Automated Derivation of Application-Aware Error Detectors

Zbigniew Kalbarczyk

K. Pattabiraman, G.P. Sagesse, N. Nakka, D. Chen, R. Iyer

Center for Reliable and High performance Computing

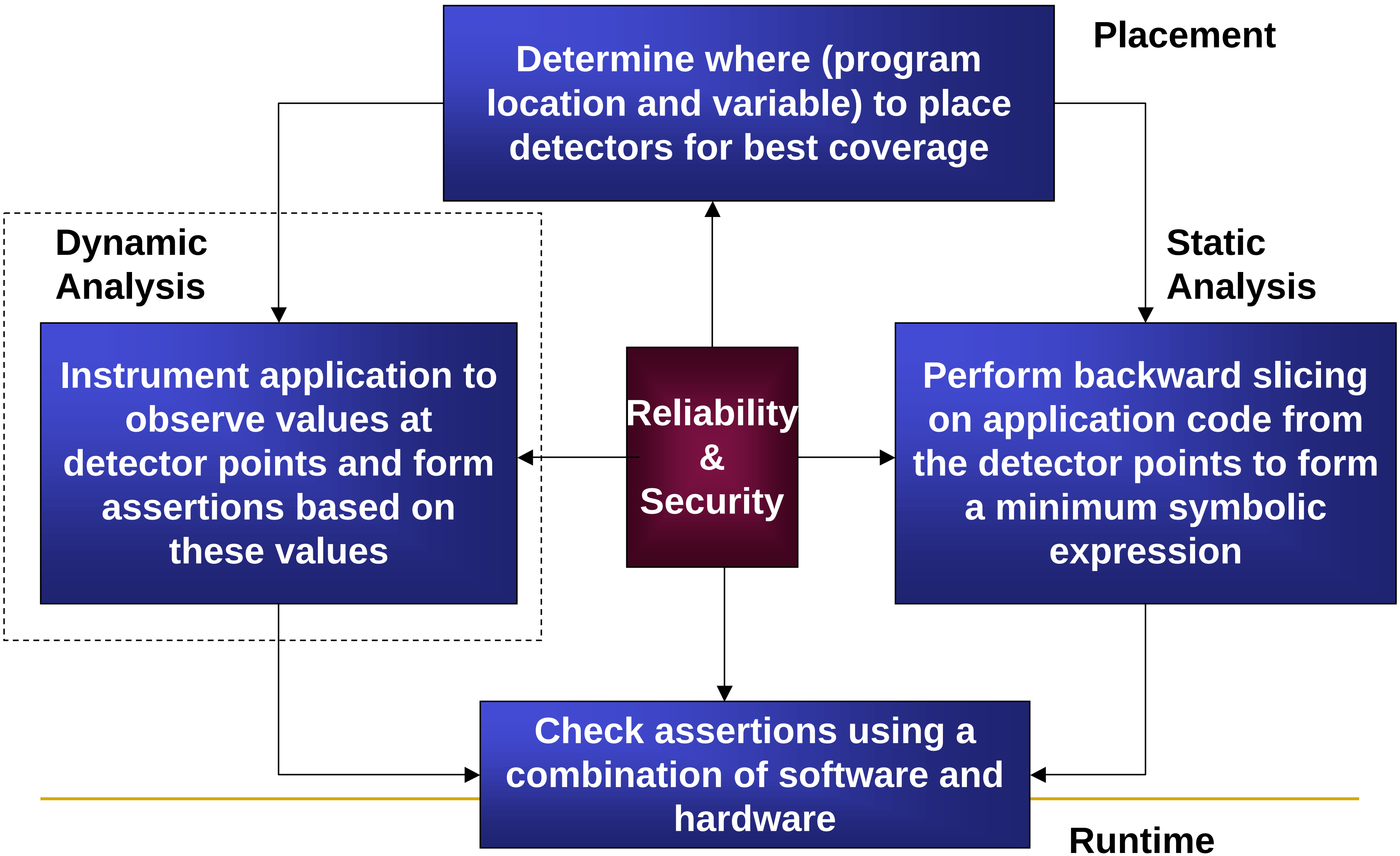
University of Illinois at Urbana-Champaign

www.crhc.uiuc.edu/DEPEND

Research Goals

- Application-aware error detectors
  - Provide application-specific error detection at low-cost for high-performance platforms
  - Limit error propagation to ensure crash-failure semantics and reduce error detection latency
- Automatically derive fine-grained detectors to
  - Maximize error detection coverage
  - Minimize performance impact
- Implement in software / hardware

