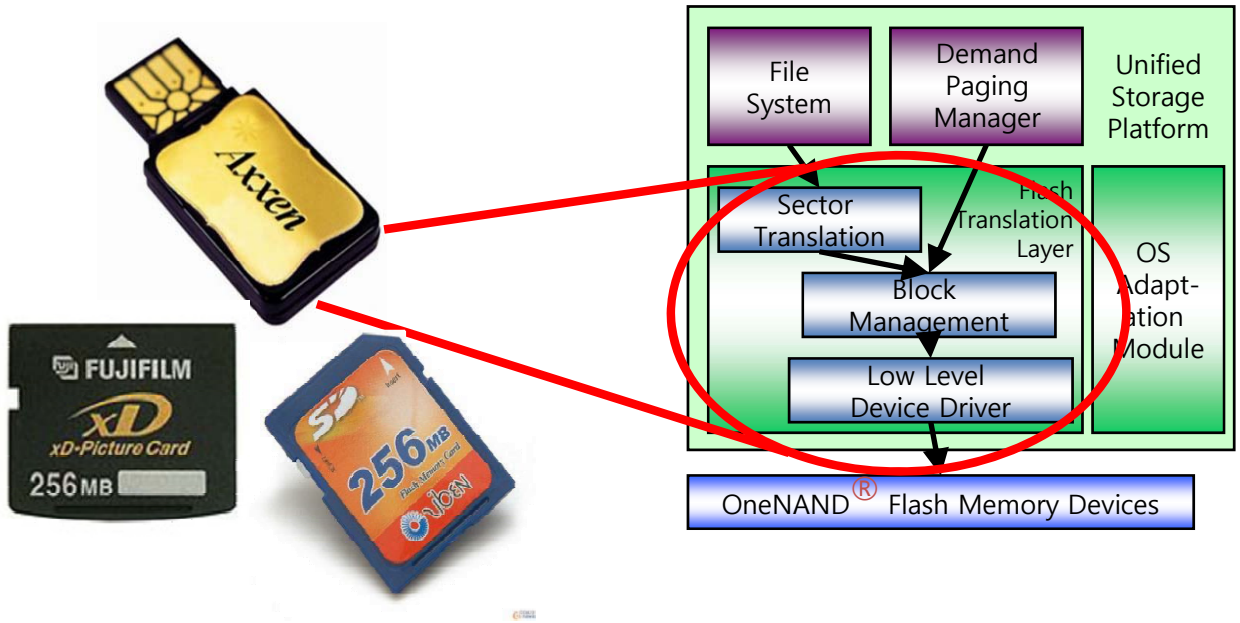
Concolic Testing of the Multi-sector Read Operation for Flash Memory File System

Moonzoo Kim and Yunho Kim
Provable Software Lab,
CS Dept, KAIST, South Korea
http://pswlab.kaist.ac.kr

Summary of the Talk



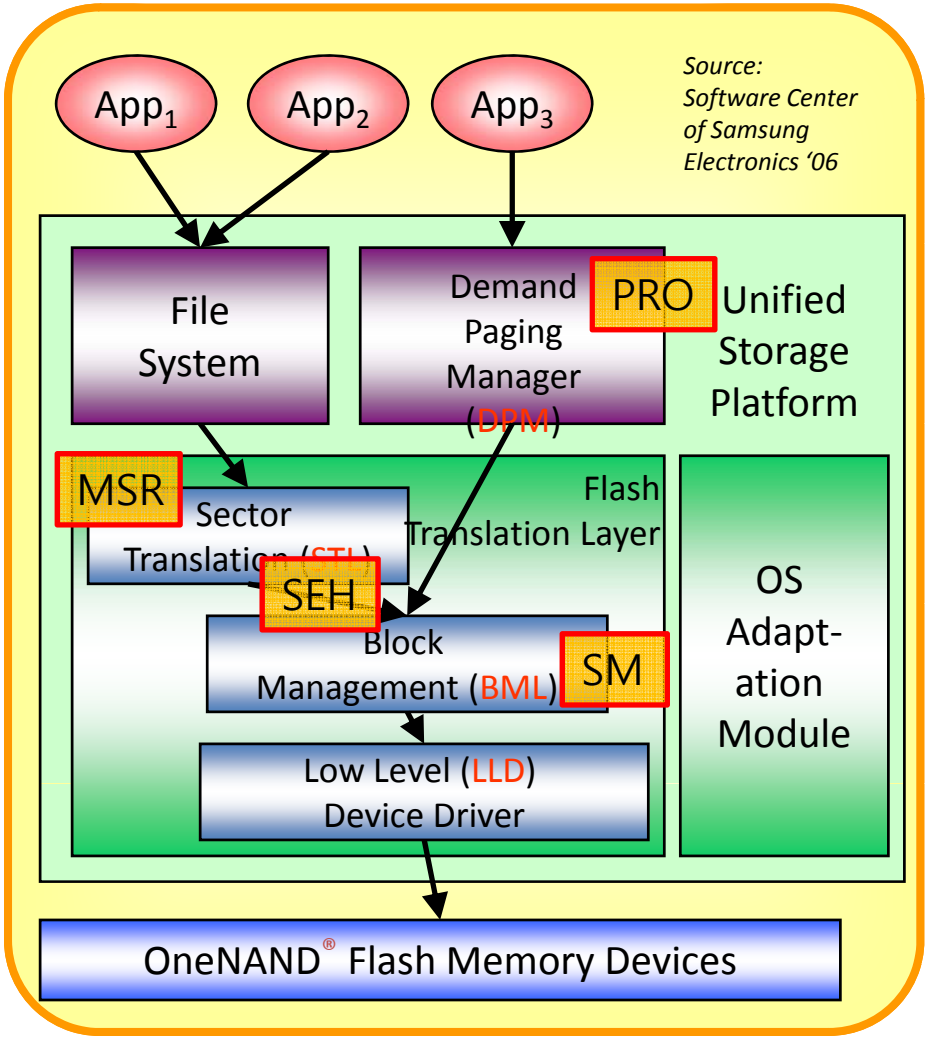
- Provable Software Lab @ KAIST has applied various formal verification technologies to the Unified Storage Platform code for the Samsung OneNAND™ flash memory
  - Conventional model checking: NuSMV and Spin [Spin 08]
  - Software model checking: C-Bounded Model Checker [ASE 08]
- In this talk, yet another approach using concolic testing.

