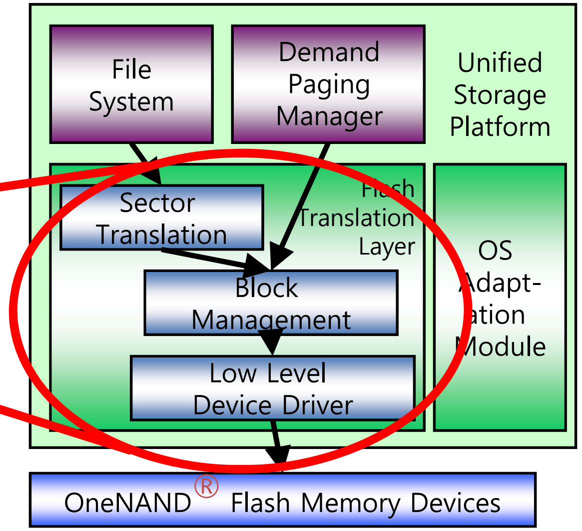
Concolic Testing of the Multi-sector Read Operation for Flash Memory File System

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

Summary of the Talk



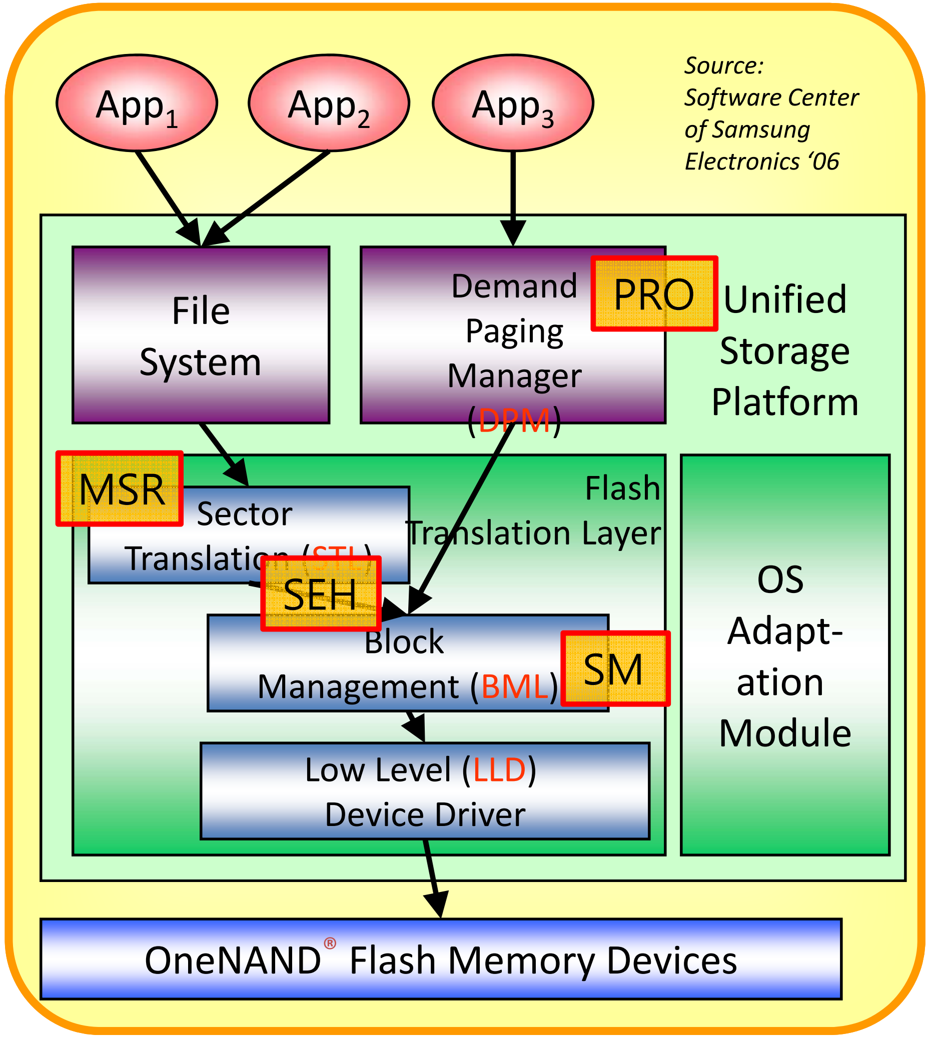
- Provable Software Lab @ KAIST has applied various formal verification technologies to the Unified Storage Platform code for the Samsung OneNAND™ flash memory
  - Conventional model checking: NuSMV and Spin [Spin 08]
  - Software model checking: C-Bounded Model Checker [ASE 08]
- In this talk, yet another 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
  - Analysis of the Symbolic Path Formulas
  - Lessons Learned
- Conclusion