Approach



Determine where (program location and variable) to place detectors for best coverage
Placement
Dynamic Analysis
Static Analysis
Instrument application to observe values at detector points and form assertions based on these values
Reliability & Security
Perform backward slicing on application code from the detector points to form a minimum symbolic expression
Check assertions using a combination of software and hardware
Runtime

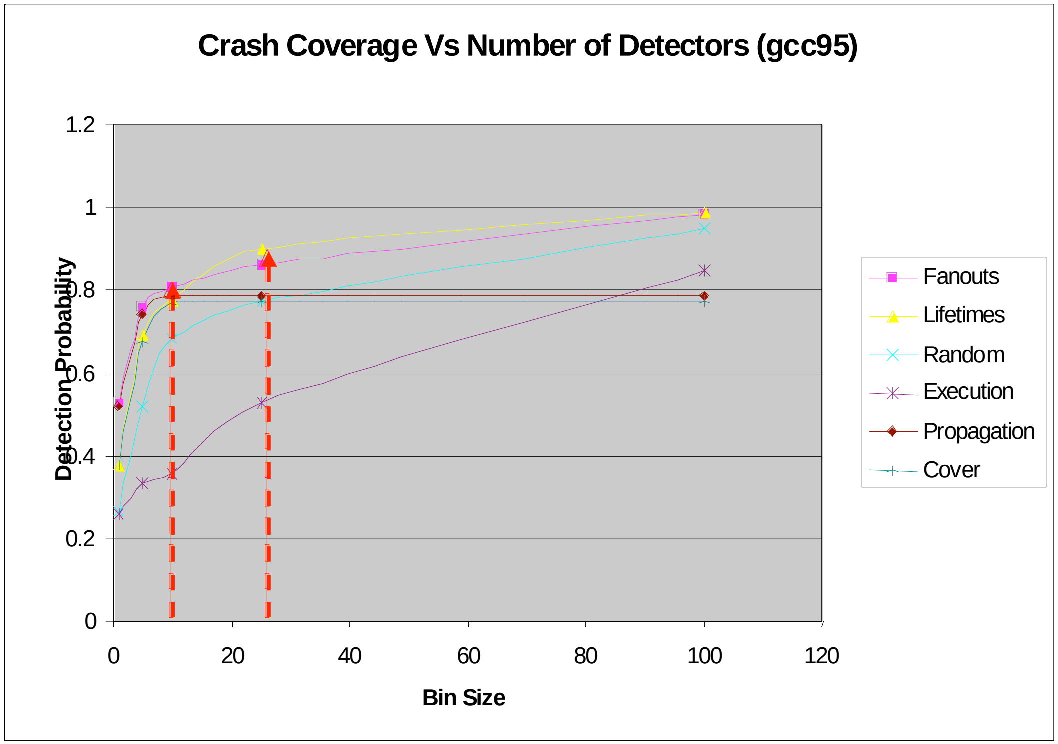
Fault Models

- Hardware errors
  - Incorrect computation (not detected by ECC)
  - Soft errors in memory, registers and cache
  - Errors in instruction issue/decode
- Software errors
  - Uninitialized values or incorrectly initialized values
  - Memory corruption, dangling pointers
  - Integer overflows, values out-of-bounds
  - Timing errors and race conditions

Where to Place the Detectors?

- Choose variable to check and location to place the detector
- Starting Point: construct Dynamic Dependence Graph of the program
- Compute metrics to choose candidate points for detector placement
  - e.g., fanout, lifetime
- Evaluate detectors placed according to different metrics
  - Fault-injections into data

Coverage for Multiple Detectors (ideal detectors)



gcc95 benchmark

- Coverage for crashes:
  - 80% with 10 detectors, 97 % with 100 detectors
- Coverage for fail-silence violations (silent-data corruptions)
  - 60% with 10 detectors, 80 % with 100 detectors
- Benign errors detected
  - 4 % with 10 detectors, 10 % with 100 detectors
- Placing detectors randomly on hot-paths:
  - Need ~100 ideal detectors to achieve 90% coverage