Overview

- Part I: Background
  - Overview of the Unified Storage Platform (USP)
  - Summary of the Previous Studies on USP
    - 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)
- Part II: Concolic testing experiments on MSR
  - Overview of Concolic Testing
  - Multisector Read Operation
  - Experiments on MSR by using Concolic Testing
    - Testbed and experiment setup
    - Experiments with a constraint-based environment model
    - Experiments with an explicit-writing environment model
  - Lessons Learned
- Conclusion

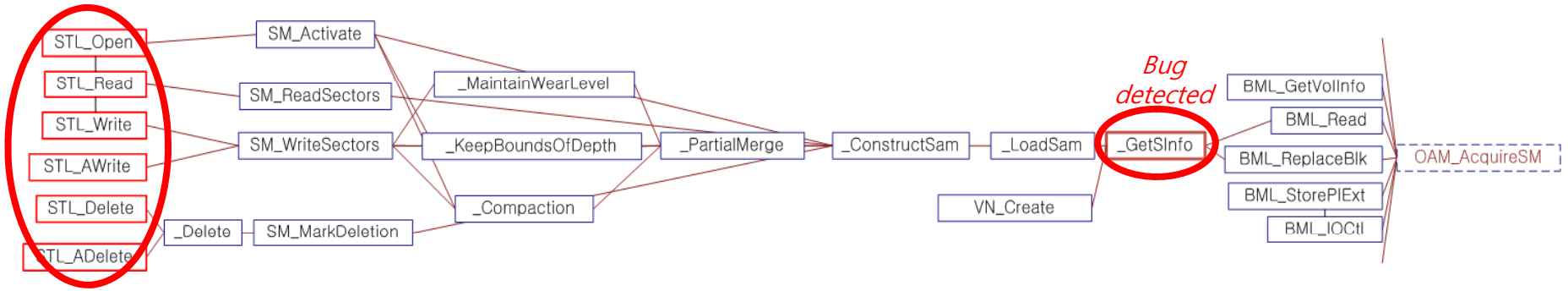
Overview of the Unified Storage Platform

- 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 scheme
    - Multiple processes access the device 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



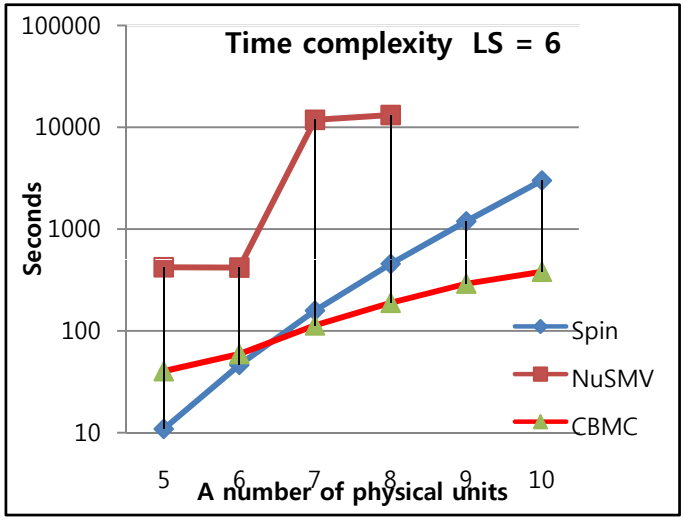
Summary of the Previous Studies (1/3)

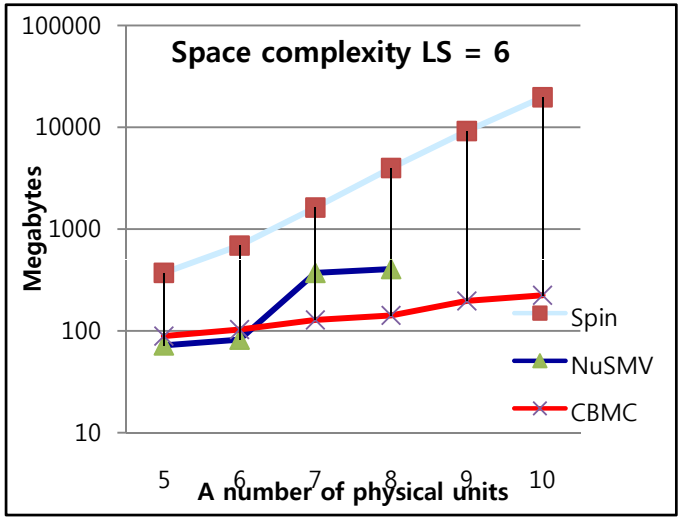
- SAT-based software model checker (i.e. CBMC) was successfully applied to industrial flash translation layer written in C [ASE 08]
  - Prioritized read operation
    - Detected a bug of not saving the status of suspended erase operation (~234 lines long)
  - Concurrency handling
    - Confirmed that the BML semaphore was used correctly in all 14 BML functions (~220 lines long on average)
    - Detected a bug of ignoring BML semaphore exceptions in a call graph (~2500 lines long on average)



