# Formal Verification of a Flash Memory Device Driver - an Experience Report


**Moonzoo Kim, Yunho Kim**

*Provable Software Lab. CS Dept. KAIST*

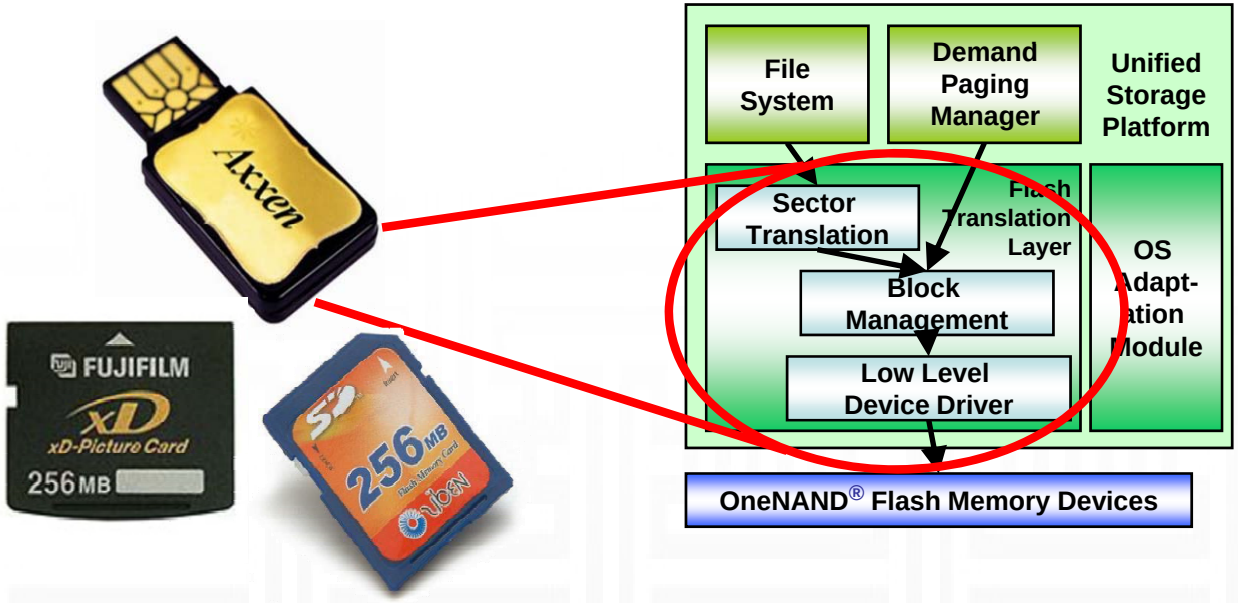

**Yunja Choi**

*School of EECS, Kyungpook National Univ.*


**Hotae Kim**

*Samsung Electronics*

# Summary of the Talk




- In 2007, Samsung requested to debug the device driver for the Samsung OneNAND™ flash memory, by using model checkers, for 6 months. This presentation describes a part of the result from the project.

# Overview

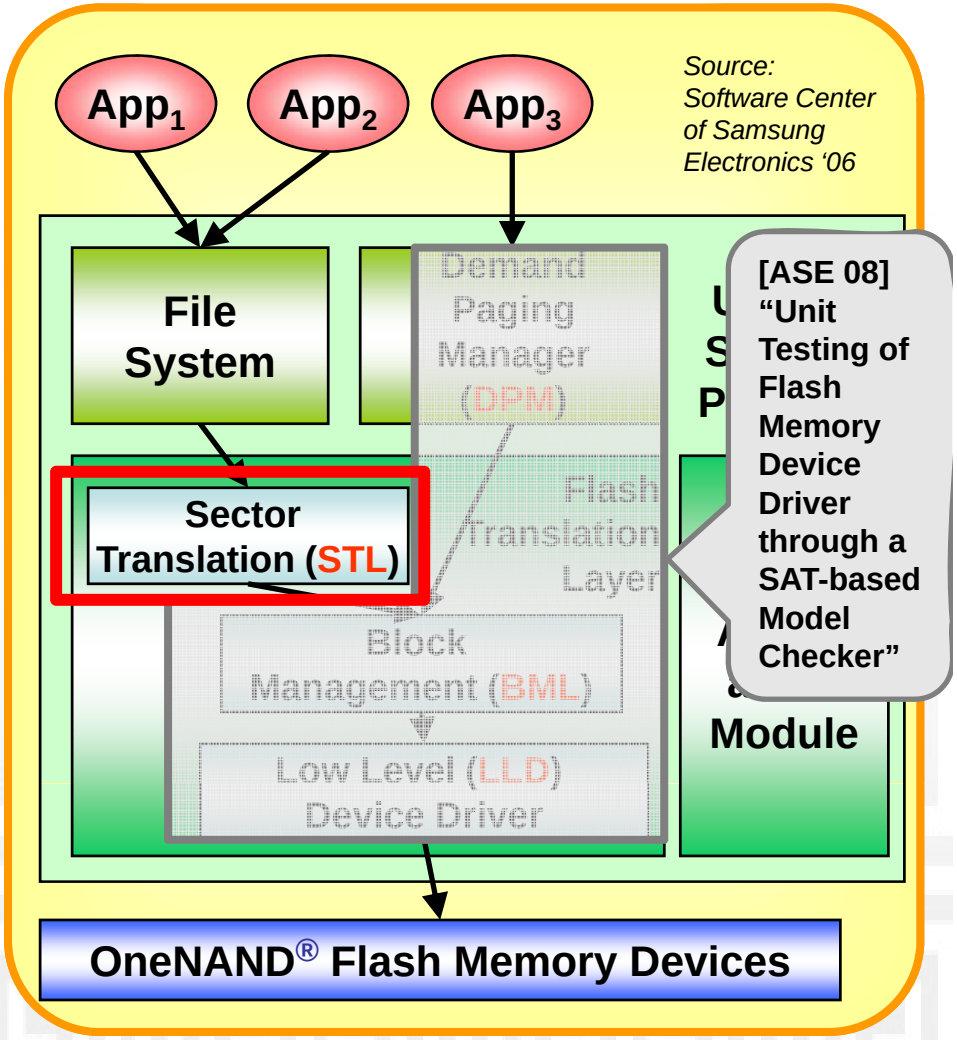
- **Background**
  - Overview of the Unified Storage Platform (USP)
  - Sector Translation Layer (STL)
  - Multi-Sector Read operation (MSR)
- **Model Checking MSR**
  - Reports on the following three aspects
    - Target system modeling
    - Environment modeling
    - Performance analysis on the verification
- **Three different types of model checkers are used**
  - BDD based symbolic model checking (NuSMV)
  - Explicit model checking (Spin)
  - C-bounded model checking (CBMC)

