Model Checking for Practical C software Analysis – Experience Reports

Moonzoo Kim

Provable Software Lab. CS Dept. KAIST

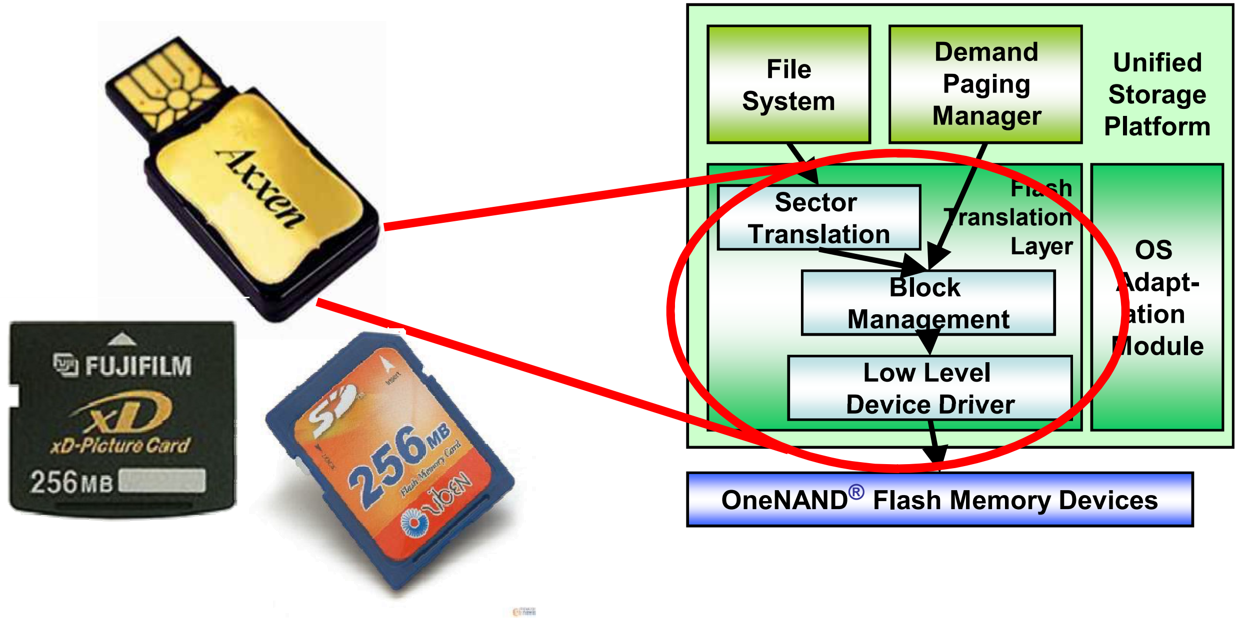
Http://pswlab.kaist.ac.kr

Prelude

Professional Cook

Knife Master

Summary of the Talk



- **The series of empirical studies on verification of Samsung OneNAND™ flash memory FTL through various off-the-shelf techniques**
  - Symbolic MC, Explicit MC, SAT-based MC, Exhaustive testing, randomized testing and concolic testing

Contents

- **Overview on Multi-sector Read Operation (MSR)**
  - Flash Translation Layer (FTL) scheme
  - MSR algorithm
- **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) — **Spin 08 paper**
  - Explicit model checking (Spin)
  - C-bounded model checking (CBMC) — **ASE 08 paper**
- **Exhaustive testing and concolic testing is applied as well**