Summary of the Previous Studies (2/3)

- Main target function: multi-sector read @ STL
  - Data intensive application due to SAMs and PUNs
  - Deterministic behaviors, except initial setting of data distribution
  - Data abstraction is barely possible for SAMs
- Performance comparison [Spin 08]
  - SAT-based bounded model checker (CBMC) > explicit model checking (Spin) > symbolic model checker (NuSMV)
  - CEGAR based software model checker (i.e. Blast) failed to analyze MSR due to its limitation on array/pointer operations





KAIST

Summary of the Previous Studies (3/3)

- However, we are still limited to miniature world (~10 PUNs) for the complete analysis. Thus, we may try
  - Theorem proving without bound
  - Testing
    - Applying concolic testing aiming for high coverage and better scalability

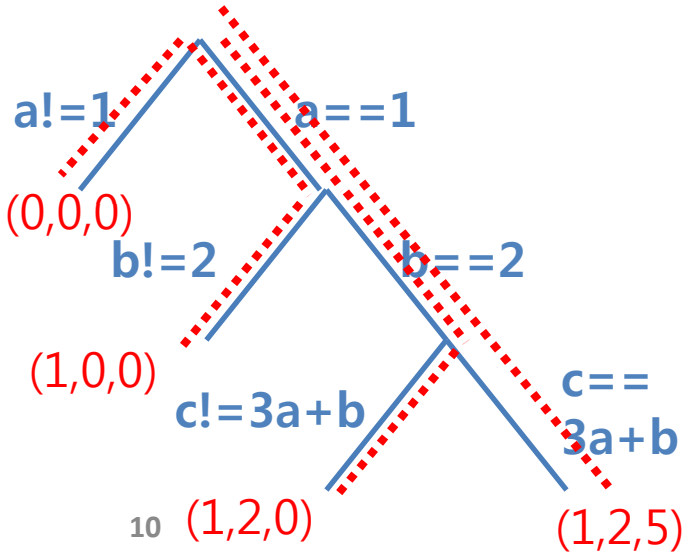
Part II: Concolic testing experiments on MSR

Concolic (CONCrete + symbOLIC) Testing

- Automated Scalable Unit Testing of real-world C Programs
  - Execute unit under test on automatically generated test inputs so that all possible execution paths are explored
    - (a.k.a) explicit path model checking
- In a nutshell
  - Use concrete execution over a concrete input to guide symbolic execution
    - A symbolic path formula is obtained at the end of an execution
  - One branch condition of the path formula is negated to generate the next execution path
  - The next execution path formula is solved by SMT solver to generate concrete input values, and so on
  - No false positives or scalability problem