# PART I: Background

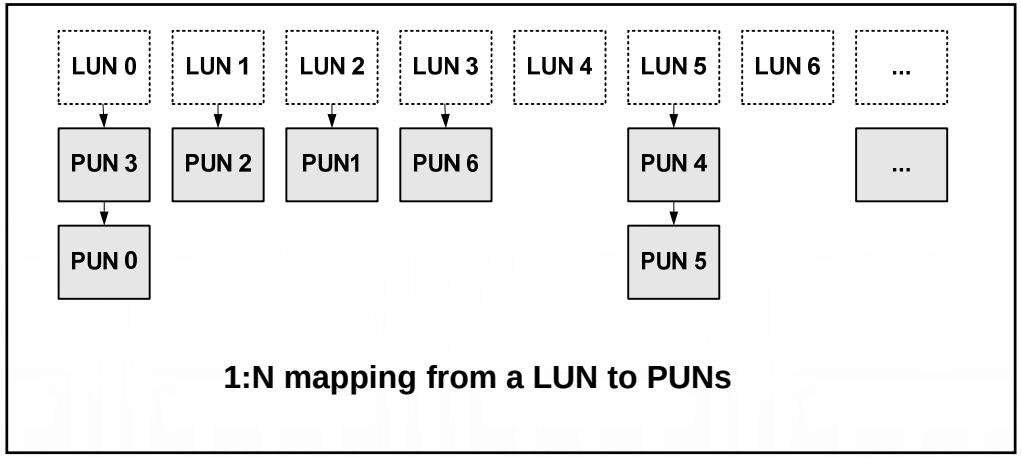
- **Unified Storage Platform (USP)**
  - Block diagram
  - Code statistics
- **Logical-to-physical sector translation**
  - Example of possible data distributions
- **Multi-Sector Read operation (MSR)**
  - Pseudo structure