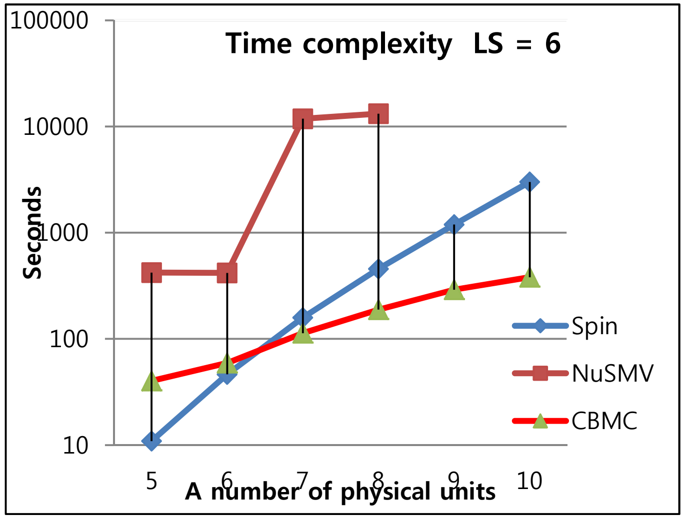
Overview of the Unified Storage Platform

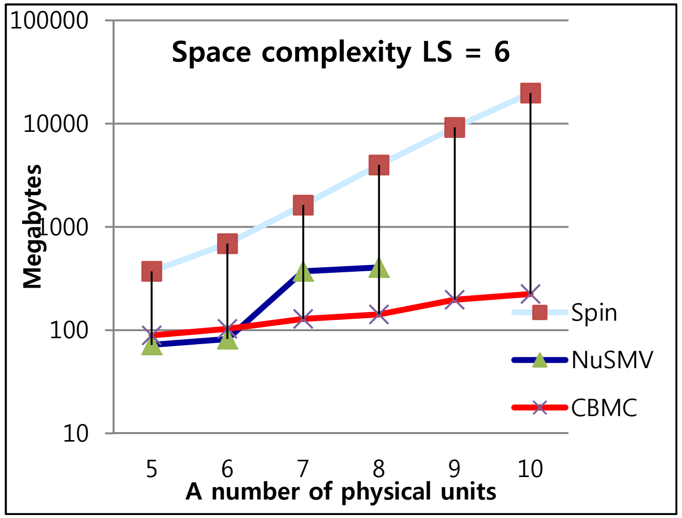
- Characteristics of OneNAND® flash
  - Each memory cell can be written limited number of times only
    - Logical-to-physical sector mapping
    - Bad block management
    - Wear-leveling
  - XIP by emulating NOR interface through demand-paging scheme
    - Multiple processes access the device concurrently
    - 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 (1/2)

- 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 (2/2)

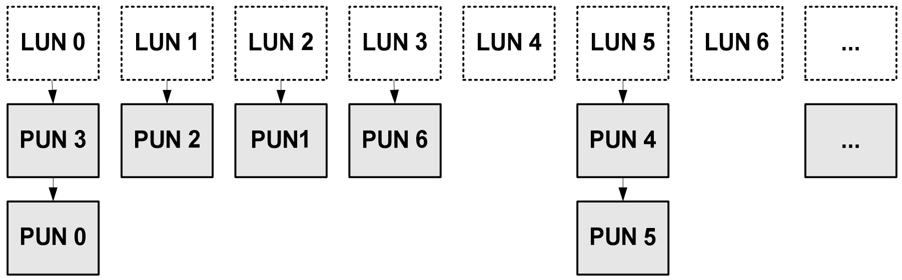
- However, we are still limited to miniature world (~10 PUNs) for the complete analysis. Thus, we may try
  - Theorem proving without bound (WHY approach)
  - Testing
    - Applying concolic testing aiming for high coverage and better scalability