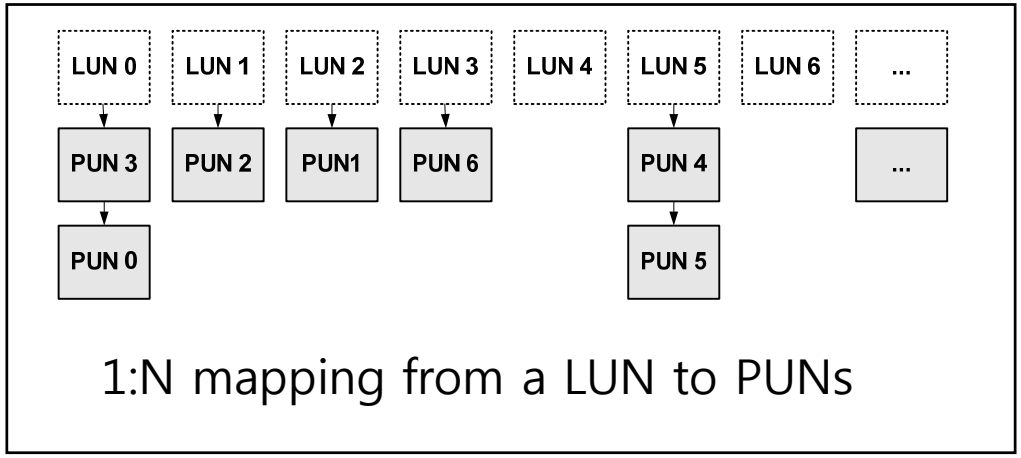
Concolic Testing Example

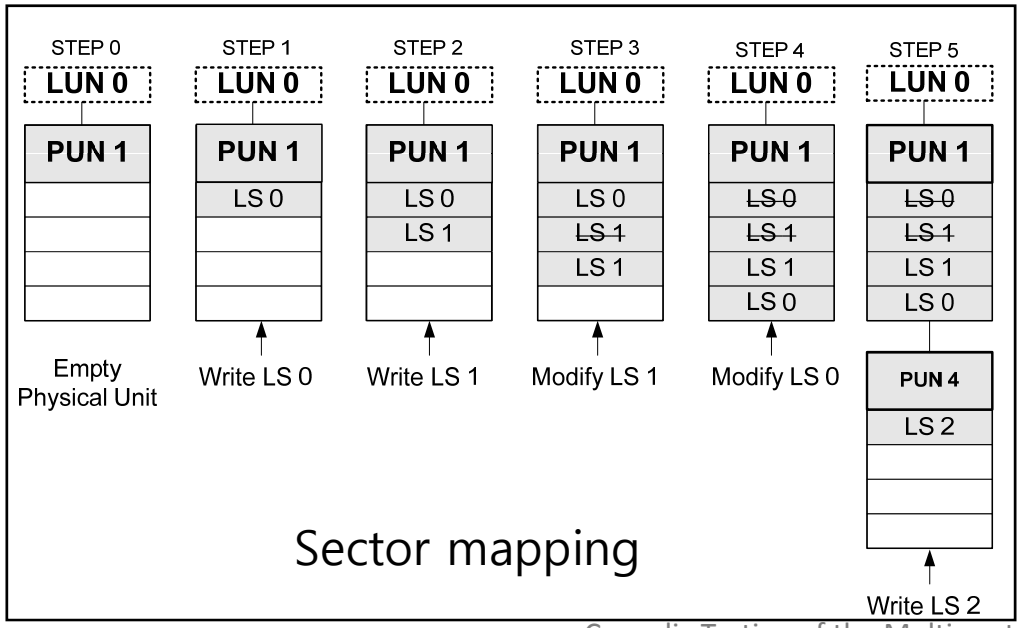
```
int main(void) {
  int a, b, c, d; // Test input
  CREST_int(a);
  CREST_int(b);
  CREST_int(c);

  if (a == 1) {
    if (b == 2) {
      if (c == 3*a + b) {
        fprintf(stderr, "GOAL!\n");
  } } } }
```

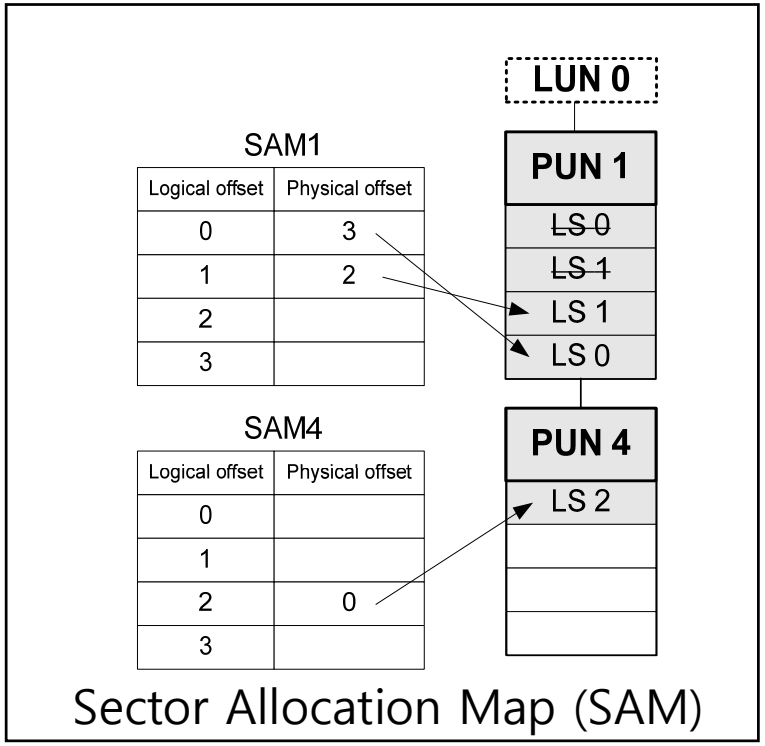


- Test input: (a,b,c)
  - a,b,c are declared as symbolic non-deterministic values by CREST_int();
- 4 test cases generated
  - (0,0,0)
    - initial random input
    - Executed path formula: a!=1
    - Next path formula: !(a!=1) (i.e, a ==1)
  - (1,0,0)
    - Executed path formula: a==1 && b!=2
    - Next path formula: a==1 && !(b!=2)
  - (1,2,0)
    - Executed path formula:
      a==1 && (b==2) && (c!=3*a +b)
    - Next path formula:
      a==1 && (b==2) && !(c!=3*a +b)
  - (1,2,5)
    - Covered all paths

Logical to Physical Sector Mapping

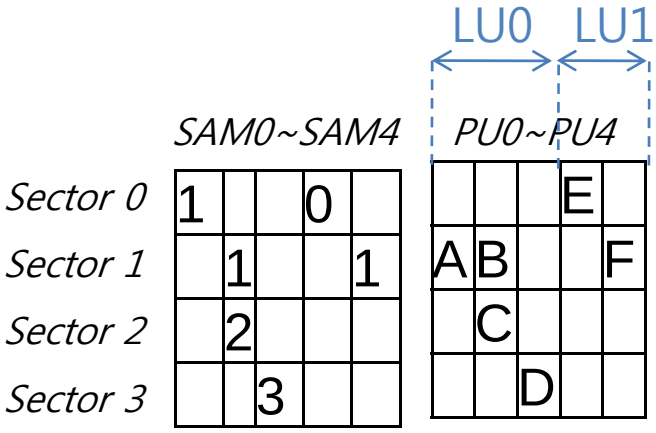




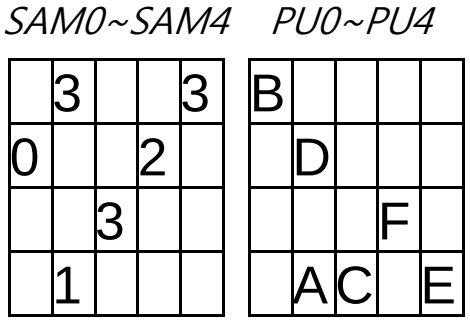


- In flash memory, logical data are distributed over physical sectors.