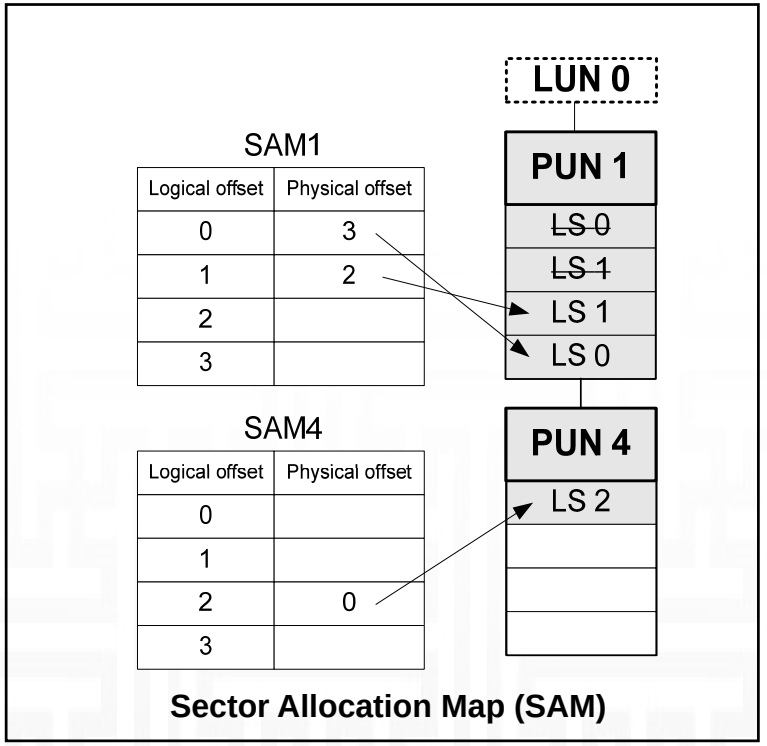
# Overview of the OneNAND® Flash Memory

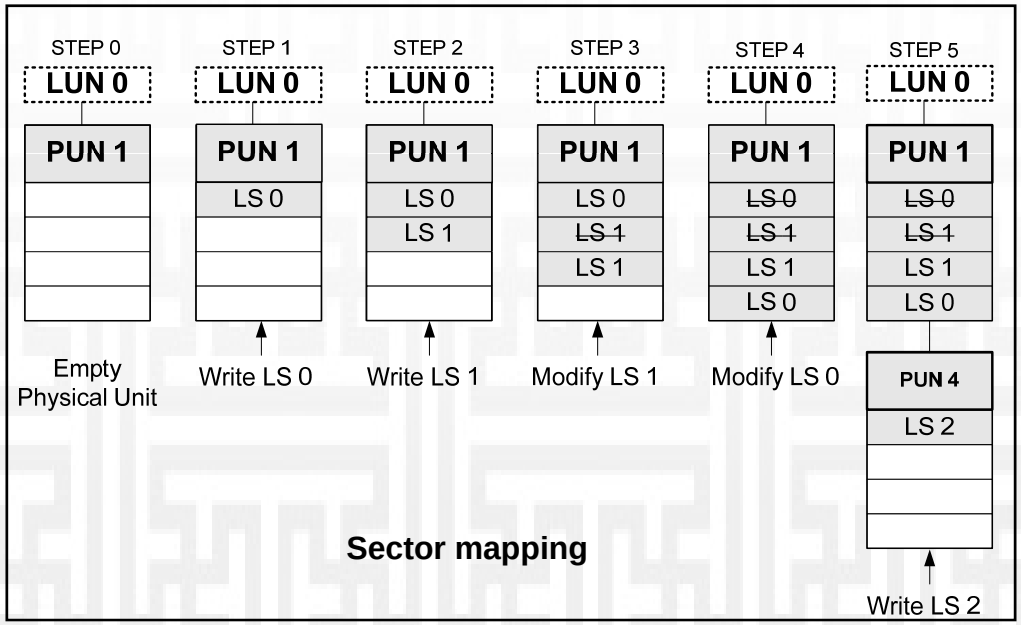
- **Characteristics of OneNAND®**
  - Each memory cell can be written limited number of times only
    - Logical-to-physical sector mapping
    - Bad block management
    - Wear-leveling
  - Performance enhancement
    - Multi-sector read/write
    - Asynchronous operations
    - Deferred operation result check



# Logical to Physical Sector Mapping

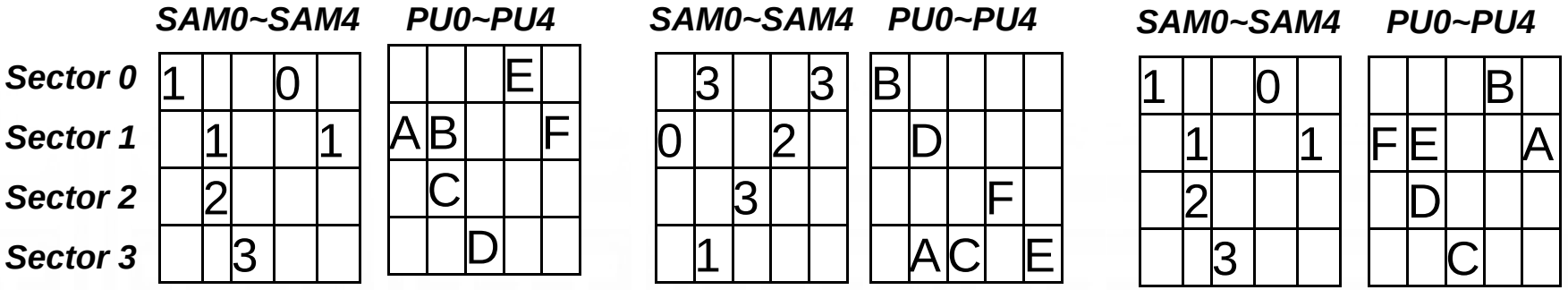



- **In flash memory, logical data are distributed over physical sectors.**

# Examples of Possible Data Distribution




(a) A distribution of "ABCDEF"

(b) Another distribution of "ABCDEF"

(c) A distribution of "FEDCBA"

- **Assumptions**
  - there are 5 physical units
  - each unit has 4 sectors
  - each sector is 1 byte long