Part II: Concolic testing experiments on MSR

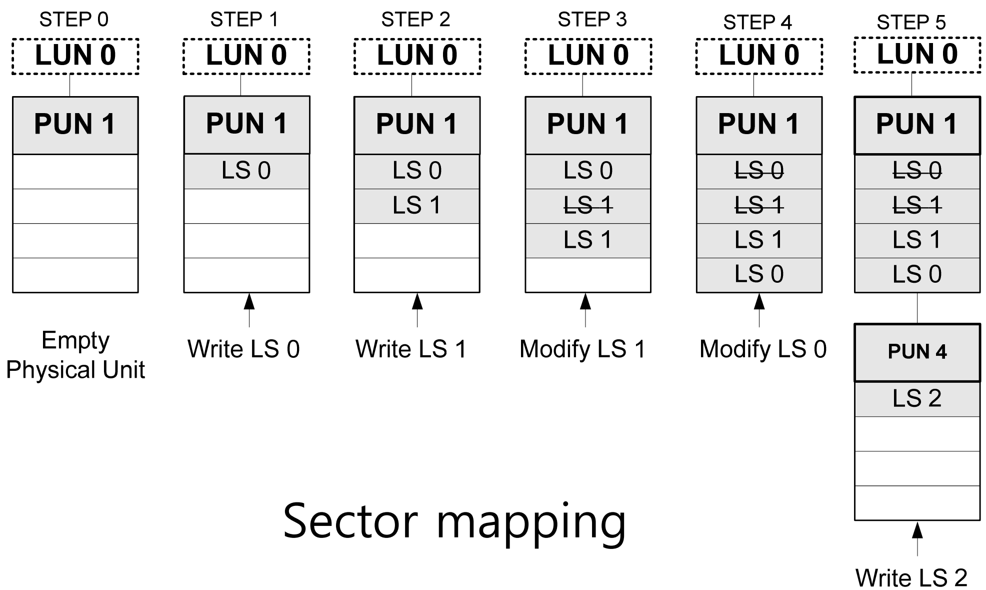
Concolic (CONCrete + symbOLIC) Testing

- 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

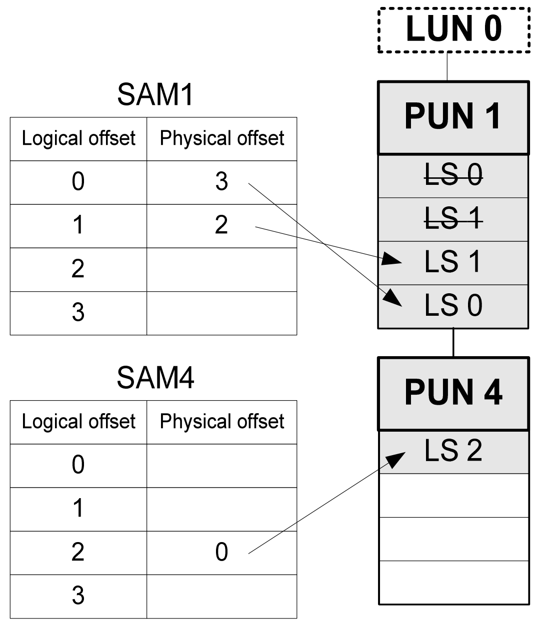
Logical to Physical Sector Mapping



1:N mapping from a LUN to PUNs



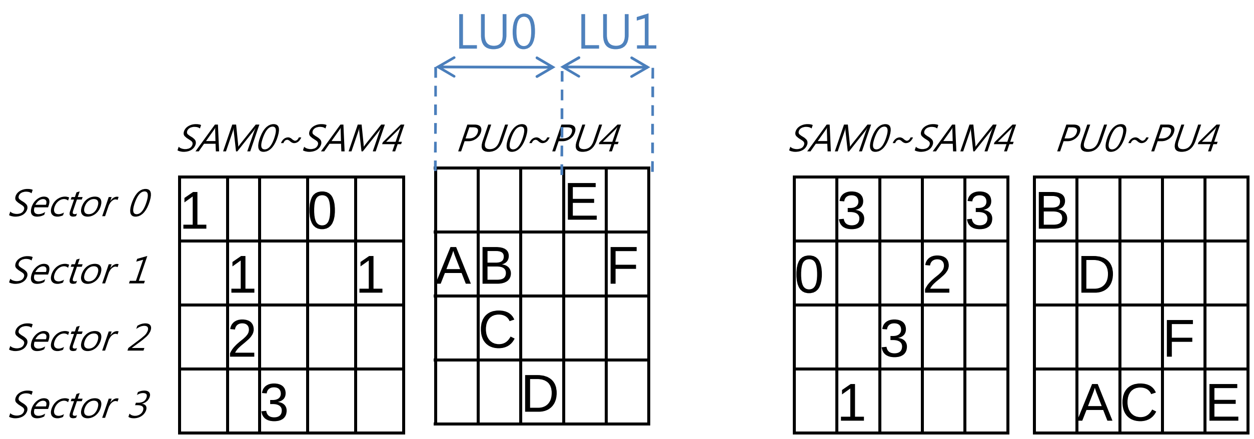
Sector mapping



Sector Allocation Map (SAM)

- 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

Loop Structure of MSR

```
curLU = LU0;
while(numScts > 0 ) {
    readScts = # of sectors to read in the current LU
    while(readScts > 0 ) {
        curPU = LU->firstPU;
        while(curPU != NULL ) {
            while(...) {
                conScts = # of consecutive PS's to read in curPU
                offset = the starting offset of these consecutive PS's in curPU
            }
            BML_READ(curPU, offset, conScts);
            readScts = readScts - conScts;
            curPU = curPU->next;
        }
    }
    curLU = curLU->next;
}
```

Loop1: iterates over LUs until all data are read

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

- 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

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:*

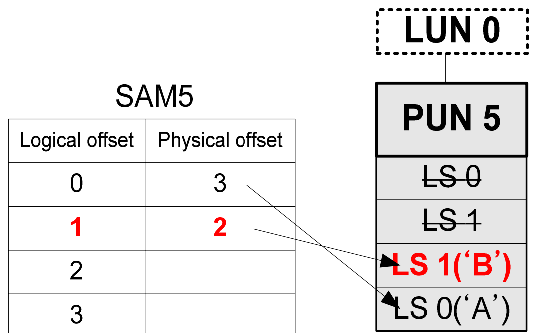
       If the *i* th LS is written in the *k* th sector of the *j* th PU, then the *i* th offset of the *j* th SAM is valid and indicates the k'th PS ,

