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?

**Invariant** 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.

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
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.
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.
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.

<table>
<thead>
<tr>
<th></th>
<th>Loop Intensive?</th>
<th>Invariant location?</th>
<th>Inv complexity</th>
<th>sim_num_insn</th>
<th>sim_cycle</th>
<th>sim_IPC</th>
<th>il1.miss_rate</th>
<th>dl1.missrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Version</td>
<td>yes</td>
<td></td>
<td></td>
<td>2456388</td>
<td>1448689</td>
<td>1.6956</td>
<td>0.0001</td>
<td>0.0002</td>
</tr>
<tr>
<td>Function Version</td>
<td>yes</td>
<td>outside of function</td>
<td>O(n)</td>
<td>?sim_cycle=195150</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macro Version</td>
<td>yes</td>
<td>outside of macro</td>
<td>O(n)</td>
<td>2854093</td>
<td>1643839</td>
<td>1.7362</td>
<td>0.0014</td>
<td>0.0002</td>
</tr>
<tr>
<td>Macro Version</td>
<td>yes</td>
<td></td>
<td></td>
<td>2454185</td>
<td>1437715</td>
<td>1.707</td>
<td>0.0001</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Vector Addition Function: Invariant Checking Inside Function</th>
<th>Vector Addition Function: Invariant Checking Outside of Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(int cnt, int *c, int *a, int *b)</td>
<td>void foo(int cnt, int *c, int *a, int *b)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>int i;</td>
<td>int i;</td>
</tr>
<tr>
<td>// Check before</td>
<td>for (i=0; i&lt;cnt; i++) {</td>
</tr>
<tr>
<td>if (sizeof(a)!=sizeof(b)) i_flag=1;</td>
<td>c[i]=a[i]+b[i];</td>
</tr>
<tr>
<td>if (sizeof(a)!=sizeof(c)) i_flag=1;</td>
<td>}</td>
</tr>
<tr>
<td>for (i=0; i&lt;cnt; i++){</td>
<td>int main()</td>
</tr>
<tr>
<td>c[i]=a[i]+b[i];</td>
<td>{</td>
</tr>
<tr>
<td>}</td>
<td>// Check before</td>
</tr>
<tr>
<td></td>
<td>if (sizeof(a)!=sizeof(b)) i_flag=1;</td>
</tr>
<tr>
<td></td>
<td>if (sizeof(a)!=sizeof(c)) i_flag=1;</td>
</tr>
<tr>
<td></td>
<td>foo(100, c, a, b);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>// Check after</td>
</tr>
<tr>
<td></td>
<td>for (j=0; j&lt;100; j++) {</td>
</tr>
<tr>
<td></td>
<td>if (c[j]==a[j]+b[j]) i_flag=1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>main()</td>
<td>linik()</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>.....</td>
<td>foo(100, c, a, b);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
Experiments: Location Check - continued

Result

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

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

Comparison of Invariant checking inside function and outside of function

<table>
<thead>
<tr>
<th>Program Invariant Checking</th>
<th>Vector addition</th>
<th>Vector Multiply</th>
<th>Vector division</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>1464732</td>
<td>1508036</td>
<td>1855598</td>
</tr>
<tr>
<td>IL1. Miss</td>
<td>1162</td>
<td>1364</td>
<td>1780</td>
</tr>
<tr>
<td>DL1. Miss</td>
<td>181</td>
<td>193</td>
<td>311</td>
</tr>
<tr>
<td>L2 Cache look up time</td>
<td>1343</td>
<td>1557</td>
<td>2093</td>
</tr>
<tr>
<td>Branch prediction miss</td>
<td>344</td>
<td>1531</td>
<td>11404</td>
</tr>
</tbody>
</table>

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.