# Multi-Sector Read Operations (MSR)

```
1032    nSamIdx = (UINT16)(nLsn % pstSHPC->nLogSctsPerUnit);
1033
1034    while (nNumOfScts > 0)
1035    {
1036        pstNew = pstSMC->pstLogUnitInfo[nLun].pstVirUnitInfo;
1037
1038        /* get the number of logical sectors to be read in a current logical uni
1039        nReadScts = ((pstSHPC->nLogSctsPerUnit - nSamIdx) > nNumOfScts) ?  nNumO
1040                                                                           (pstSH
1041        /* update nNumOfScts */
1042        nNumOfScts -= nReadScts;
1043
1044        if (pstNew != NULL)
1045        {
1046            /* construct SAM table */
1047            if (_ConstructSam(pstSMC, nLun, STL_LRU_POLICY) != STL_SUCCESS)
1048            {
1049                SM_ERR_PRINT((TEXT("[SM :ERR] _ConstructSam fail!! (Vol %d, Part
1050                              pstSMC->nVol, pstSMC->nPartID));
1051                SM_LOG_PRINT((TEXT("[SM :OUT] --SM_ReadSectors()\r\n")));
1052                return STL_CRITICAL_ERROR;
1053            }
1054
1055            while (nReadScts > 0)
1056            {
1057                pstCurrent   = pstNew;
1058                nFirstOffset = 0xFFFFFFFF;
1059                nScts        = 1;
1060                nReadScts--;
1061
1062                do
1063                {
1064                    if (pstCurrent->pSam[nSamIdx] < SM_SAM_DELETED)
1065                    {
1066                        /* get first sector offset */
1067                        nFirstOffset = pstCurrent->pSam[nSamIdx];
1068                        nSamIdx++;
1069
1070                        /* get the number of sequential sectors */
1071                        while (nReadScts > 0)
1072                        {
1073    1   2   3 4   5   6 7                     if ((nFirstOffset + nScts) == pstCurrent->pSam[nSamI
1074                            {
1075                                nScts++;
1076                                nReadScts--;
1077                                nSamIdx++;
1078                            }
1079                            else
1080                            {
1081                                break;
1082                            }
1083                        }
1084
1085                        /* read multiple sectors through BML */
1086                        nBErr = BML_MRead(pstVNC->nVol,
1087                                          pstSMC->pstSHPC->nStartVsn + nFirstOff
```

- **MSR reads consecutive physical sectors together for improving read performance**
- **Statistics**
  - 157 lines long
  - 4 level nested loops
  - 4 parameters to specify logical data to read (from where, to where, how long, read flag )

# Loop Structure of MSR

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

Loop1: iterates over LUs

Loop2: iterates until the current LU is read completely

Loop3: iterates over PUs linked to the current LU

Loop4: identify consecutive PS's in the current PU

# PART II: 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 -> ($\forall$ i. logical_sectors[i] == buf[i])
- **We compared these three model checkers empirically**

# Verification by NuSMV

- **NuSMV was the first choice as a verification tool, since**
  1. BDD-based symbolic model checkers have been known to handle large state spaces
  2. MSR operates with a semi-random environment (i.e. all possible configurations of PUs and SAMs analyzed)
  3. Data structure of MSR can be abstracted in a simple array form with assignments and equality checking operations only
  4. MSR is a single-threaded program

# Target Model Creation in NuSMV

- **We had to introduce control points variables, since**
  - C is control-flow based
  - NuSMV modeling language is dataflow-based
- **Linked list is replaced by an array operation.**
  - Array index variables should be statically expanded, since NuSMV does not support index variables
- **As a result, the final NuSMV model is more than 1000 lines long**

| A fragment of C | Conversion to parallel statements based on control and data dependency | Corresponding NuSMV code |
|---|---|---|
| 1: x=x-1; ← DP1<br>2: while(x>=0){<br>3: y = x; ← DP2<br>4: x --;} ← DP3 | 0: DP1=0; DP2=0; DP3=0;<br>1: if (!DP1) { x=x-1; DP1 =1;}<br>2: if ((DP1 \|\| DP3) && x>=0) {<br>y = x; DP2=1; DP3=0;<br>}<br>3: if (DP2) {<br>x--; DP3=1; DP2=0;<br>} | init(DP1):=0; init(DP2):=0; init(DP3):=0;<br>next(DP1):= 1;<br>next(DP2):= case (DP1 \| DP3) & (x >= 0) : 1;<br>DP2 : 0;<br>1 : DP2;<br>esac;<br>next(DP3):= case (DP1 \| DP3) & (x >= 0) : 0;<br>DP2 : 1;<br>1 : DP3;<br>esac;<br>next(x):= case !DP1 \| DP2 : x-1;<br>1 : x;<br>esac;<br>next(y):= case (DP1 \| DP3) & (x >= 0) : x;<br>1 : y;<br>esac; |