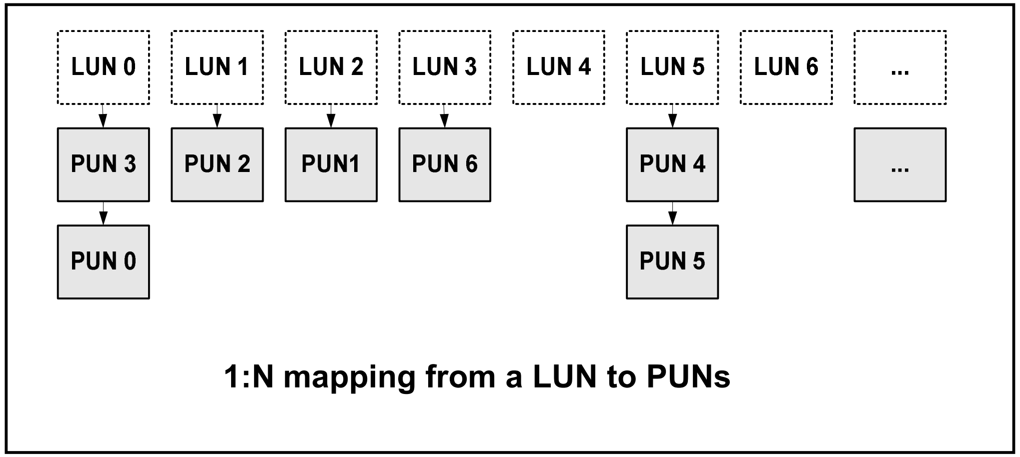
ICST 08 paper

SBMF 09 "Grand Challenge in Verified Software" invited paper

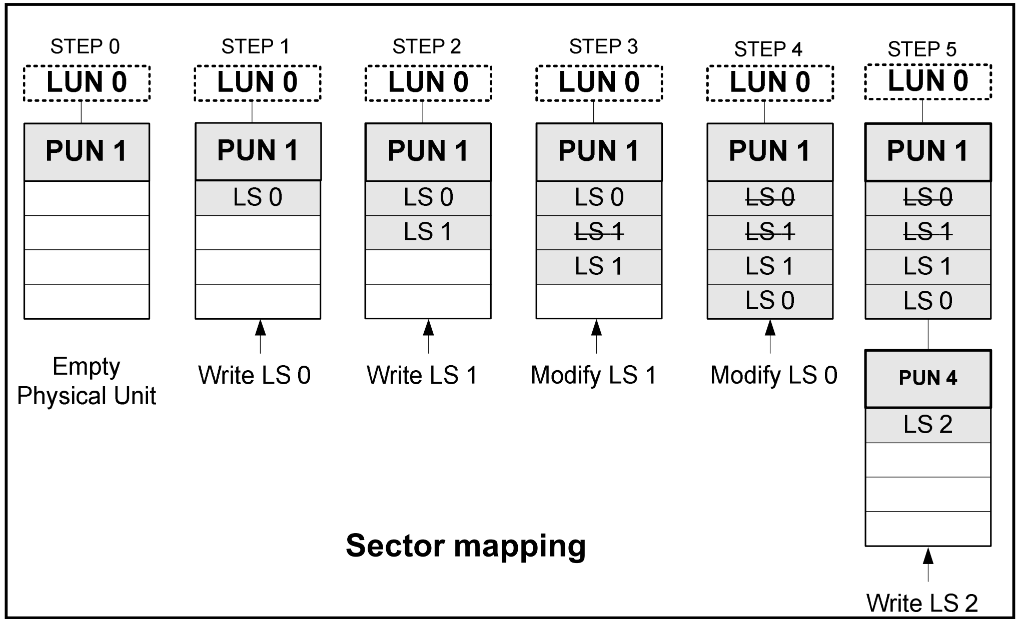
PART I: MSR Overview

- FTL basics
- Example of logical data distribution on physical unit
- Exponential increase of possible distributions
- MSR structure

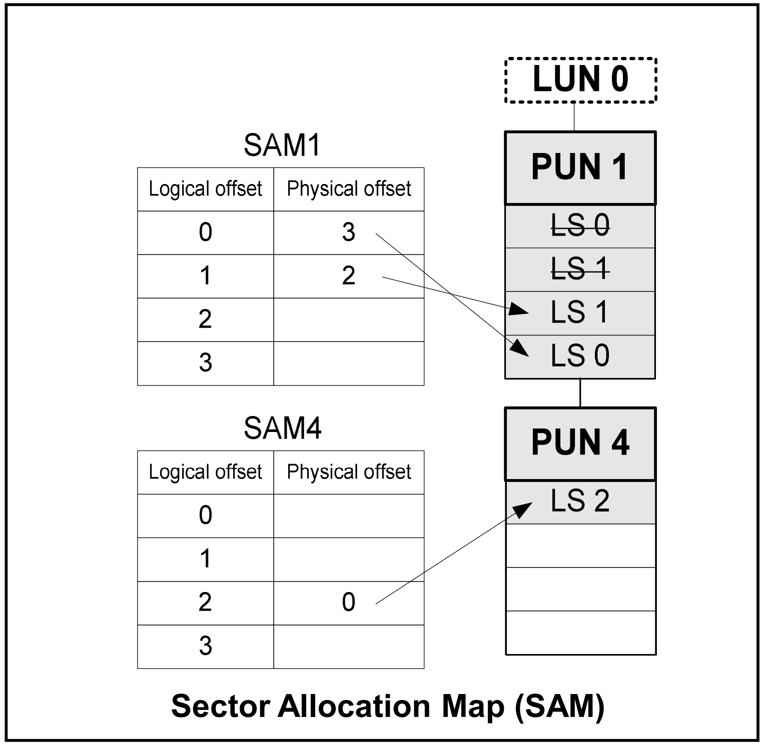
Logical to Physical Sector Mapping



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"

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

(b) Another 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

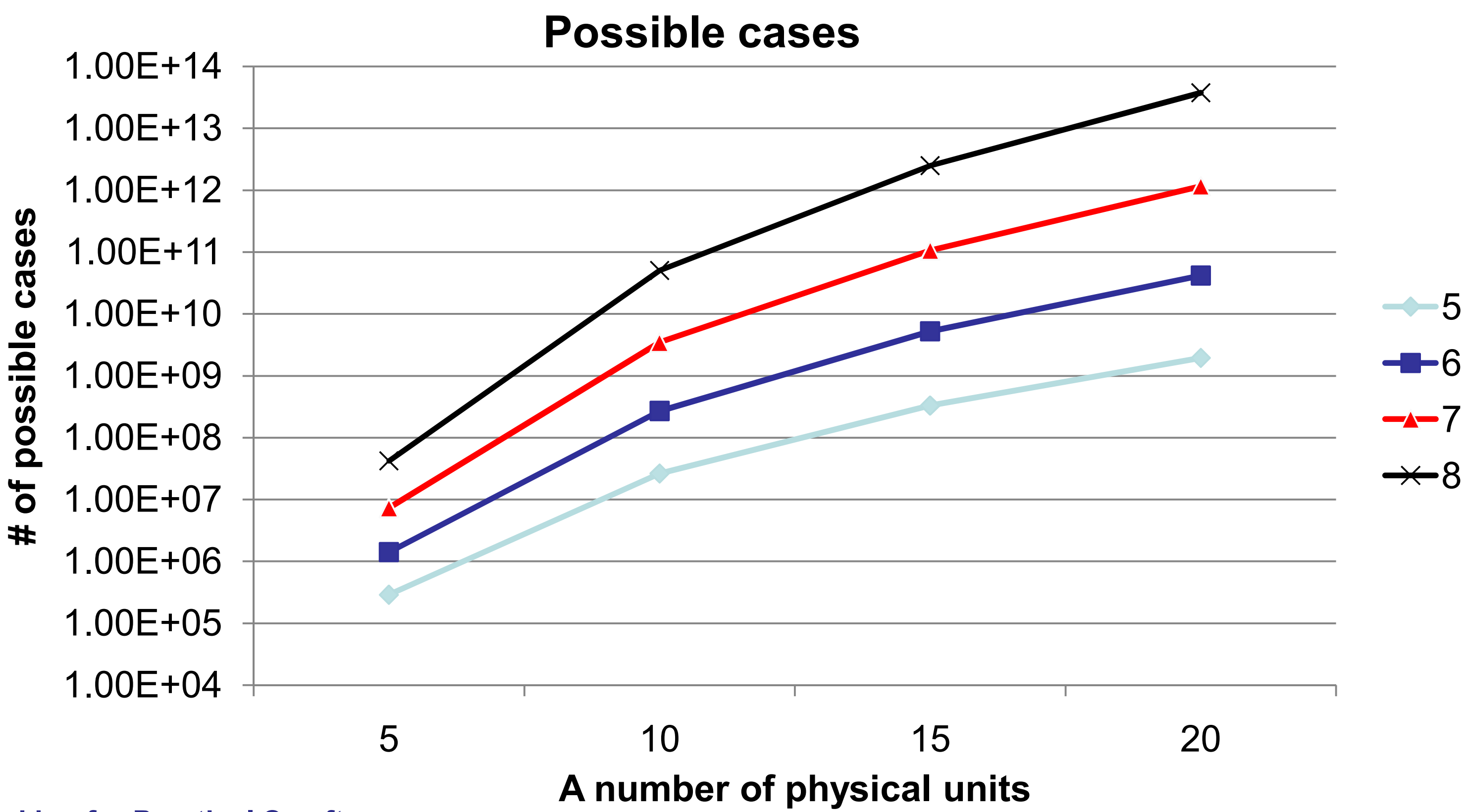
(c) A distribution of "FEDCBA"

