Increasing Confidence in Proper Execution through Invariant Checking

Kim, Miryung

Petersen, Andrew Kinoshita

Outline

- Motivation
- Invariants
- Related Work
- Overview of a Self Verifying Architecture
- Experiment
- Current Results
- Future Work

Motivation : why does this need to be done?

- Processors make errors.
  - The size of processors is increasing.
  - The distance between components is decreasing.
  - Thus, the probability of transient errors occurring during execution is increasing.
- Are we OK with this trend towards higher error rates? – NO!
  - For safety-critical and high reliability systems, errors are not acceptable.
  - For simulations that take days or weeks to complete, this is simply not tolerable.

Motivation : how should this be done?

- What cost are we willing to pay?
  - Increased hardware complexity or redundant hardware?
  - Static analysis to prove correctness?
- Hardware Solutions
  - Hardware is already very complex.
  - Proving the correctness of redundant hardware may be just as difficult a problem.
- Software Solutions
  - Developing “proofs” of correctness is not possible...and economically infeasible.
  - Static techniques don’t work well on run-time problems.

Invariants : what are they?

in·var·i·ant *adj.*

1. Not varying; constant.
2. *Mathematics.* Unaffected by a designated operation, as a transformation of coordinates.

n.

An invariant quantity, function, configuration, or system.

(Dictionary.com, February 25, 2002)

Invariants : why use them?

- Programmers are already using them (implicitly) during development. (e.g. ASSERT)
- An invariant is what a programmer wants to guarantee at a certain point in program execution.
- Thus, invariants are used for program verification, code documentation, test suite validation, etc.
- Why wouldn’t they be applicable in dynamic verification?

Invariants : how do we find them?

Manual static analysis:

- Hoare Triples
  - Pre-conditions / Post-conditions
- Loop Invariants
- The Drawback:
  - You’ll only live about 100 years...

Invariants : Static vs. Dynamic Detectors

- Static Invariant Detector
  - “Houdini” performs static analysis and suggest candidate invariants.
  - “ESC/JAVA” analyzes the code and proves the correctness of asserted invariants.
- Dynamic Invariant Detector
  - “Daikon”, given a large test suite, runs the program and detects invariant properties, with respect to the test suite.

Invariants : what is the state of the art?

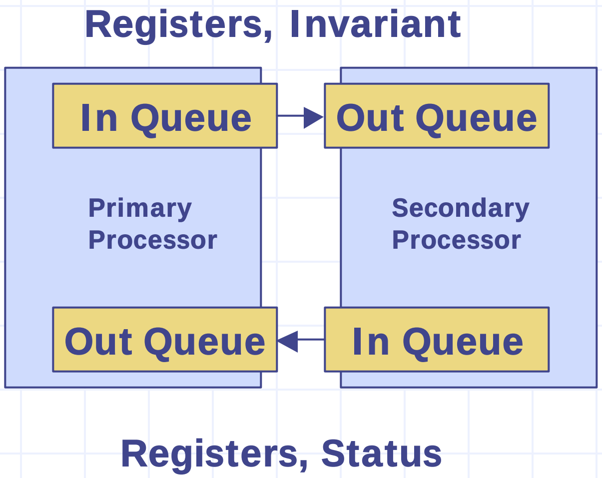
- Discovering a complete set of invariants is undecidable.
- “Daikon” looks for invariants related either to function parameters or return values.
    - Thus, “Daikon” cannot detect invariants at the statement level or within in-line macros.
- A large test suite is needed to remove data dependencies.
- Despite of these shortcomings, we chose to use Daikon.

Related Work

- Diva, *Todd M. Austin, et al* : Redundant processor
  - Verifies every instruction with a second processor
  - Hardware costs are increased
- HAT(Hardware access table)
  - Accelerates table lookup for safe pointer checking
  - Is yet another additional piece of hardware

Related Work, Continued

- Self-Verifying Architecture, *Jeong and Jamison* :
    - Derived from Necula's "proof-carrying code"
    - Features a fast primary processor and slower secondary processor
    - The secondary processor verifies groups of instructions by computing invariants inserted by the programmer.
    - Drawbacks: Additional hardware (including means for inter-processor communications). Invariant generation is time consuming.