       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

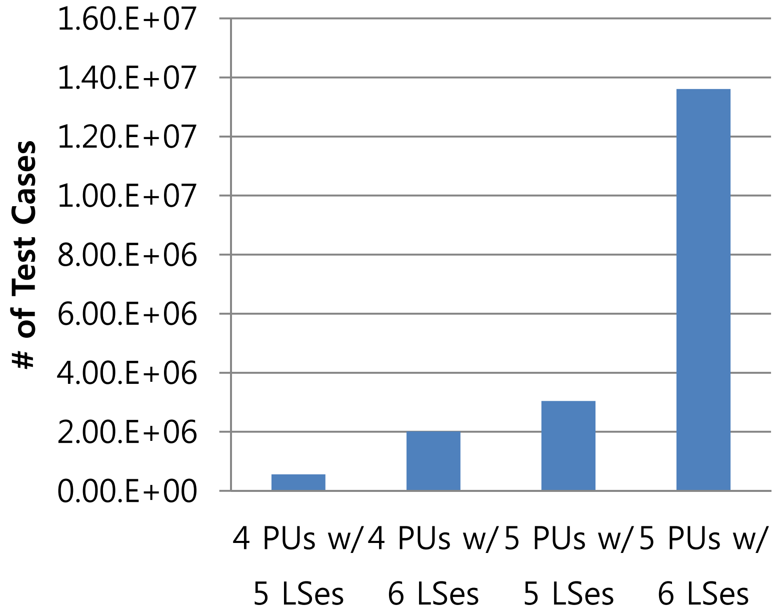
Constraint-based Environment Model

- 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);

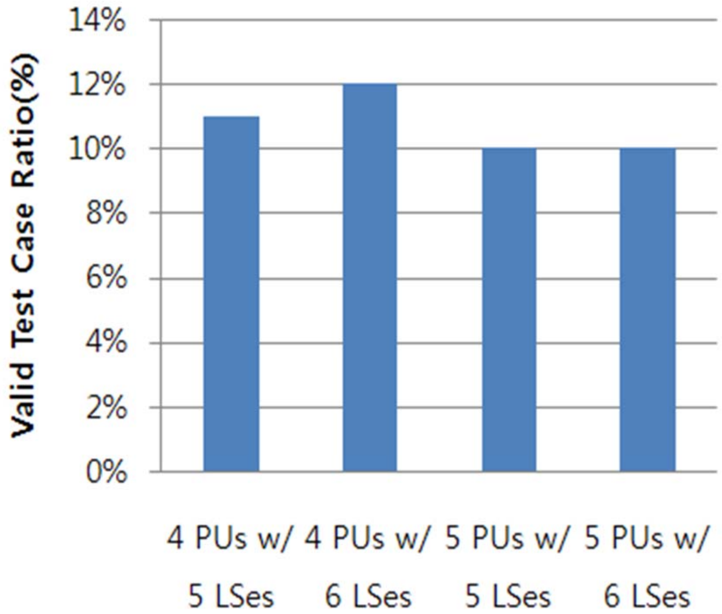
```
for (i=0; i<NUM_PUN; i++){ for (j=0; j<SECT_PER_U; j++){
  CREST_unsigned_char(pun[i].sect[j]);
  CREST_unsigned_char(SAM[i].offset[j]); } }

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; }
            ...
```

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

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



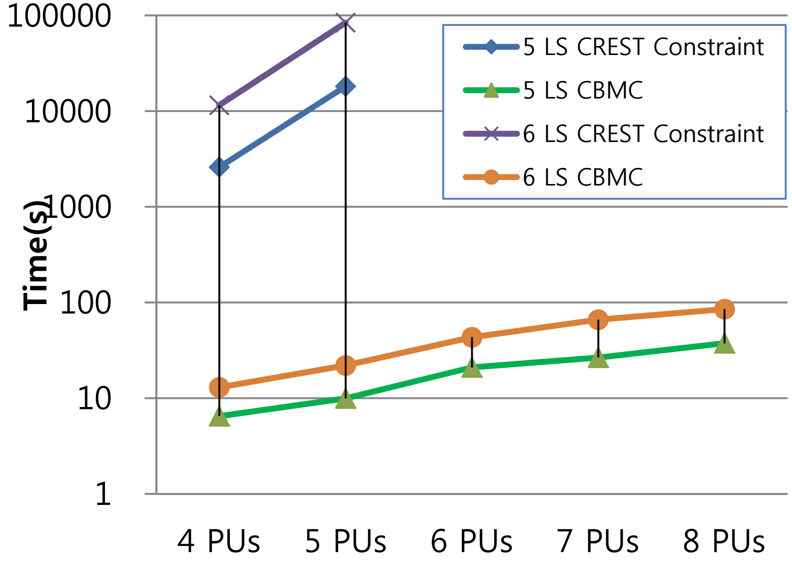
(a) Total number of test cases generated



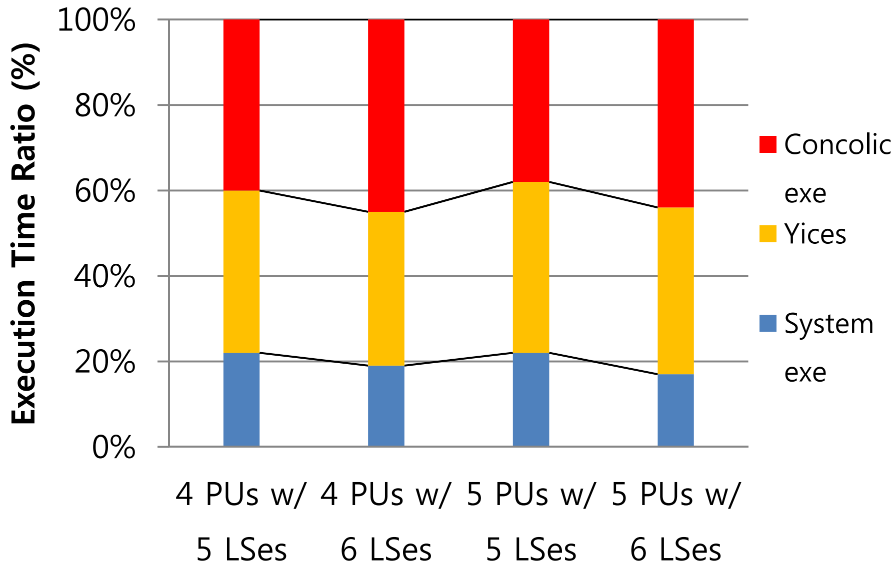
(b) Ratio of valid test cases/all test cases