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

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

Exponential Increase of Distribution Cases

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



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

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 Exp.

- **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**

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 software

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
2: while(x>=0){
3: y = x; ← DP2
4: x --;} ← DP3
	0: DP1=0; DP2=0; DP3=0;
	1: if (!DP1) { x=x-1; DP1 =1;}
	2: if ((DP1 || DP3) && x>=0) {
	y = x; DP2=1; DP3=0;
	}
	3: if (DP2) {
	x--; DP3=1; DP2=0;
	}
		init(DP1):=0; init(DP2):=0; init(DP3):=0;
		next(DP1):= 1;
		next(DP2):= case (DP1 | DP3) & (x >= 0) : 1;
		DP2 : 0;
		1 : DP2;
		esac;
		next(DP3):= case (DP1 | DP3) & (x >= 0) : 0;
		DP2 : 1;
		1 : DP3;
		esac;
		next(x):= case !DP1
			DP2 : x-1;
			1 : x;
			esac;
			next(y):= case (DP1 | DP3) & (x >= 0) : x;
			1 : y;
			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

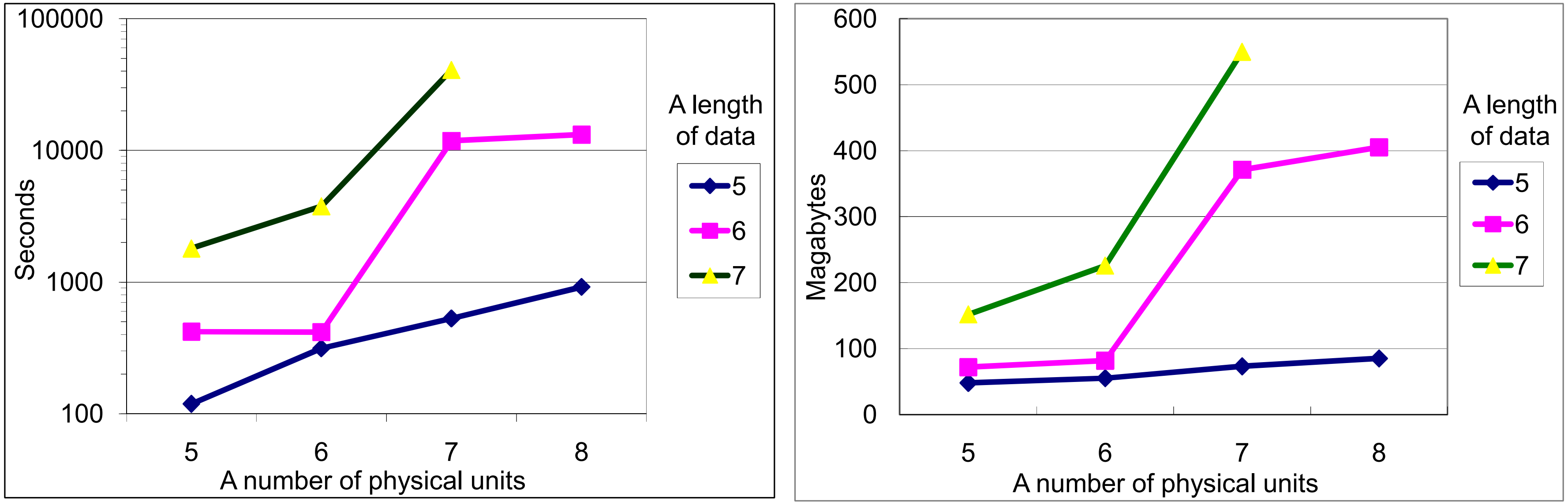
       i=3 k=3 j=2
    3. *For one LS, there exists only one PS that contains the value of the LS:*

       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.

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

	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 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 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