# Modeling in NuSMV (2/2)

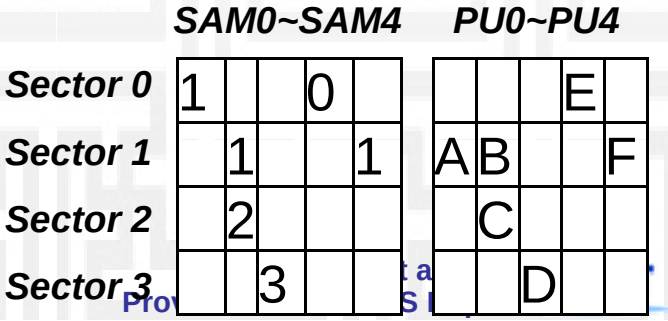
- **Environment model creation**
  - The environment of MSR (i.e., PUs and SAMs configurations) can be described by invariant rules. Some of them are
    1. One PU is mapped to at most one LU
    2. *Valid correspondence between SAMs and PUs:*

       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 ,

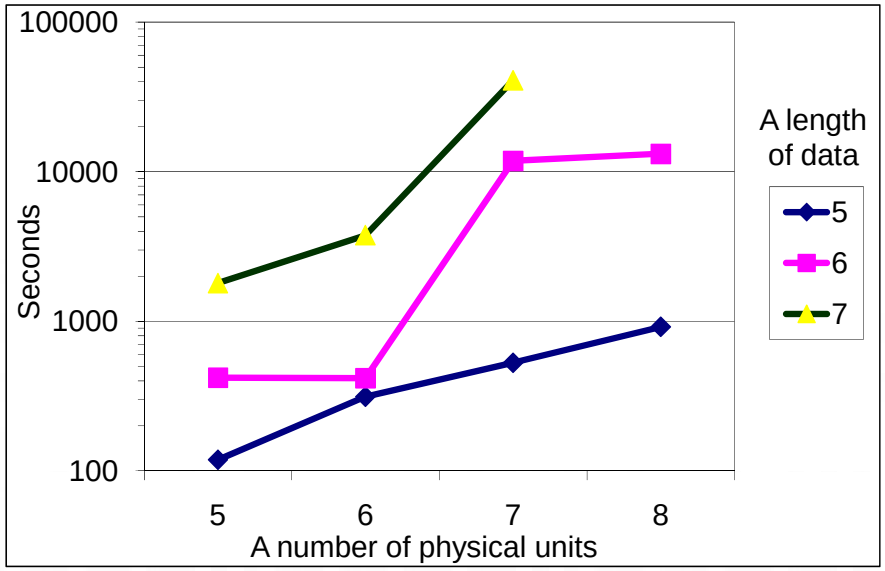
       Ex> 3rd LS ('C') is in the 3rd sector of the 2nd PU, then SAM1[2] ==2

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

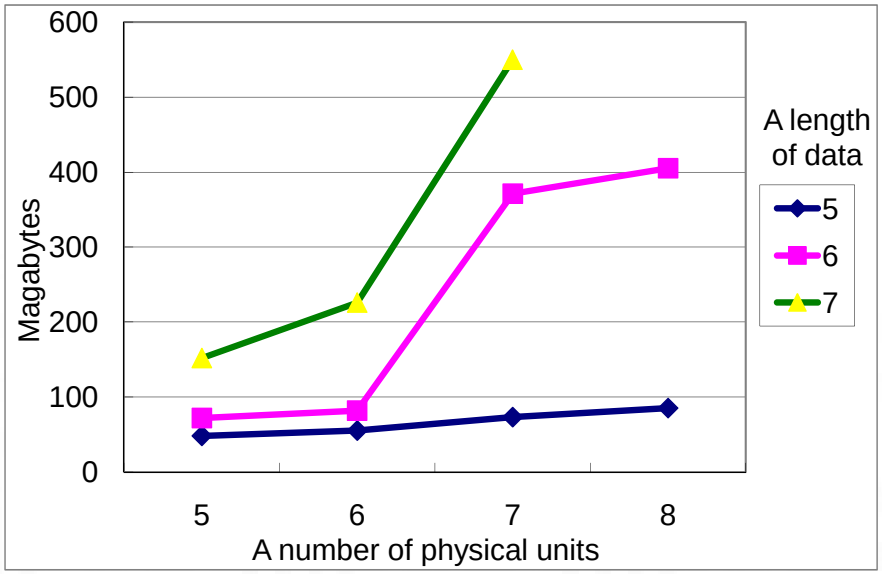
       The PS number of the *i* th LS must be written in only one of the (*i* mod *4*) th offsets of the SAM tables for the PUs mapped to the corresponding LU.