- 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

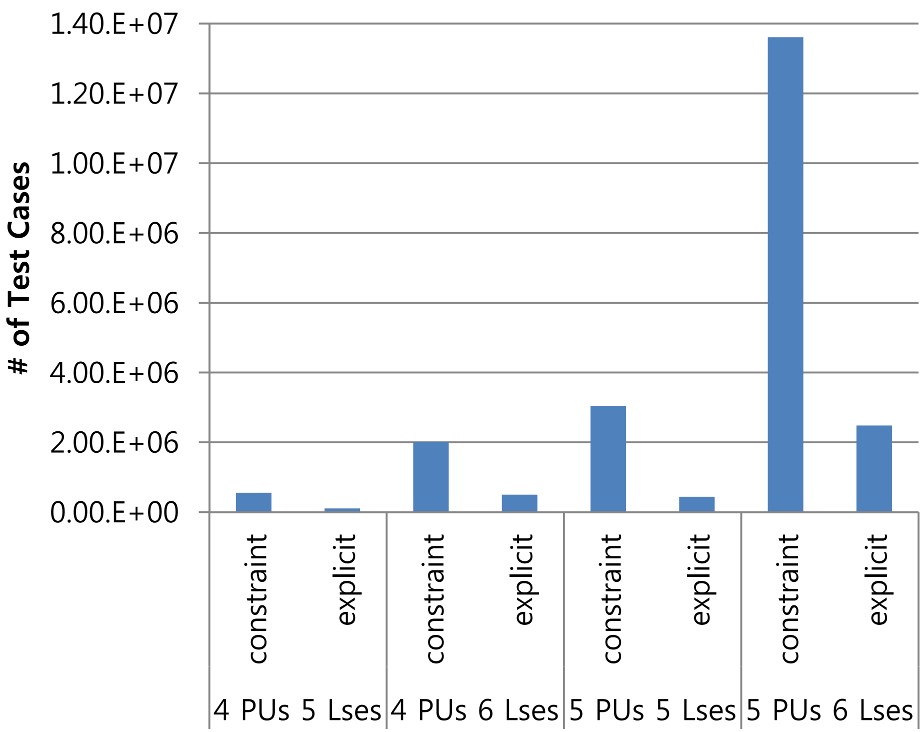
- Concolic testing is order of magnitude slower than CBMC
  - Concolic execution, SMT solving, system execution (i.e process fork and release) constitutes the overall overhead
  - Particularly, numerous invalid test cases (~90% of all test cases) worsen the performance

Explicit Environment Model

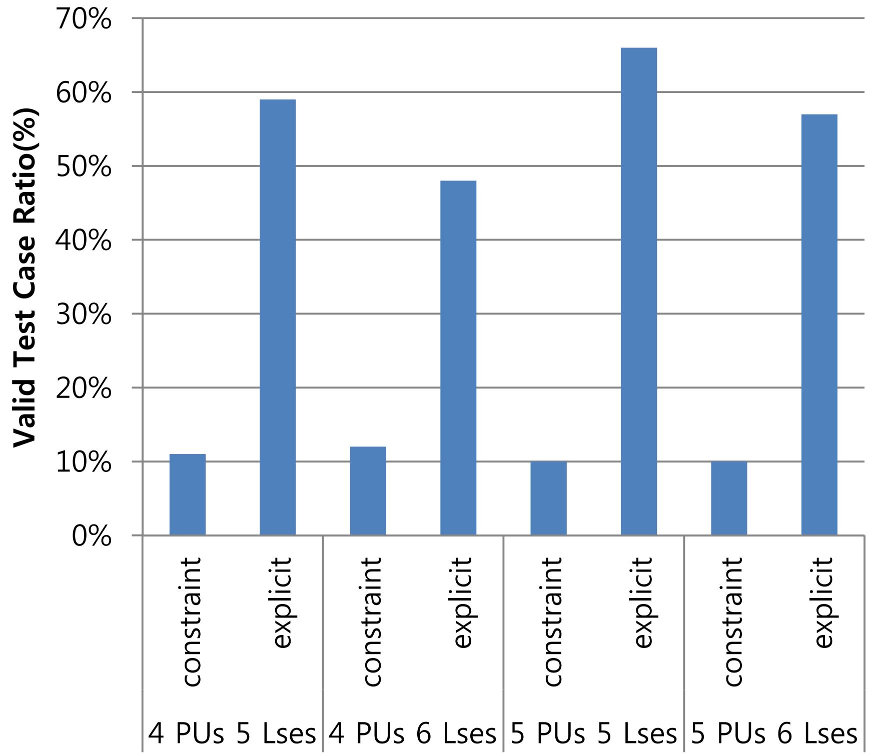
- Explicit environment model writes data to physical sectors explicitly
  - Thus , creating invalid test cases much less than the constraint-based model
- Test input variables
  - idxPU and idxSect for each logical data
- CREST has a limitation on array index variable
  - We should expand array index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

```
for (i=0; i< NUM_LS; i++){
  unsigned char idxPU, idxSect;
  CREST_unsigned_char(idxPU);
  CREST_unsigned_char(idxSect);
...
// The switch statements encode the following statements:
// PU[idxPu].sect[idxSect]= LS[i];
// SAM[idxPu].sect[i]= idxSect;
switch(idxPU){
   case 0: switch(idxSect) {
             case 0: PU[0].sect[0] = LS[i];
                     SAM[0].offset[i] = idxSect; break;
             case 1: PU[idxPU].sect[1] = LS[i];
                     SAM[0].offset[i] = idxSect; break;
                ... }
              break;
case 1: switch(idxSect) {
```