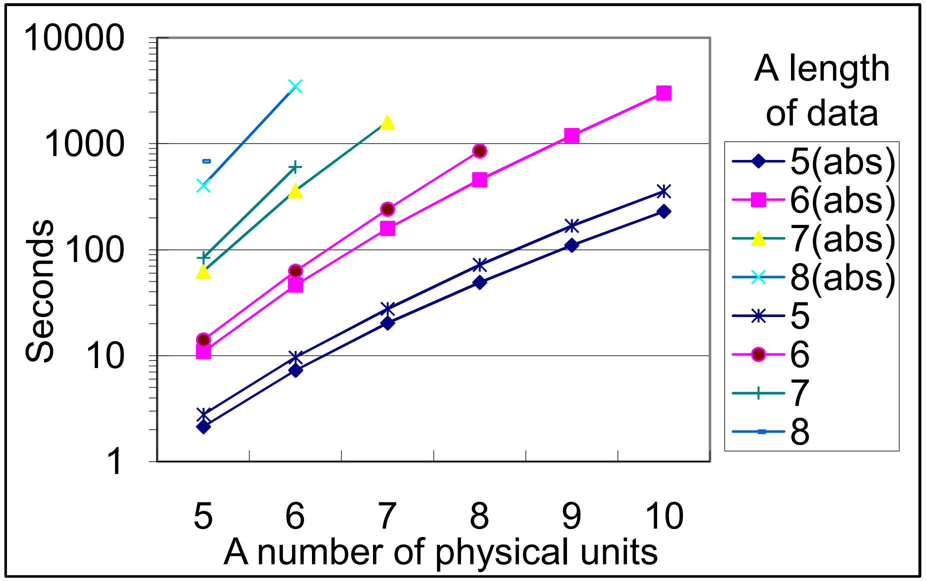
SAM0~SAM4

Sector 0	1			0	
Sector 1		1			1
Sector 2		2			
Sector 3			3		

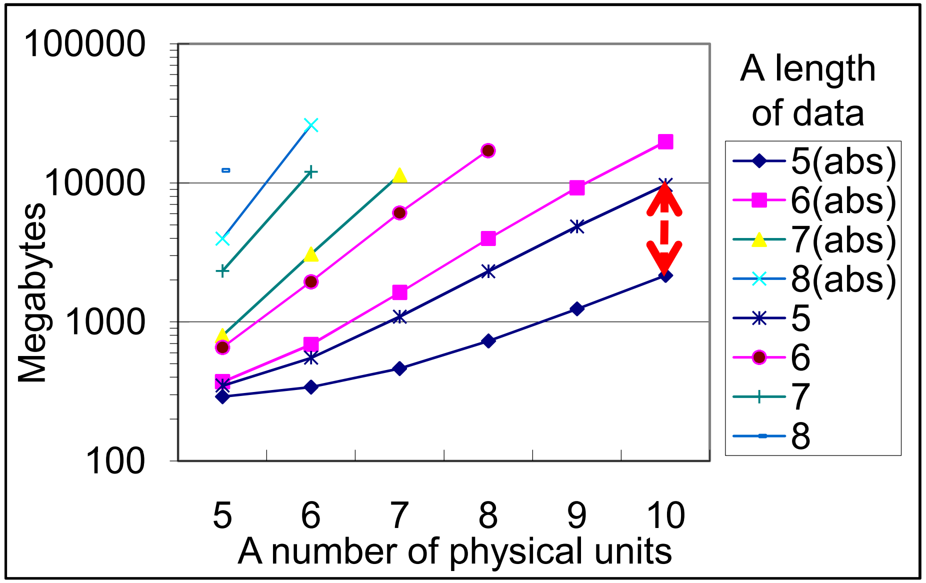
PU0~PU4

Sector 0				E	
Sector 1	A	B			F
Sector 2		C			
Sector 3			D		

Verification Performance of 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