Registers, Invariant

In Queue

Out Queue

Primary Processor

Secondary Processor

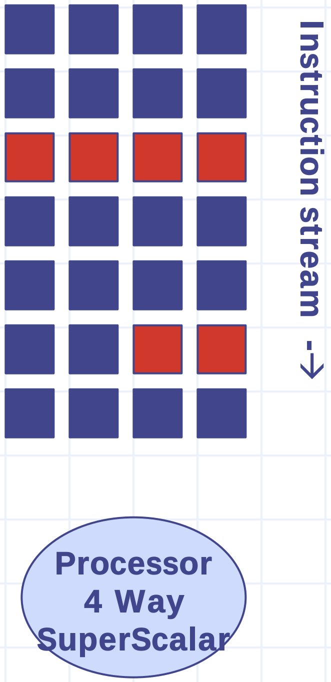
Out Queue

In Queue

Registers, Status

Overview of a Self-Verifying Architecture

- Moves the problem to software
    - Invariants are detected and checks are added to the source code.
    - If an invariant is violated, either:
        - The processor made a mistake.
        - The invariant is incorrect.
        - The program is incorrect.
- No additional hardware is required.

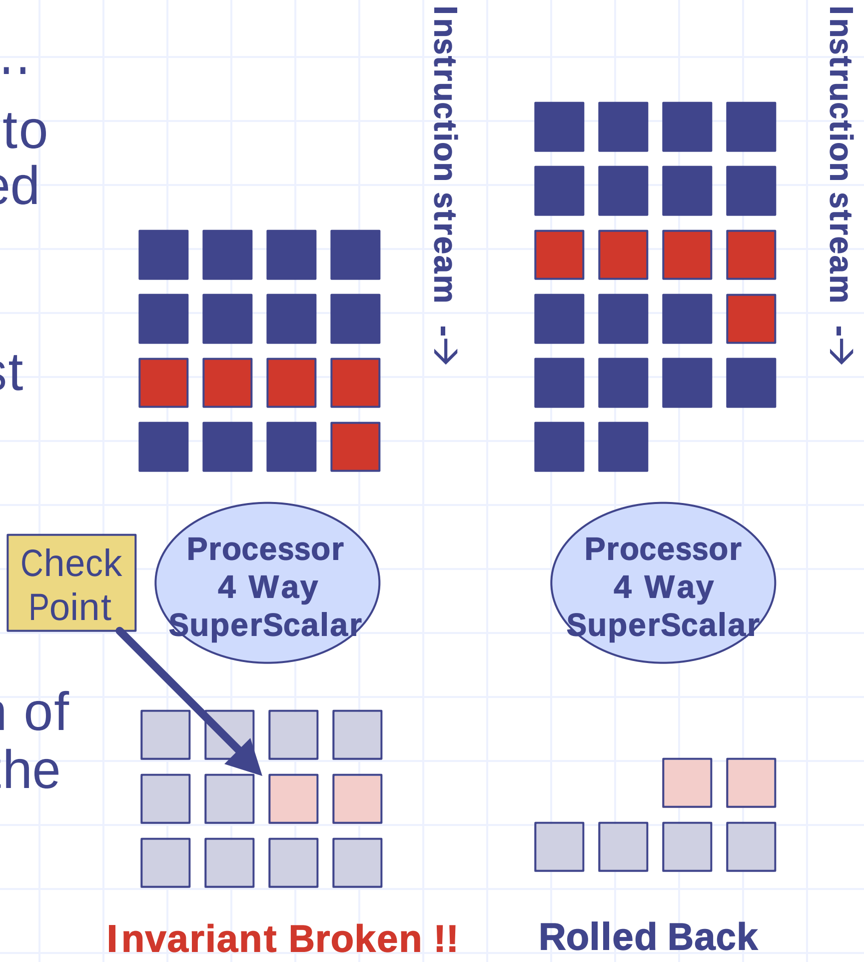


🟥 **Invariant checking instruction**

🟦 **Original instruction**

If an Error is Detected...

- The Processor Erred...
    - The processor needs to load an earlier, verified state.
    - The code must be re-executed from the last successful invariant check.
- Or The Invariant or Program is Incorrect
    - The offending section of code will never pass the invariant check.
    - Execution must terminate.



Experiments: Invariant Granularity

- Objective:
  - To contrast performance differences between invariants at the function level and those at the statement level.
- Experiment
  - Function version , Macro version program with the same functionality and same iterations.
  - Add the same complexity of invariants for function version and in-line version.
- Result
  - Even though, they had a same functionality and same complexity(e.g # of iteration, scope of variable), adding invariant checking for function version was slightly slower than macro.

