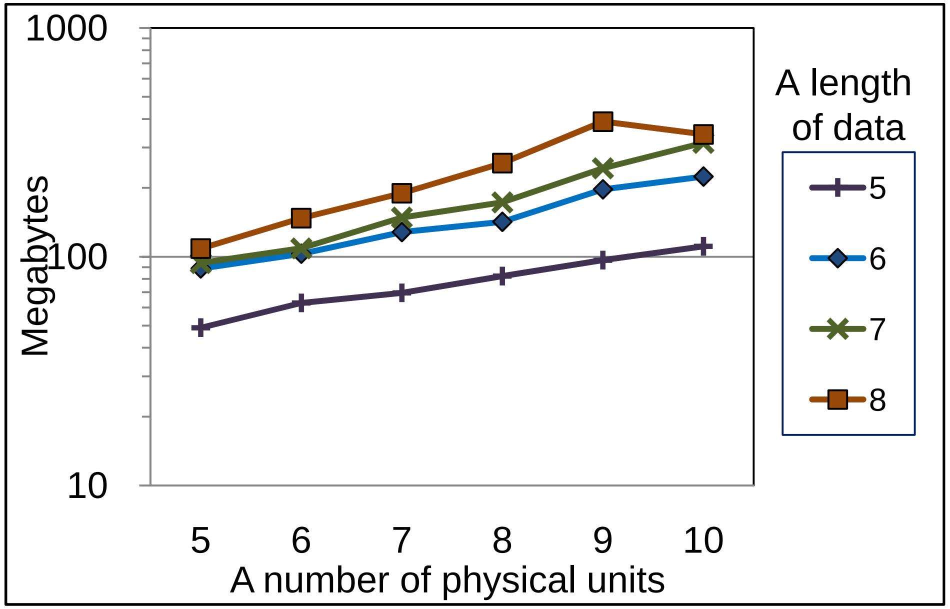
	SAM0~SAM4				
Sector 0	1			0	
Sector 1		1			1
Sector 2		2			
Sector 3			3		