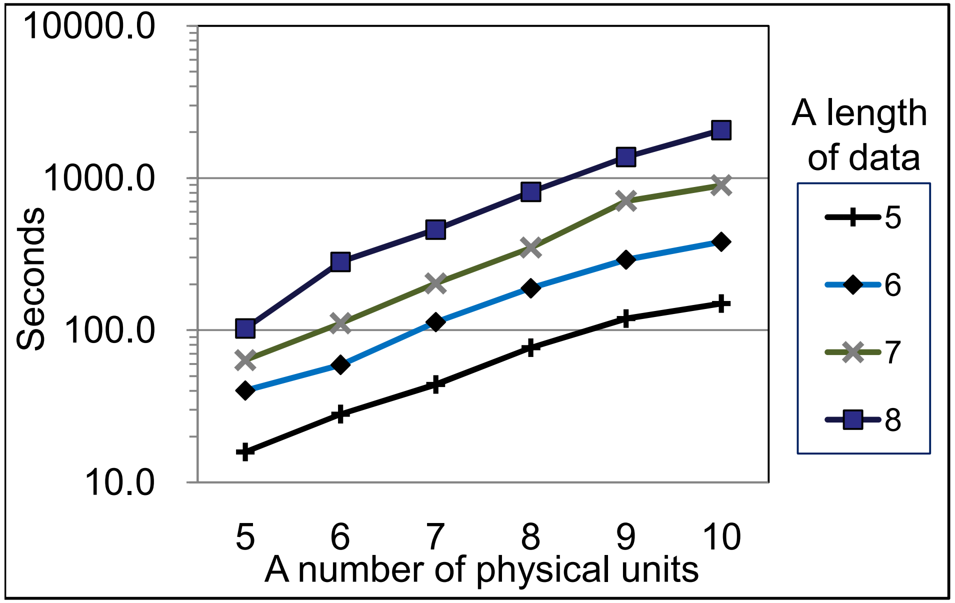
SAM0~SAM4

Sector 0	1			0	
Sector 1		1			1
Sector 2		2			
Sector 3			3		

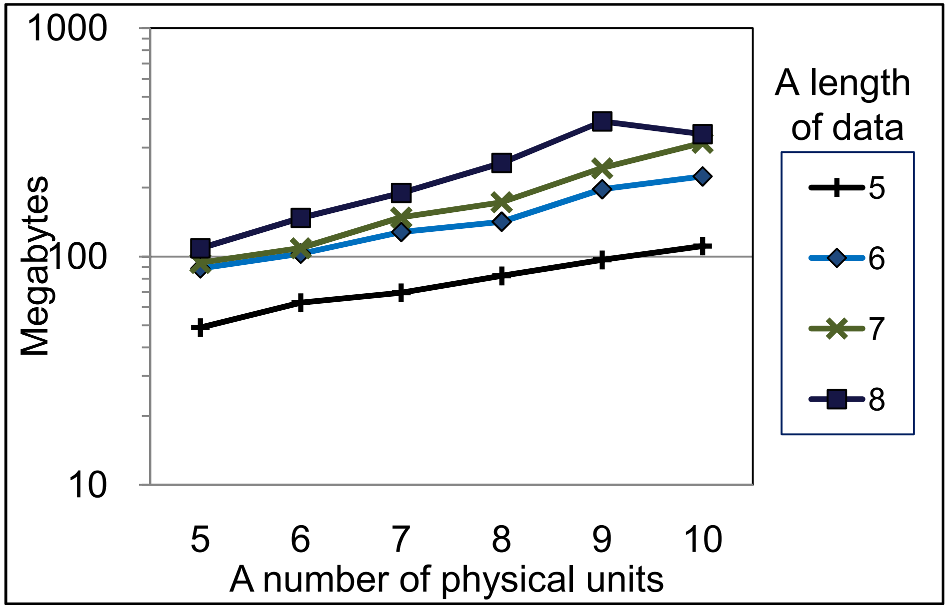
PU0~PU4

Sector 0				E	
Sector 1	A	B			F
Sector 2		C			
Sector 3			D		

Verification Performance of CBMC



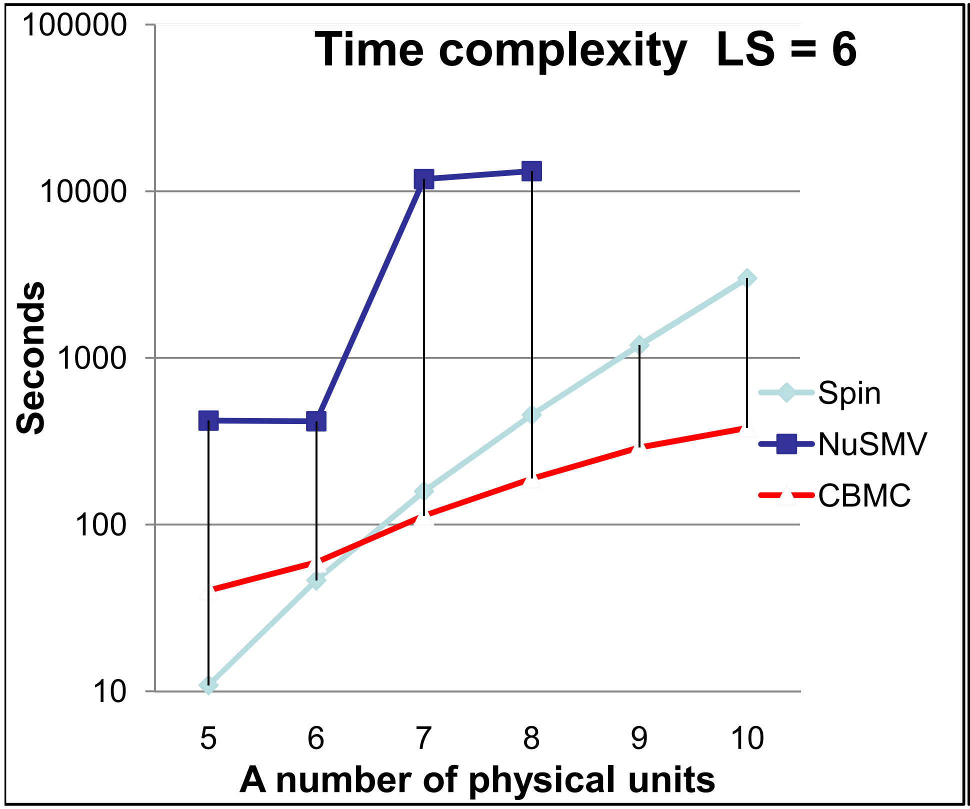
(a) Time consumption

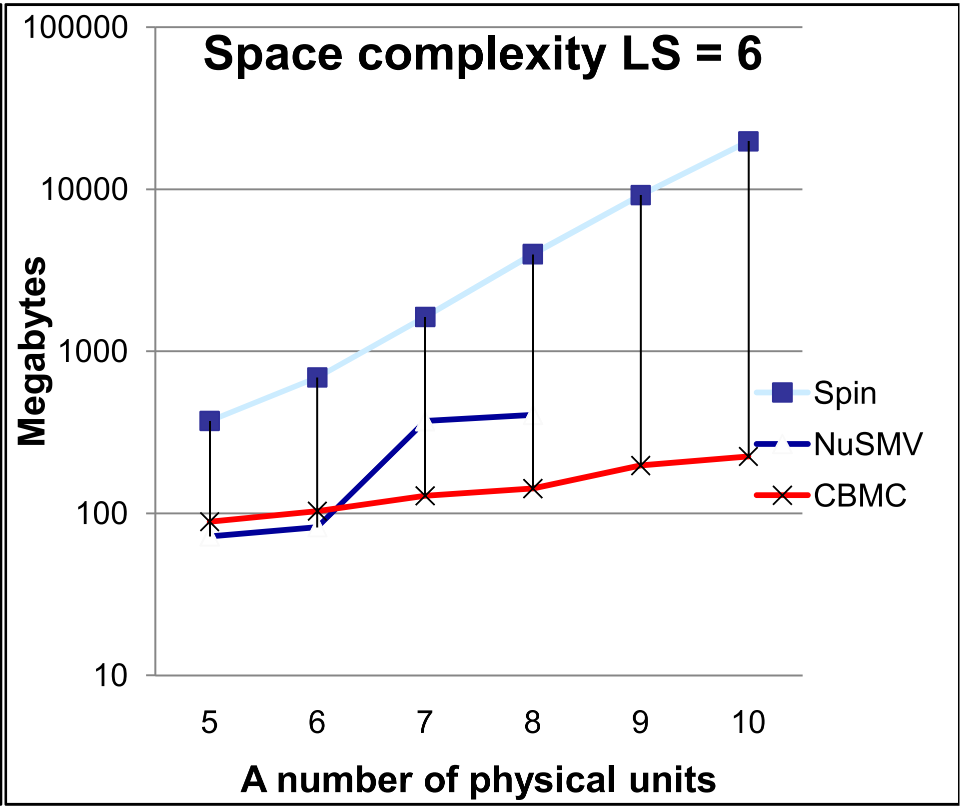


(b) Memory consumption

- 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 8.6×10^5 variables and 2.9×10^6 clauses.