Result w/ Explicit Environment Model (1/2)



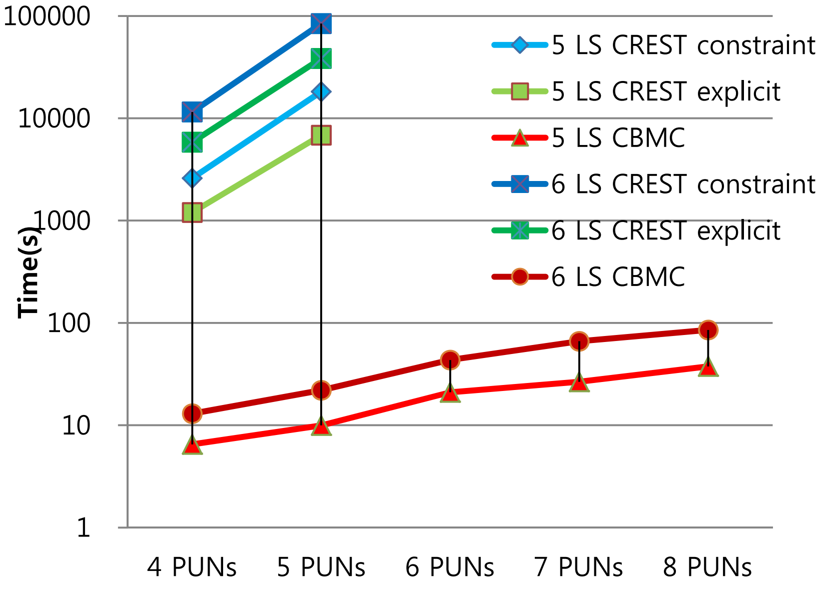
(a) Total number of test cases generated



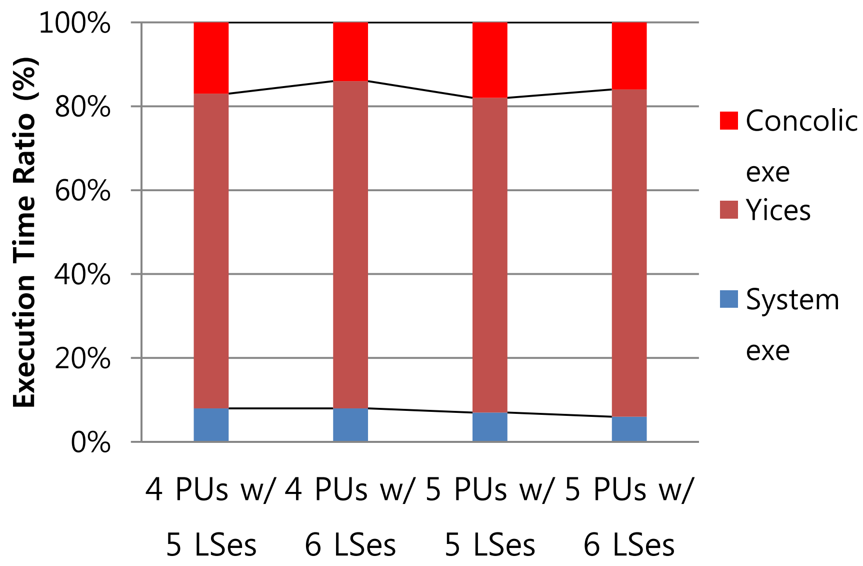
(b) Ratio of valid test cases/all test cases

- ~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

Analysis of the Symbolic Path Formulas

- Background on the SMT path formulas generated by CREST
- Path formula reduction techniques of CREST
- Statistics on the path formulas