Examples of Possible Data Distribution



(a) A distribution of "ABCDEF"



(b) Another distribution of "ABCDEF"

- Assumptions
  - there are 5 physical units
  - each unit has 4 sectors
  - each sector is 1 byte long
- Exponentially many distributions according to size of data and # of PUNs
  - ex> 2.7 x 10^8 distributions for 6 sectors long data over 10 PUNs

Environment Modeling

- 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:*

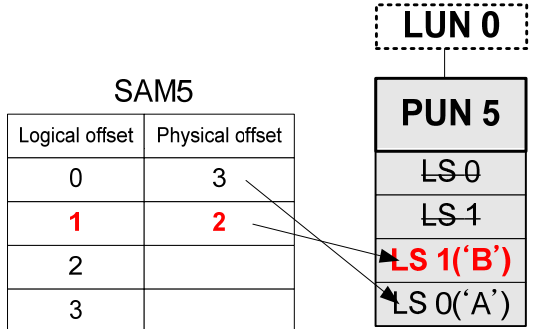
       If the *i* th LS is written in the *k* th sector of the *j* th PU, then the *i* th offset of the *j* th SAM is valid and indicates the k'th PS ,

       Ex> 1st LS ('B') is in the 2nd sector of the 5th PU, then SAM5[1] ==2

       i=1 k=2 j=5
    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.

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



Experiment Setup

- Hypotheses
  - H1: Concolic testing is effective for analyzing the MSR code
  - H2: Concolic testing is more efficient than model checking for analyzing the MSR code
- Effectiveness evaluation through mutation analysis
  - We injected the three types of frequent bugs and one corner case bug
    - 3 instances of off-by-1 bugs b_{11} to b_{13}
      - Ex. while(numScts>**0**) -> while(numScts>**1**)
    - 3 instances of invalid condition bugs b_{21} to b_{23}
      - Ex. if(SAM[i].offset[j]**!=**0xFF) -> if(SAM[i].offset[j]**==**0xFF)
    - 3 instances of missing statement bugs b_{31} to b_{33}
      - Ex. Missing nScts=1 in the second loop
    - A corner case bug b_c
      - 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')

Testbed for the Concolic Testing

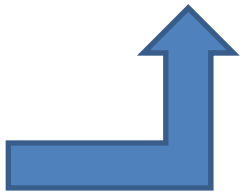
- Intel Core2Duo 3Ghz processor and 16 gigabytes of memory
- For concolic testing, CREST 0.1.1 with DFS option was used
  - CREST does not support dereferencing of pointers and array index variables in the symbolic analysis.
    - the target MSR code was modified to use an array representation of the SAMs and PUs.
  - gcc 4.3.0, Yices 1.0.19
- For model checking, CBMC 2.6 and MiniSAT 1.14 were used.
  - The target MSR codes used for concolic testing and model checking are identical

Constraint-based Environment Model

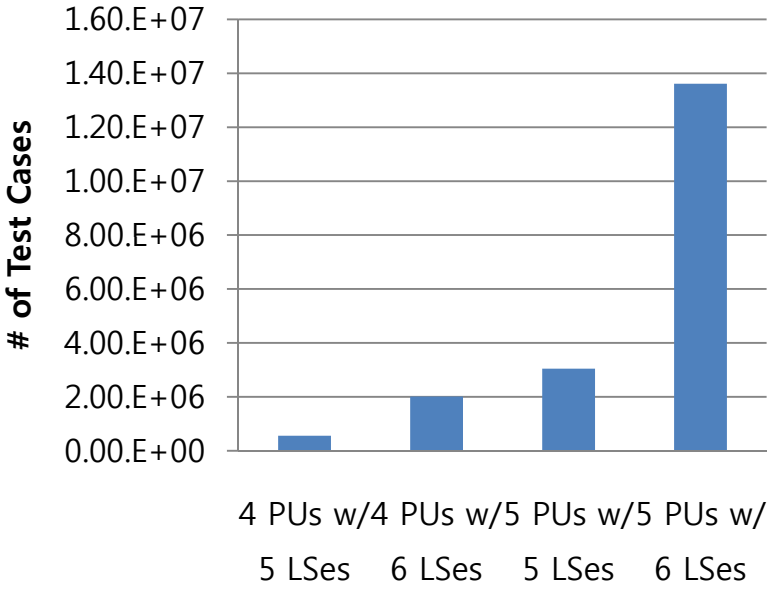
- We have to specify test input variables as symbolic variables
  - pun[i].sect[j]
  - SAM[i].offset[j]
- and put constrains on them
  - If assigned input value does not satisfy the constraints (i.e. invalid test case generated), a current iteration terminates immediately without testing MSR (goto out);

```
for (i=0; i<NUM_PUN; i++){ for (j=0; j<SECT_PER_U; j++){
  CREST_unsigned_char(pun[i].sect[j]);
  CREST_unsigned_char(SAM[i].offset[j]); } }

for (i=0; i<NUM_LS_USED; i++){
    for (j=0; j<NUM_PUN; j++){
        for (k=0; k<SECT_PER_U; k++){
            if (pun[j].sect[k] == 'a'+i){
                if (i < SECT_PER_U  && j < NUM_PUN_LUN0 ||
                    SECT_PER_U <= i && j >= NUM_PUN_LUN0){
                    valid[i] = 1;
                }else{ goto OUT; }
            }else continue;
            if (!(!('a' + i == pun[j].sect[k]) ||
                ( SAM[j].offset[((i>=SECT_PER_U)?
                  (i-SECT_PER_U):i)]==k)
               )){ goto OUT; }
            ...
```