Performance Comparison





Comparison of Model Checking Techniques

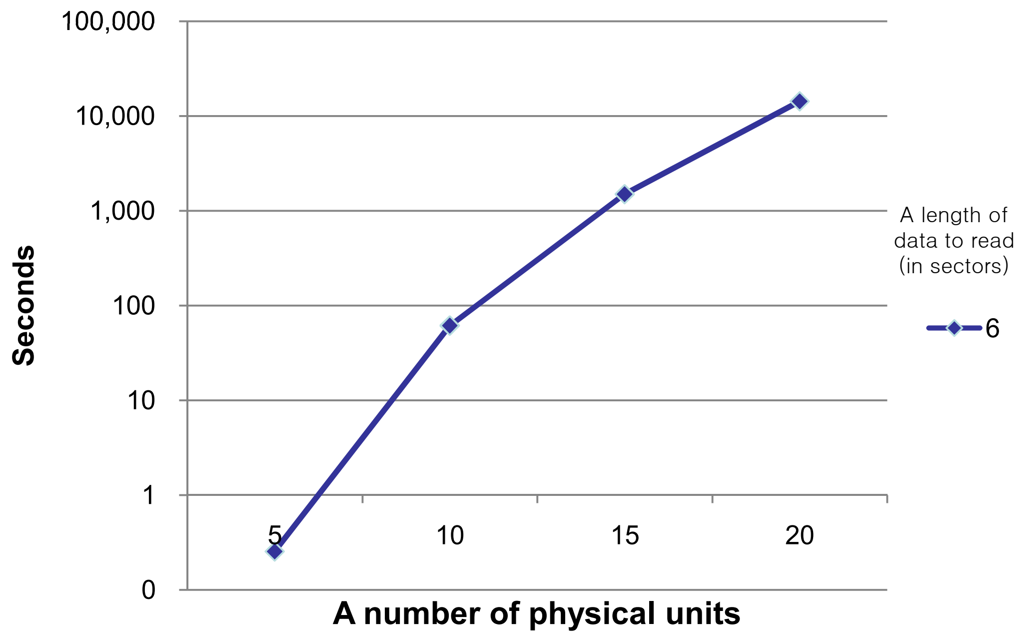
- **Application of Model Checking to Industrial SW Project**
  – Current off-the-shelf model checkers showed their effectiveness to debug a part of industrial software, if a target portion is carefully selected
  – Although model checker worked on a small scale problem, it still contributes due to its 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

Part III: Experiments on Testing MSR

Exhaustive Testing

- **Exhaustive testing on a small flash**
  - We developed a testing environment and an abstracted version of MSR(), called S_MSR()
  - The reuse of formal environment models reduces the testbed setup time
  - Exhaustive testing is roughly 6 times faster than CBMC



Randomized Testing

- **Randomized testing on a large flash**
  - Model checking and exhaustive testing cannot handle a large flash
    - We cannot find a bug on a large flash
  - We performed randomized testing on 10^{11} randomly chosen cases with 6 sectors long data distributed over 1000 PUs
    - This takes 8 hours 20 minutes
  - This test cover only $\frac{1}{3.9\times10^{11}}$ cases among all possible cases
  - 1GB flash has more than 2^{19} (a half million) physical units
    - 2^{19} units * 4 sector/unit * 512 bytes/sector

=> Randomized testing cannot provide sufficient coverage ever

Concolic (CONCrete + symbOLIC) Testing

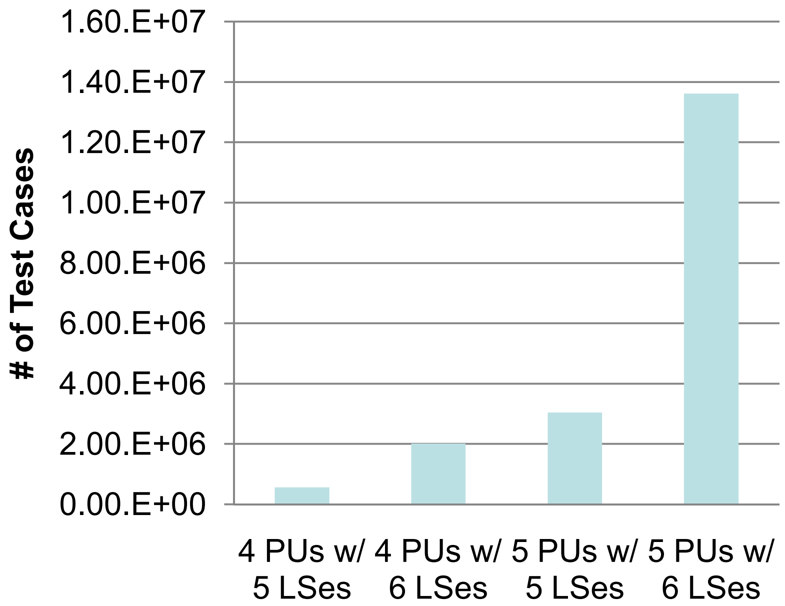
- **Automated Scalable Unit Testing of real-world C Programs**
  - Execute unit under test on automatically generated test inputs so that all possible execution paths are explored
    - Explicit path model checking
- **In a nutshell**
  - Use concrete execution over a concrete input to guide symbolic 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 issue like in model checking