# 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

# Performance Analysis

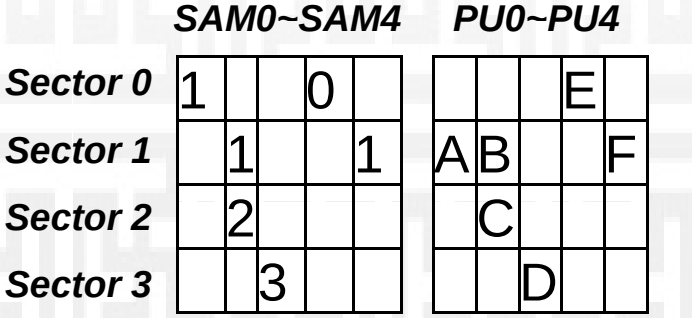
- **The MSR model (5 LS's and 5 PUs) has 365 BDD variables for its symbolic representation**
  - At least 240 BDD variables are required for PUs and SAMs
    - 5 (# of PUs) x 4 (sectors/PU) x 2 (current/next) x 3 (bits)
- **The same MSR model generated 1.2 million BDD nodes.**
- **Dynamic reordering takes more than 90% of total verification time**
  - Time is the bottleneck in this NuSMV verification task

# Modeling by Spin

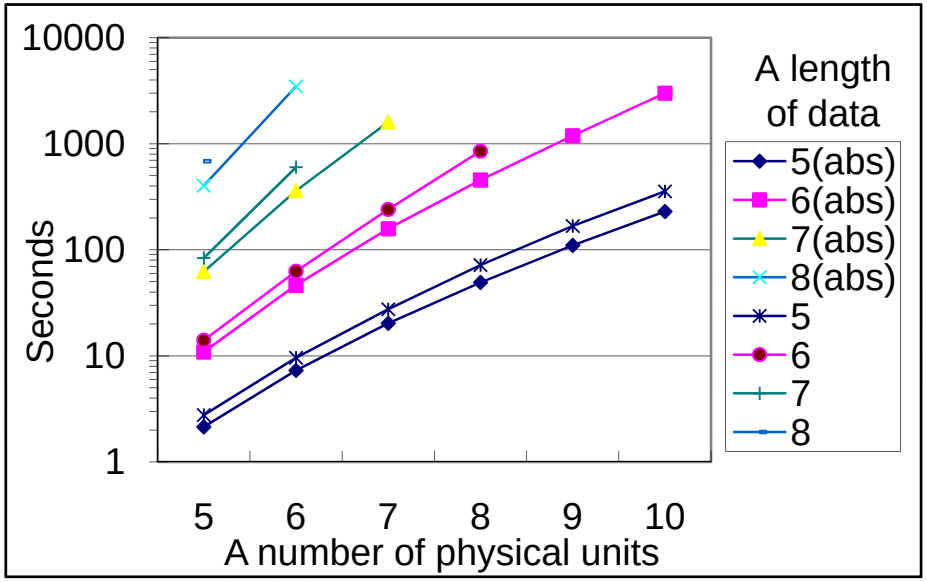
- **A target model**
  - Translated from the MSR C code through Modex which is an automated C-to-Promela translator with embedded C statements
    - Modex translates MSR into the same 4 level-nested loop control structure
- **An environment model**
  - PUs and SAMs, which takes most of memory, are tracked, but not stored in the state vector through a data abstraction technique
    - `c_track` keyword and `Unmatched` parameter
    - Based on the observation that SAMs and PUs are sparse
    - Only a unique signature of the current state of PUs and SAMs is stored succinctly
      - <(0,1),(1,1),(1,2),(2,3),(3,0),(4,1)>