	Loop Intensive?	Invariant location?	Inv complexity	sim_num_insn	sim_cycle	sim_IPC	il1.miss_rate	dl1.missrate
Function Version	yes			2456388	1448689	1.6956	0.0001	0.0002
Function Version Invariant Checking	yes	outside of function	O(n)	2854093	?sim_cycle=195150 1643839	1.7362	0.0014	0.0002
Macro Version	yes			2454185	1437715	1.707	0.0001	0.0002
Macro Version Invariant Checking	yes	outside of macro	O(n)	2851990	?sim_cycle=194879 1632594	1.7469	0.0014	0.0002

Comparison between function version and macro version

Experiments : Location of Checks

- Objective:

  To contrast performance between programs instrumented with invariants inside functions versus those instrumented outside functions.

- Example:

Vector Addition Function: Invariant Checking Inside Function

```
void foo(int cnt,int *c, int* a,int* b)
{
        int i;
        //Check before
        if (sizeof(a)!=sizeof(b)) i_flag=1;
        if (sizeof(a)!=sizeof(c)) i_flag=1;
        for (i=0;i<cnt;i++){
          c[i]=a[i]+b[i];
        }
        //Check after
        for (i=0;i<cnt;i++){
          if (c[i]!=a[i]+b[i]) i_flag=1;
        }
}
int main ()
{   .....
        foo(100,c,a,b);
    .....
}
```

Vector Addition Function: Invariant Checking Outside of Function.

```
void foo(int cnt,int *c, int* a,int* b)
{
        int i;
        for (i=0;i<cnt;i++){
          c[i]=a[i]+b[i];
        }
}
int main ()
{ .....
        //Check before
        if (sizeof(a)!=sizeof(b)) i_flag=1;
        if (sizeof(a)!=sizeof(c)) i_flag=1;
        foo(100,c,a,b);
        //Check after
        for (j=0;j<100;j++{
          if (c[j]!=a[j]+b[j]) i_flag=1;
        }
 ......
}
```

Experiments : Location Check - continued

- Result
  - Inserting invariant checking outside of function has less performance sacrifice.

Static Vector Addtion	Invariant location?	Inv complexity	sim_num_insn	sim_cycle	delta sim_cycle	sim_IPC	il1.miss_rate	dl1.missrate
	Original file		2456388	1448689		1.6956	0.0001	0.0002
	outside of function	O(1)	2537493	1510387	61718	1.68	0.0016	0.0002
	inside of function	O(1)	2474158	1464732	17770	1.6892	0.0005	0.0002
	outside of function	O(n)	2854093	1643839	195150	1.7362	0.0014	0.0002
	inside of function	O(n)	2788458	1603996	155307	1.7384	0.0001	0.0002

Comparison of Invariant checking inside function and outside of function

Program	Vector addition		Vector Multiply		Vector division	
Invariant Checking	Inside	Outside	Inside	Outside	Inside	Outside
Cycle	1464732	1510387	1508036	1520287	1855598	1856132
IL1. Miss	1162	4135	1364	4135	1780	1947
DL1. Miss	181	191	193	191	311	313
L2 Cache look up time	1343	4326	1557	4326	2093	2262
Branch prediction miss	344	1531	1531	1531	11404	11405

Comparison of Invariant Checking inside function and outside of function for different programs

A Concern : Complexity

- Checking some code for invariants can be expensive!
  - Checking a list's size is O(1) (hopefully).
  - Checking that two lists are identical is O(n + m).
  - Checking sometimes need to allocate extra variables.
- How exhaustive must we be?
  - Must we test the value of each element of an array, for example?
  - Or, can we test a random selection of the array and get almost the same confidence of success?

In Summary : Current Results

- Statement vs. Functional Granularity
  - Adding invariants to macros is less expensive than adding invariants to functions.
  - Current invariant detection technology does not support the less costly alternative.
  - For object oriented programming, we can’t avoid using method call.
- Location of Checks
  - Placing checks as close as possible to the code being verified reduces the execution penalty.
  - Most of the difference comes from instruction cache misses.
- Consideration of Complexity of Invariant Checking

Future Work : This quarter, we hope?

- Measuring Invariant Checking Penalties
  - Our current results are misleading, as they depend on too wide a set of factors.
  - We are searching for a metric (or set of metrics) that adequately account for the most important factors.
- Implementing Safe Pointer Checking
  - Invariant checking can be used to catch illegal memory accesses (array out of bounds, unallocated memory).
- Measure Invariant Benefits when Errors Occur
  - We will modify SimpleScalar so it periodically miscomputes some instruction(s).
  - Using this version of SimpleScalar, we hope to get an idea of the “break-even” point for our technique.