Constraint-based Environment Model

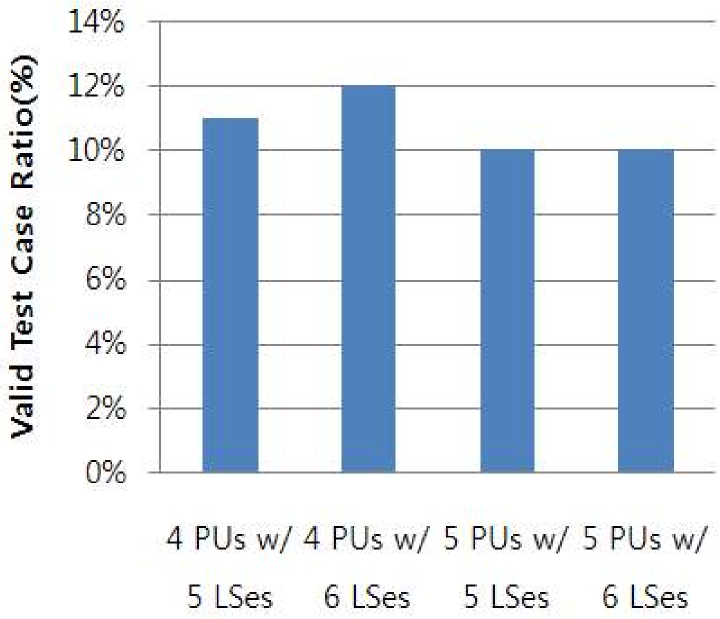
- **We have to specify symbolic variables and put constrain them**
  - If assigned input value does not satisfy the constraints (i.e. invalid test case generated), a current iteration terminates immediately without testing MSR (goto out);

```
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; }

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

Result with Constraint-based Model (1/2)

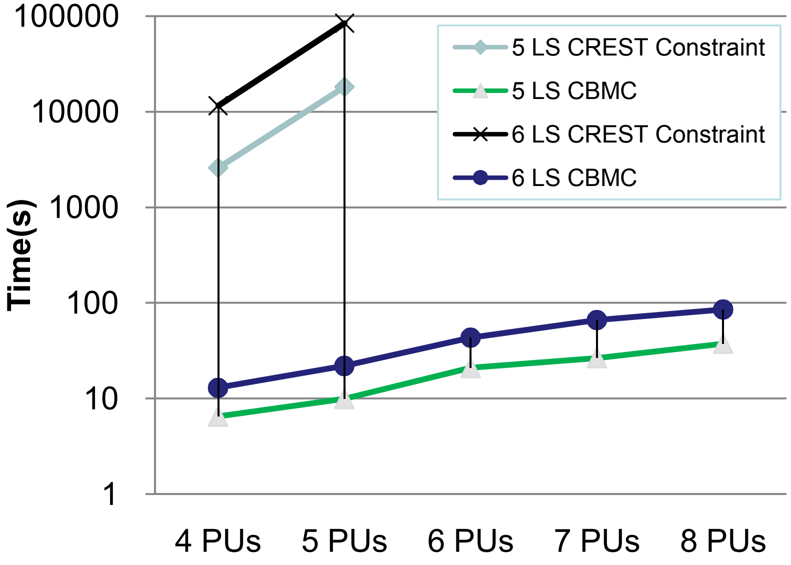


(a) Total number of test cases generated

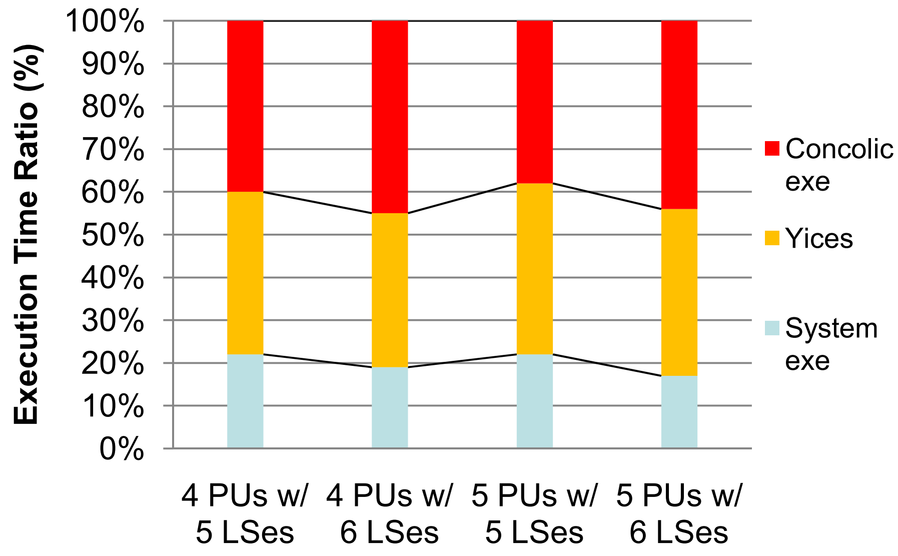


(b) Ratio of valid test cases/all test cases

Result with Constraint-based Model (2/2)



(a) Total analysis time



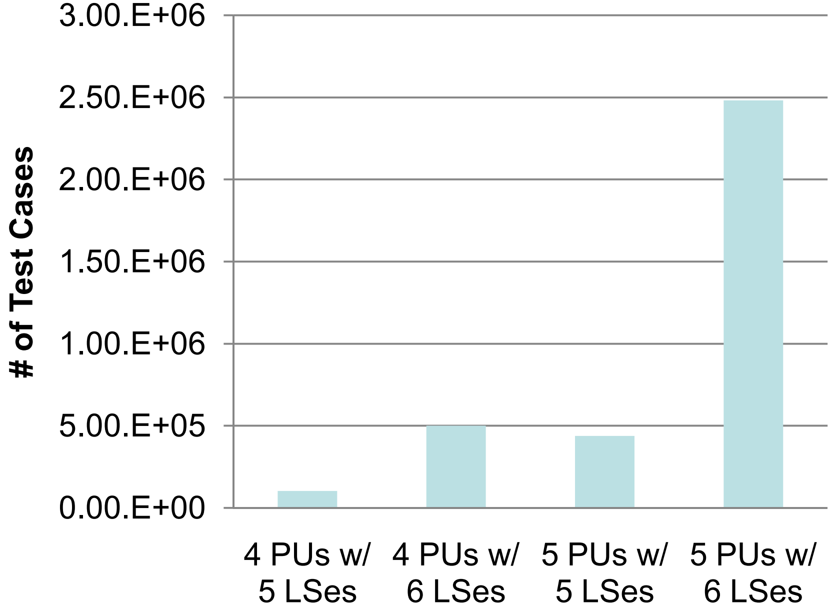
(b) Time ratio of analysis steps

Explicit Environment Model

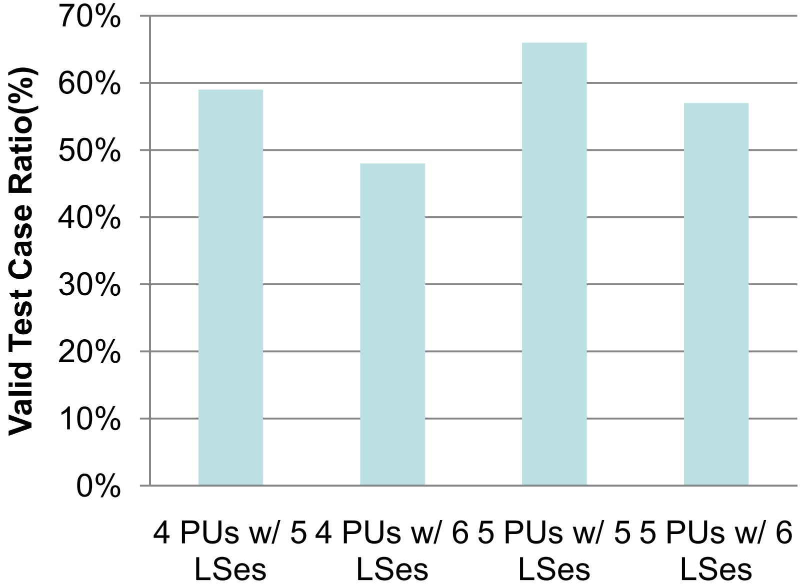
- Explicit environment model create invalid test cases much less than the constraint-based model
- CREST has a limitation on array index variable