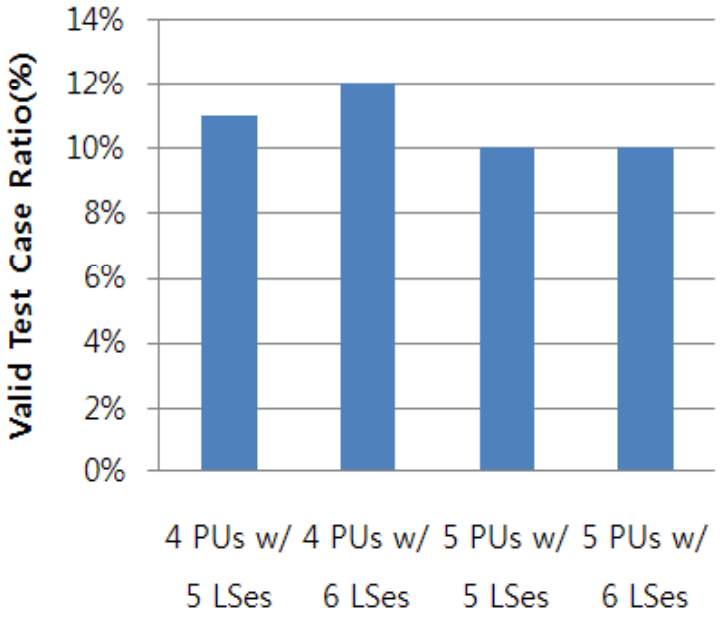
$$\forall i, j, k\ (LS[i] = PU[j].sect[k] \rightarrow (SAM[j].valid[i \bmod m] = true \\ \& SAM[j].offset[i \bmod m] = k \\ \& \forall p.(SAM[p].valid[i \bmod m] = false) \\ \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th} LU))$$



Result w/ Constraint-based Model (1/2)



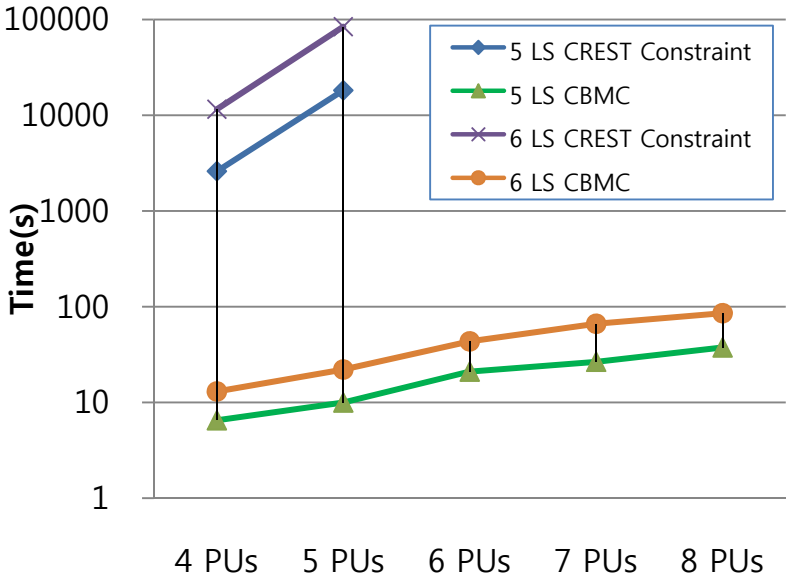
(a) Total number of test cases generated



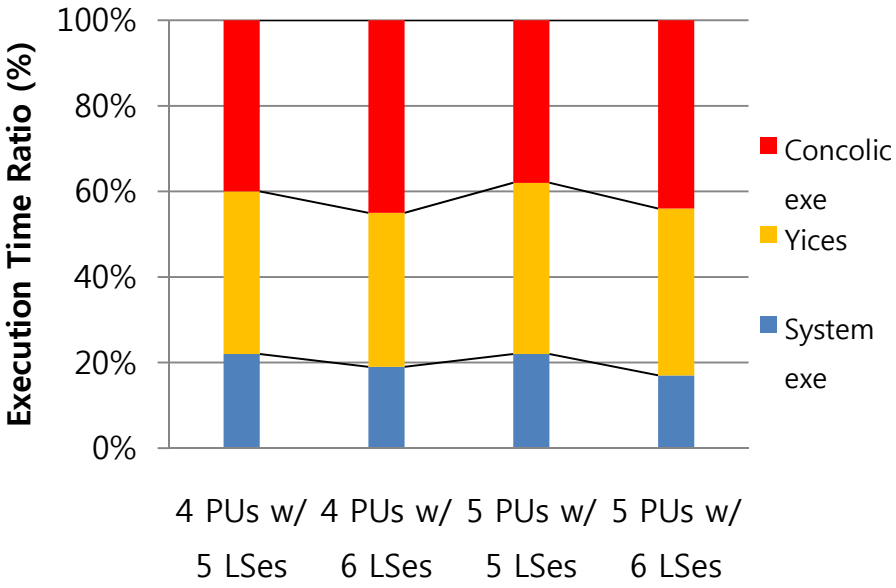
(b) Ratio of valid test cases/all test cases