      is the signature of the following

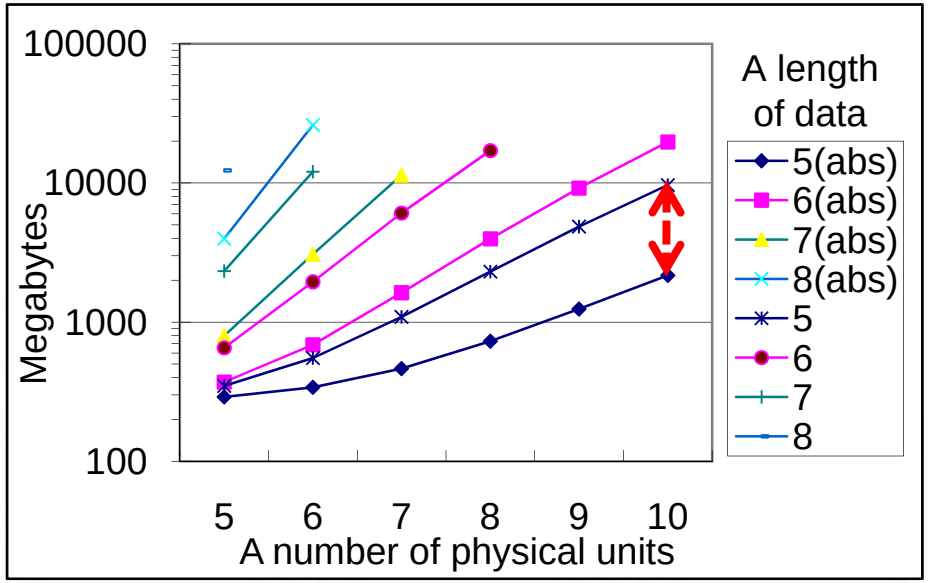
      PUs and SAMs configuration



# 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% |

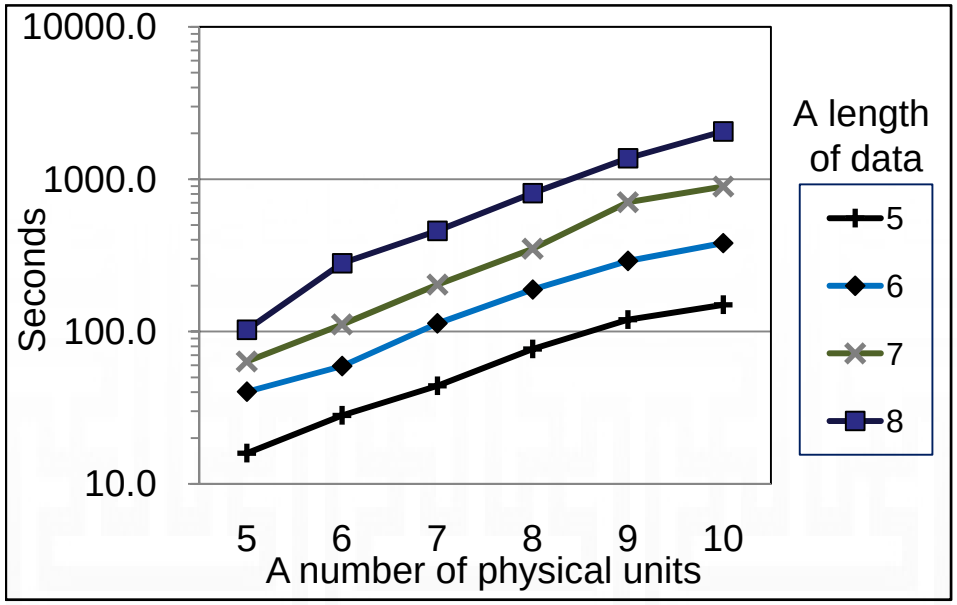
# Modeling by CBMC

- **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)