```
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 two 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 with Explicit Model (1/2)

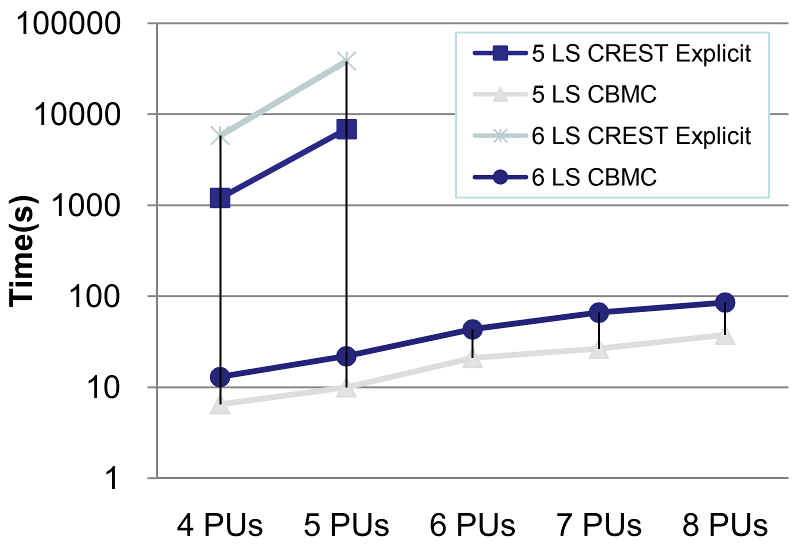


(a) Total number of test cases generated

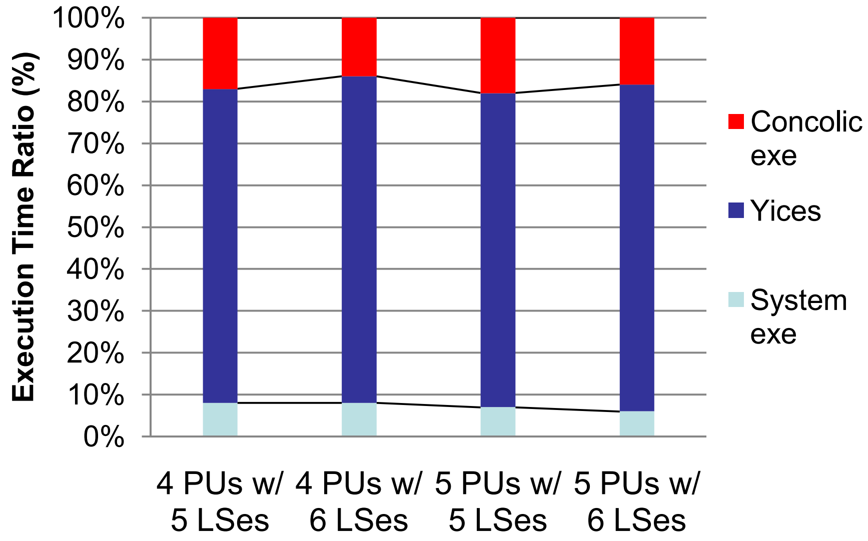


(b) Ratio of valid test cases/all test cases

Result with Explicit Model (2/2)



(a) Total analysis time



(b) Time ratio of analysis steps

Overall Observations

- **There are multiple useful off-the-shelf analysis tool to improve the reliability of target C programs in practice**
  – Knowing characteristics of them and underlying mechanisms is essential for successful analysis
- **Systematic heuristics techniques for searching "XXX" space are important**
  – Good tradeoff between completeness and effectiveness
- **Abstract environment modeling is very important for in-depth target system analysis**
  – This area still largely relies on human expertise

Lessons Learned

- **Necessity of Benchmarks for the purpose of SW analysis**
  - To encourage comparative studies of various analysis methods
- **Importance of target application selection**
  - Several restrictions from industrial partner
  - Open source target application
- **SMT techniques have large rooms for improvement**
  - Pos: You can join the competition now !!!
  - Cons: You may better to use other analysis engine, at least in a few years