- Only ~10% of generated test cases are valid
  - Causing significant overhead
- However, valid test cases generated cover all distribution cases
  - i.e. 100% path coverage achieved
  - Consequently, all bugs b_{11} to b_{13} as well as b_c were detected

Result w/ Constraint-based Model (2/2)



(a) Total analysis time



(b) Time ratio of analysis steps

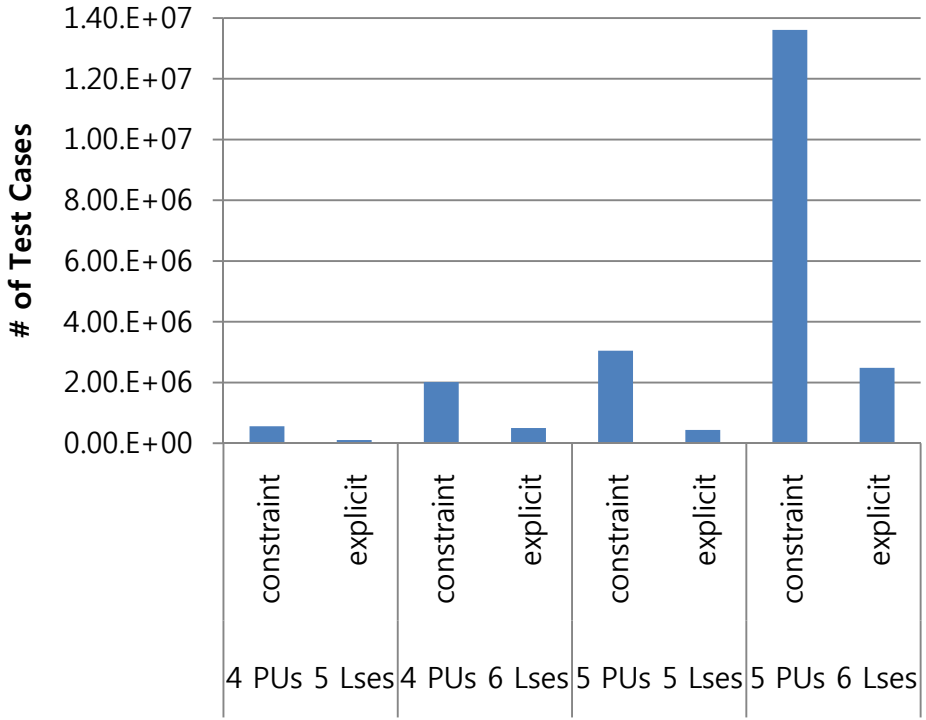
- Concolic testing is order of magnitude slower than CBMC
  - Concolic execution, SMT solving, system execution (i.e process fork and release) constitutes the overall overhead
  - Particularly, numerous invalid test cases (~90% of all test cases) worsen the performance