Background on the SMT path formulas generated by CREST

- A symbolic path formula ϕ' generated by CREST is a conjunction of atomic clauses $c1, c2, ...cn$ (i.e., path conditions without boolean connectives)
  - CREST transforms a target C program P into a canonical form P'
- ϕ' is a conjunction of 8 path conditions
  - x3 at line 1 is a symbolic variable name for idxSect which indicates an offset of a physical sector containing the first logical sector (i.e., 'A')
  - Line 2 and line 3 specify that idxSect is an 8 bit unsigned integer
  - Line 4 (i.e., x3<4) indicates idxSect should be less than a number of sectors per unit (4 in our experiments).
  - Line 5 to line 8 (x3≠0, x3≠1, x3≠2, and x3=3) correspond to the switch statements which test the value of idxSect.
  - Finally, line 9 is a negated path condition and it indicates that idxSect contains an invalid value (i.e., x3=255), which is clearly not true.
    - Since Yices detects that ϕ' is unsatisfiable, CREST generates another path formula by negating a different path condition of ϕ'

```
1:(define x3::int)
2:(assert (>= x3 0))
3:(assert (<= x3 255))
4:(assert (< (+ -4 (* x3 1)) 0))
5:(assert (/= (+ 0 (* x3 1)) 0))
6:(assert (/= (+ -1 (* x3 1)) 0))
7:(assert (/= (+ -2 (* x3 1)) 0))
8:(assert (= (+ -3 (* x3 1)) 0))
9:(assert (= (+ -255 (* x3 1)) 0))
```

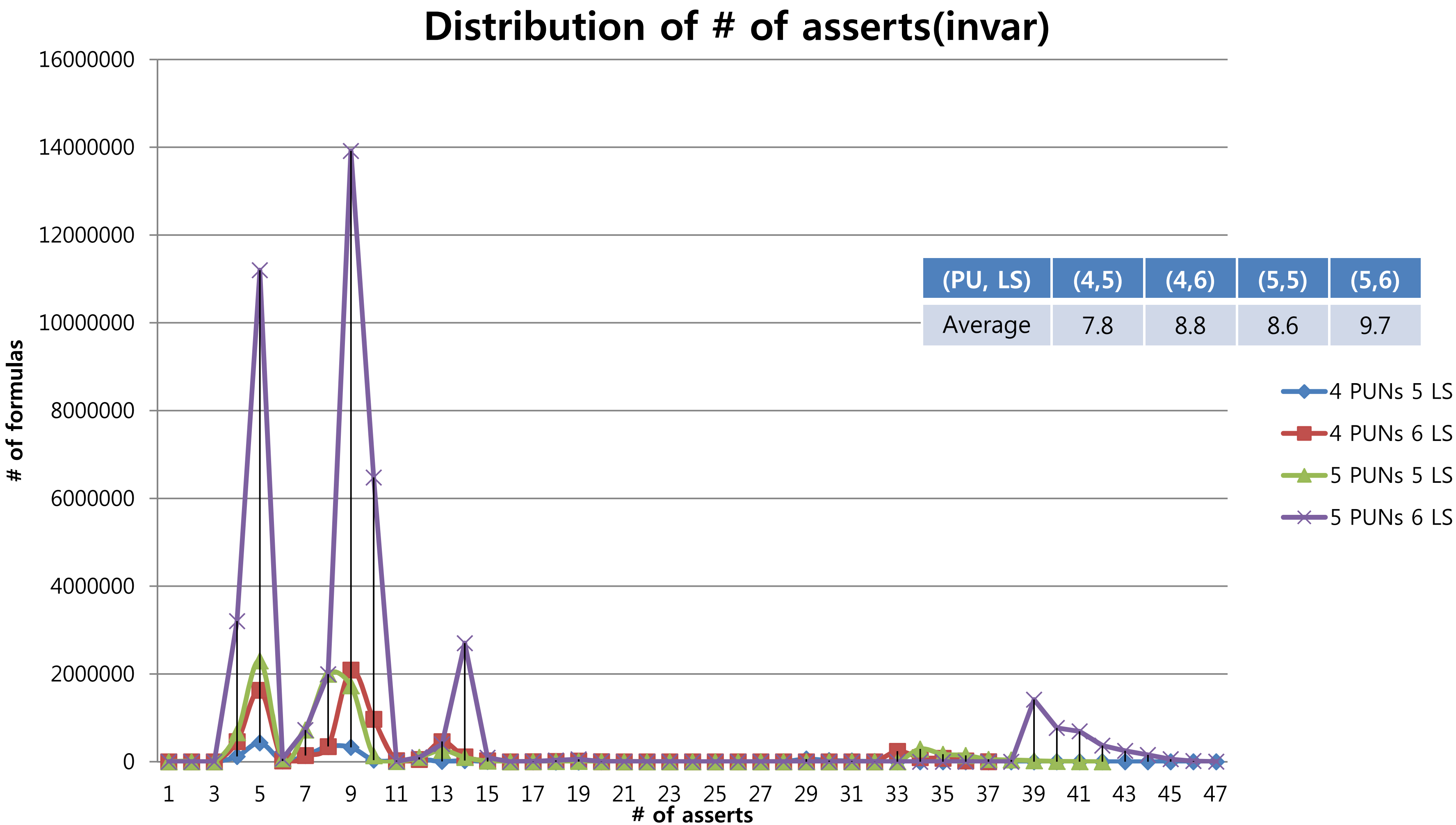
Path Formula Reduction Techniques (1/2)

- *Syntactic contradiction check*:
  - Given a generated path formula ϕ' : $c1 \wedge \ldots \wedge \neg cn$ with a negated path condition $\neg cn$, CREST checks whether there exists ci which is syntactically identical to cn (i.e., ϕ' is unsatisfiable because ci is contradictory to $\neg cn$).
  - For example, given a ϕ': $x = 0 \wedge \ldots \wedge x \neq 0$ with $x \neq 0$ as $\neg c_n$, CREST detects that ϕ' is unsatisfiable because $c_1(x = 0)$ is identical to $c_n(x = 0)$ and removes ϕ'.