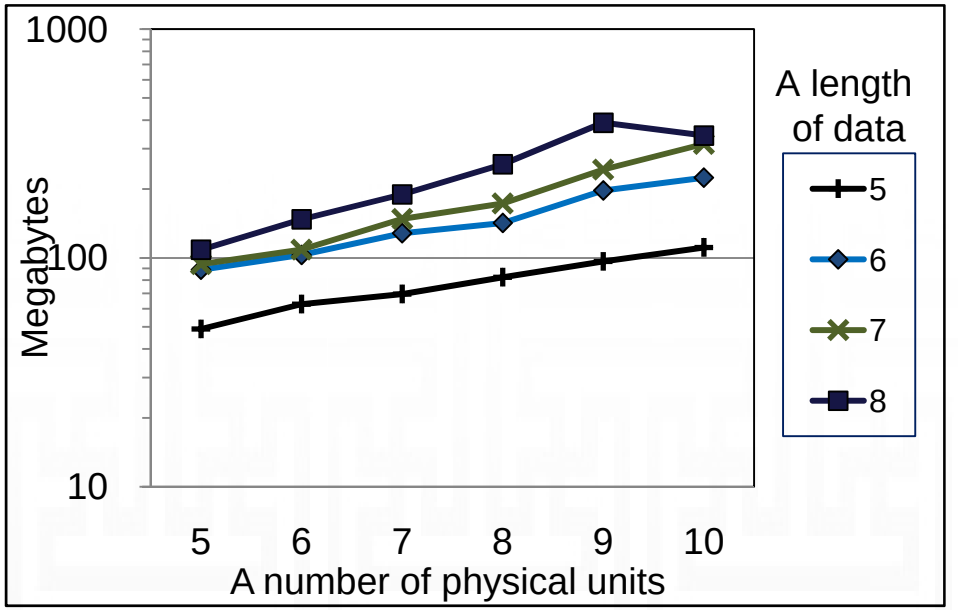
**L=2, M=5**

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

# Verification Performance of CBMC



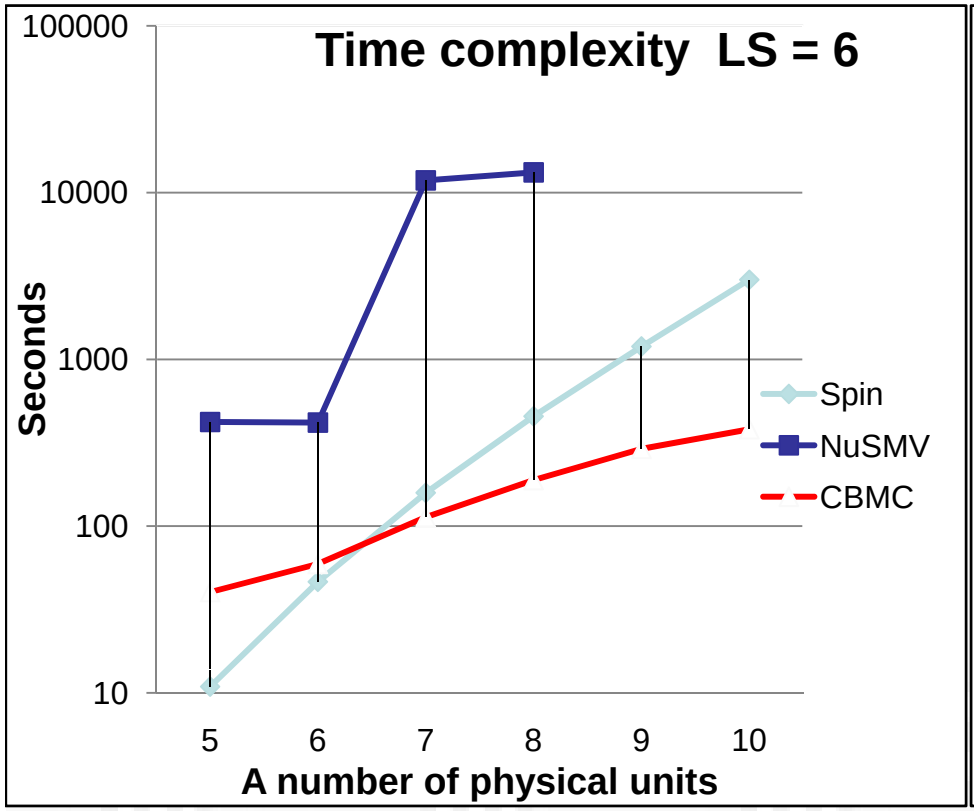

(a) Time consumption

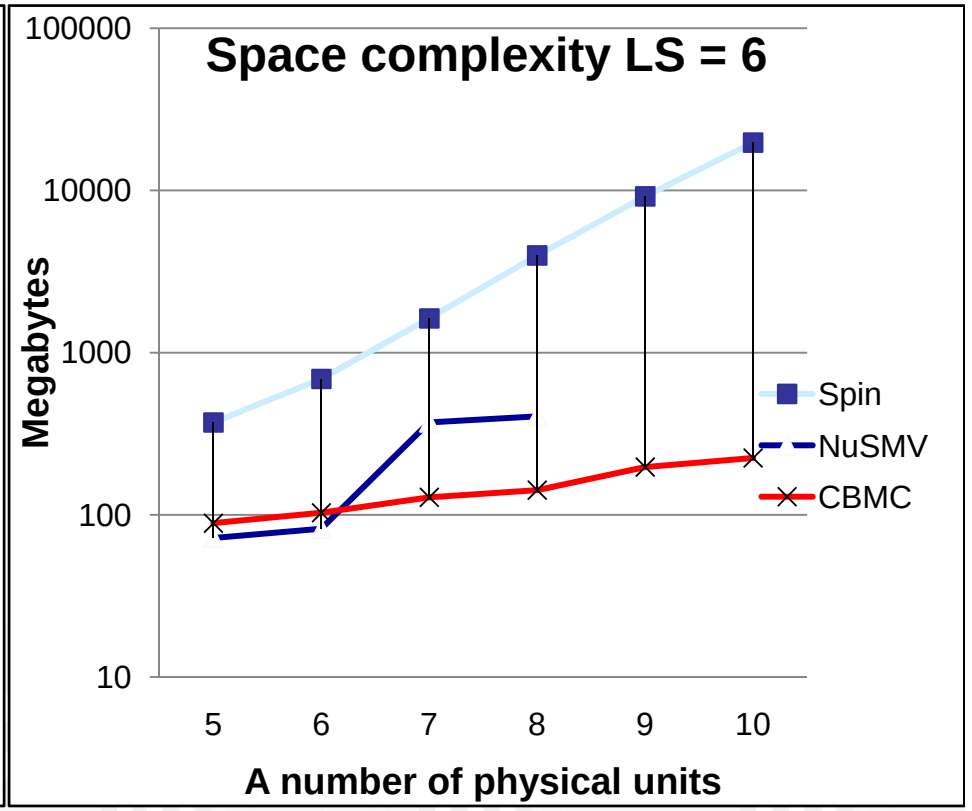



(b) Memory consumption

- 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 $8.6 \times 10^5$ variables and $2.9 \times 10^6$ clauses.

# Performance Comparison



# Conclusion

- **Application of Model Checking to Industrial SW Project**
  - Current off-the-shelf model checkers showed their effectiveness to debug a part of industrial software, if a target portion is carefully selected
  - Although model checker worked on a small scale problem, it still contributes due to its exhaustive exploration which is complementary to the testing result
- **Comparison among the Three Model Checkers**

| | Modeling Difficulty | Memory Usage | Verification Speed |
|---|---|---|---|
| NuSMV | Most difficult | Good | Slow |
| Spin | Medium difficult | Poor | Fast |
| CBMC | Easiest | Best | Fastest |