	PU0~PU4				
Sector 0				E	
Sector 1	A	B			F
Sector 2		C			
Sector 3			D		

Verification Performance of CBMC



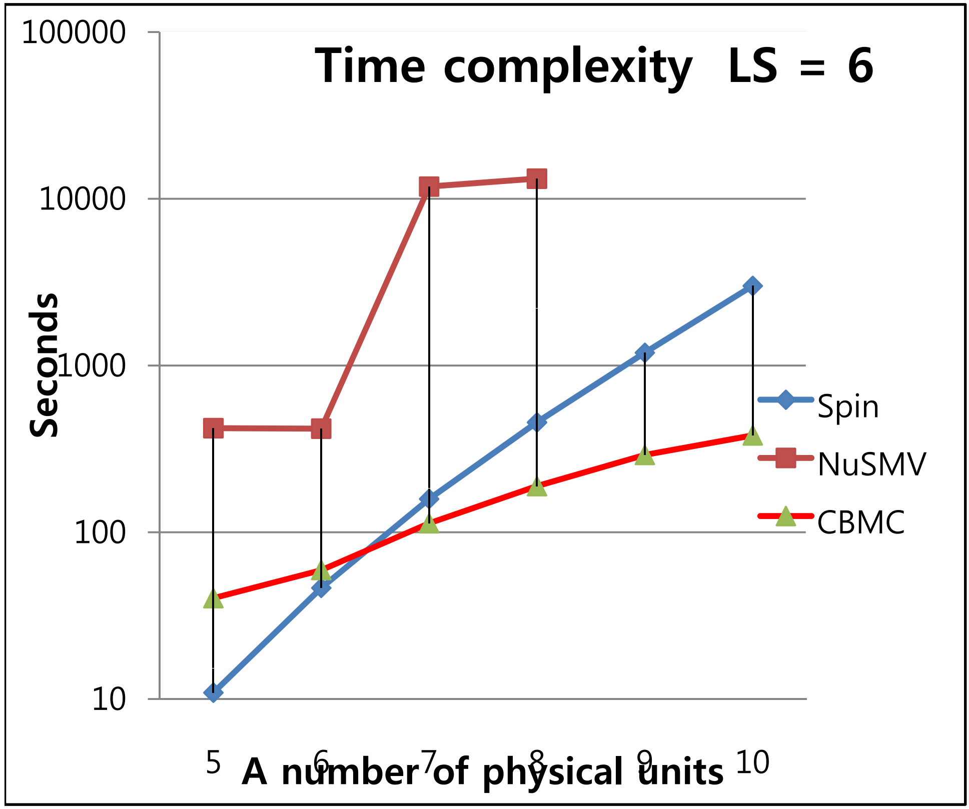
(a) Time consumption

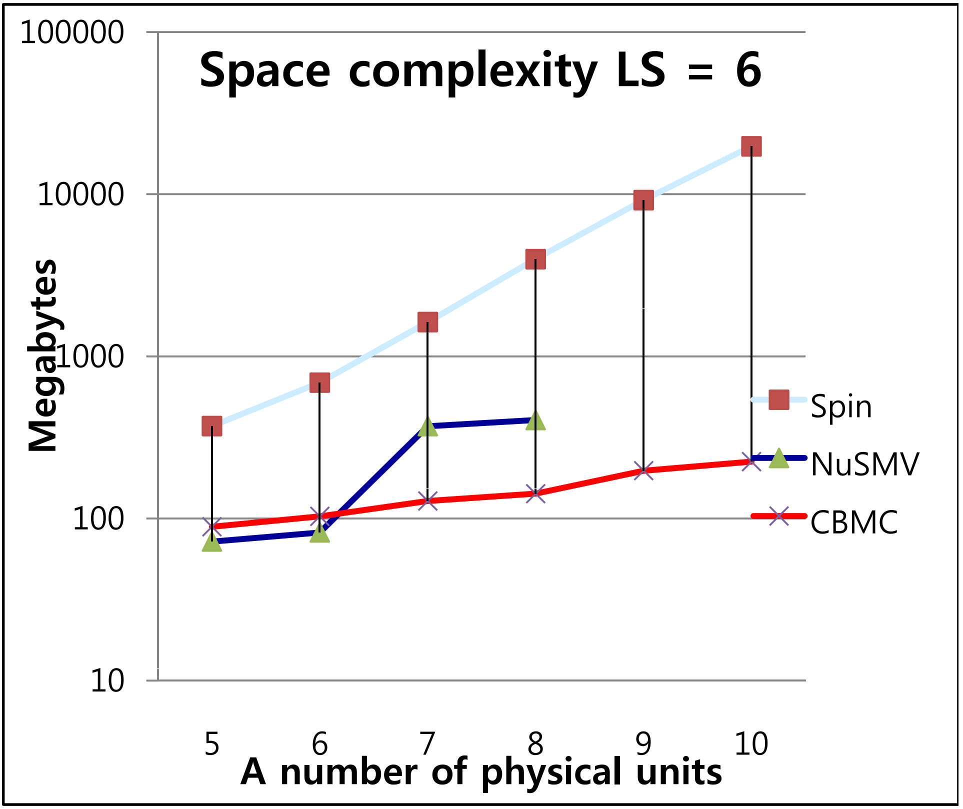


(b) Memory consumption

- Exponential increase in both time and memory. However, the slope is much lower than those of NuSMV and Spin, which makes CBMC perform better for large problems
- A problem of 10 PUs and 8 LS's has 8.6x10^5 variables and 2.9 x 10^6 clauses.

Performance Comparison





Conclusion

- We successfully applied CBMC to detect hidden bugs in the device driver for Samsung's OneNAND flash memory
  - Also, we established confidence in the correctness of the complex MSR
- Lessons learned
  - Software model checker as an effective unit testing tool
    - CBMC took modest amount of memory and time to detect bugs in USP
    - Exhaustive analysis can detect hidden bugs
  - Advantages of a SAT-based model checker
    - Analysis capability of whole ANSI-C
    - No abstract model required
- We believe that a SAT-based model checker can be utilized effectively as a unit testing tool to complement conventional testing