Explicit Environment Model

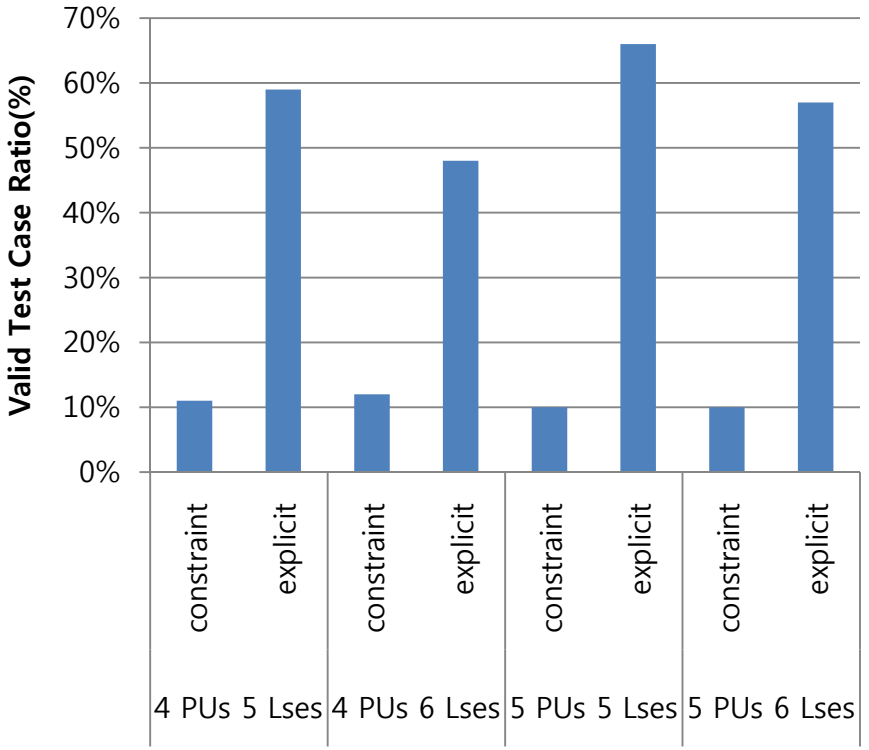
- Explicit environment model writes data to physical sectors explicitly
  - Thus , creating invalid test cases much less than the constraint-based model
- Test input variables
  - idxPU and idxSect for each logical data
- CREST has a limitation on array index variable
  - We should expand array index variables using 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
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) {
```

Result w/ Explicit Environment Model (1/2)



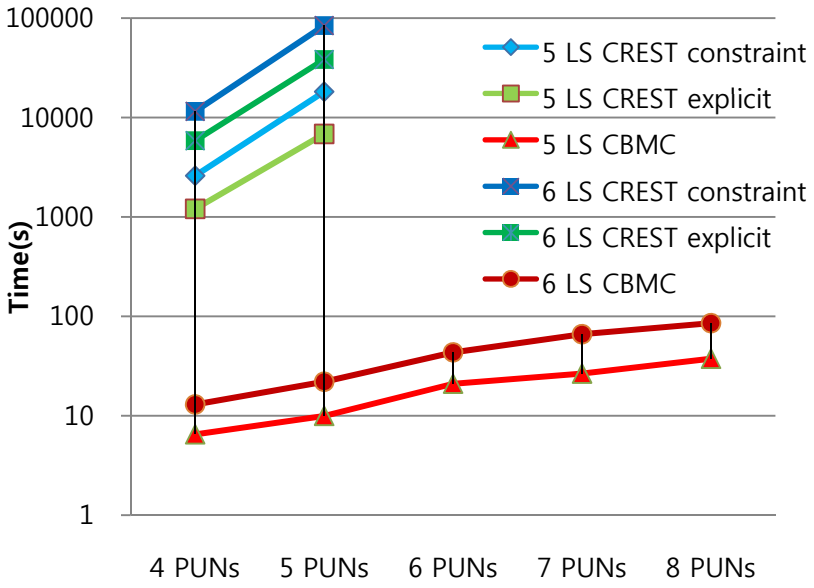
(a) Total number of test cases generated



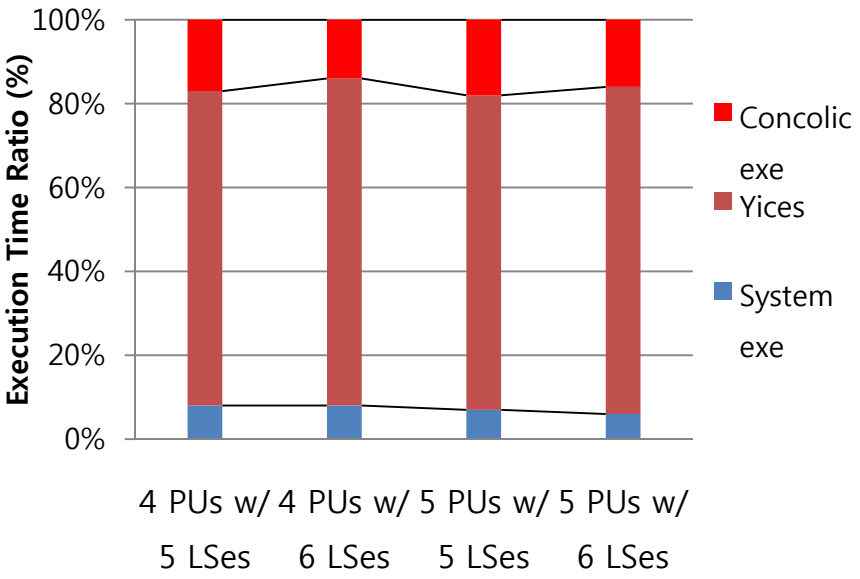
(b) Ratio of valid test cases/all test cases

- ~60% of generated test cases are valid
  - total test cases generated is 1/5 of the constraint-based one
- Again, valid test cases generated cover all distribution cases
  - Consequently, all bugs b_{11} to b_{13} as well as b_c were detected

Result w/ Explicit Environment Model (2/2)



(a) Total analysis time



(b) Time ratio of analysis steps

- Still, concolic testing is order of magnitude slower than CBMC
  - In this case, SMT solving is a major bottleneck, taking ~75% of total execution time

Lessons Learned

- Effectiveness of Concolic Testing
- Low Efficiency of Concolic Testing
  - Poorer performance compared to CBMC
  - But still it can be practically scalable by aiming branch coverage, not path coverage
- Importance of an Environment Model
  - Environment model constitutes an important part of any serious verification tasks
- Hard characteristic of MSR for Concolic testing
  - Different values of one SAM entries leads to different execution paths
  - Hard to apply abstraction

Future Works

- Study characteristics of symbolic path formulas
  - Apply heuristics to optimize solving performance
- Build a concolic testing tool which overcomes the limitation of CREST and can be tuned for embedded software environment
  - Currently discussing with Samsung Advanced Institute of Technology.
- Build a mock flash FTL, which can be used in a concolic testing framework
  - Inspired by Microsoft [AST 2009]