Path Formula Reduction Techniques (2/2)

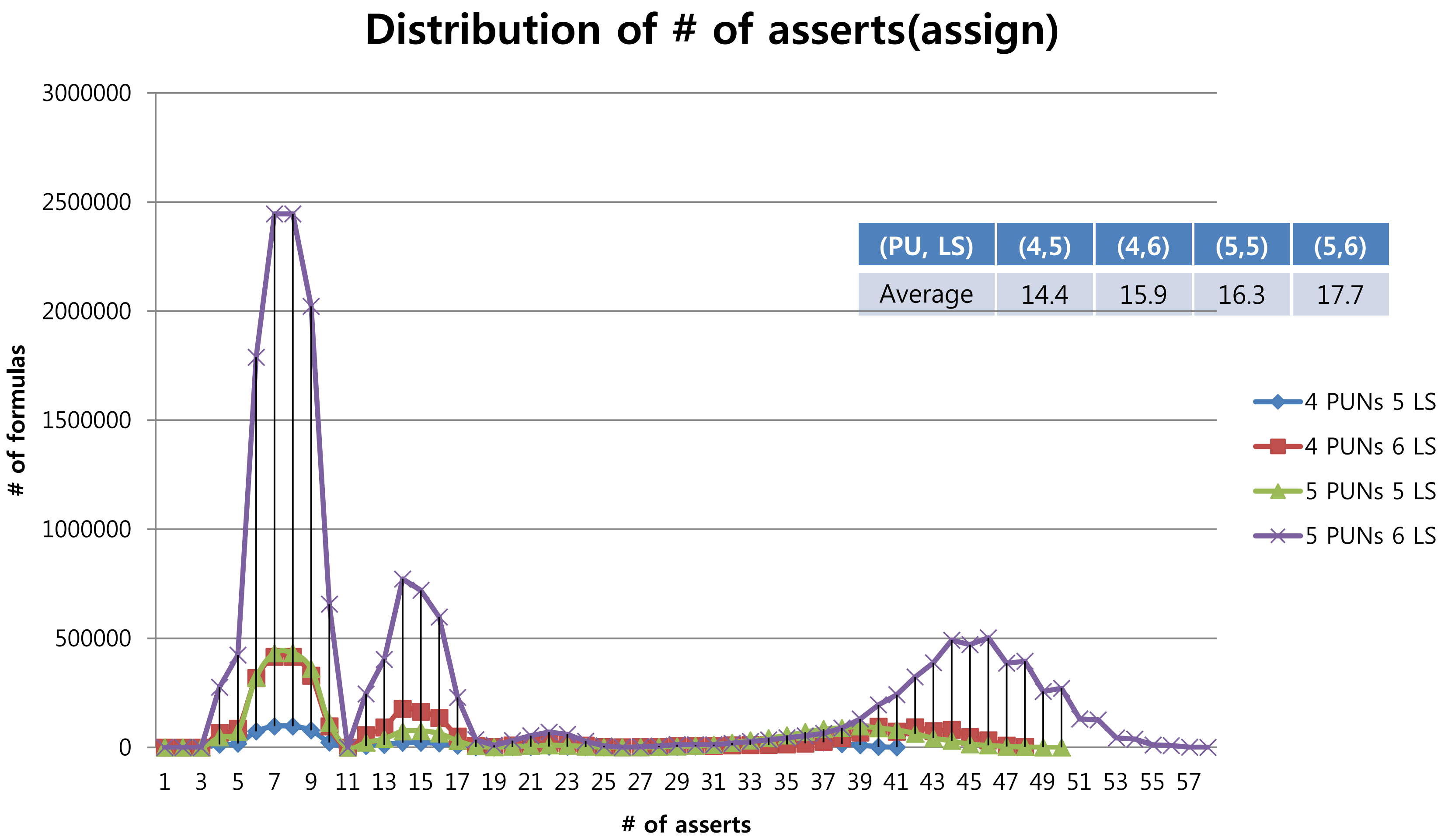
- *Slicing for the negated path condition*:
  - Suppose that *cj* of ϕ is to be negated to generate ϕ'.
  - Then, ϕ' consists of $\neg cj$ and *only* path conditions of ϕ which are dependent on *cj* through variables in terms of satisfiability.
  - CREST invokes Yices on this simplified ϕ' and get a solution for those variables.
    - Thus, the next input values are the same as the previous input values except the variables in the solution.
    - Note that this technique utilizes the fact that path formulas share many path conditions in common
  - For example, given $\phi : a < b \wedge c < d \wedge d < e \wedge e < f$ with $e < f$ as a path condition to negate, CREST generates $\phi' : c < d \wedge d < e \wedge \neg(e < f)$ without $a < b$
    - since $a < b$ is not dependent on *e* nor *f*.

Symbolic Path Formula statistics



(PU, LS)	(4,5)	(4,6)	(5,5)	(5,6)
Average	7.8	8.8	8.6	9.7

Symbolic Path Formula statistics



(PU, LS)	(4,5)	(4,6)	(5,5)	(5,6)
Average	14.4	15.9	16.3	17.7

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 2909]