Detector Classes

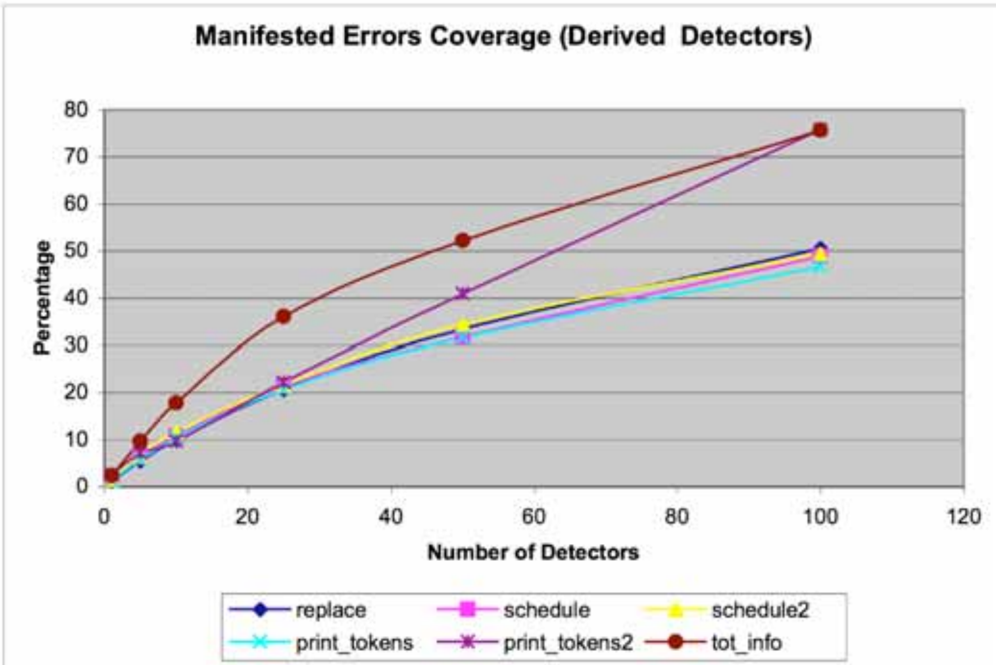
- A detector is a check on the value of a program variable memory location at a particular execution point in the program code

Class Name	Example of Checking Expression
Constant	a[i] == c
Alternate	(a[i] ==x and a[i-1]==y) or (a[i]==y and a[i-1]==x)
Multi-Value	(a[i] in *Values*), where *Values* is a set of possible values
Range	min <= a[i] <= max
ConstantDiff	(a[i] – a[i-1]) == c
BoundedDiff	min <= (a[i] – a[i-1]) <= max

Experimental Setup

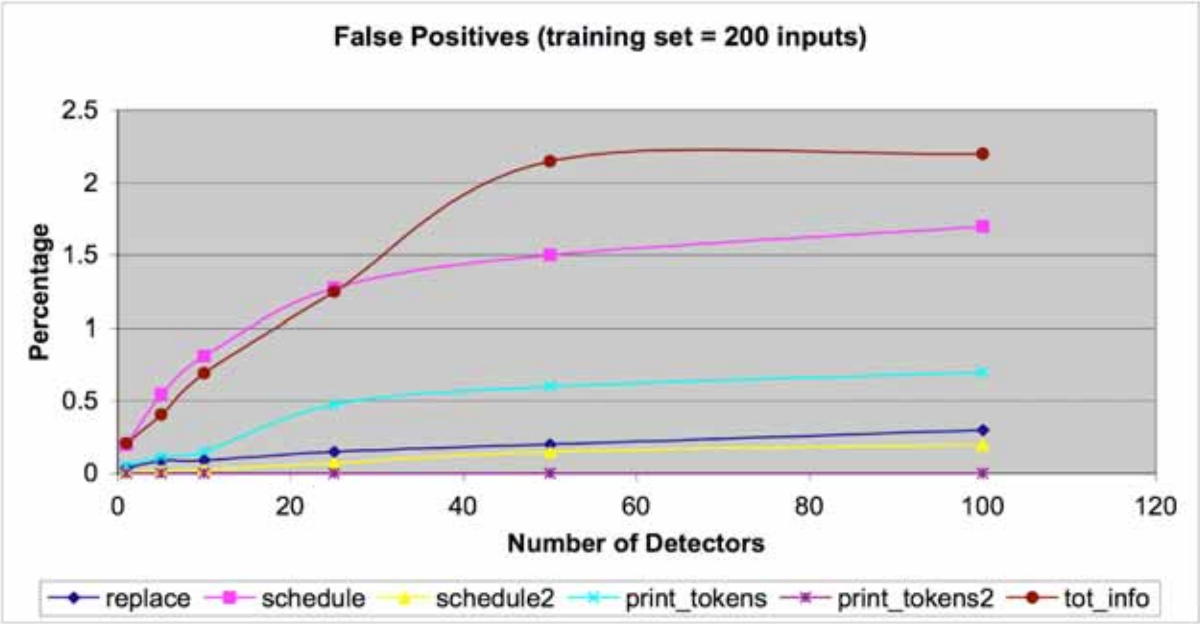
- Steps in Evaluation:
  - **Analysis**: Detector placement and code instrumentation
  - **Training**: Learning detectors using representative inputs
  - **Testing:** Fault-injection in application data
- Tool used for evaluation: modified version of Simplescalar simulator (functional simulation)
- Application Workload: Siemens suite
  - C programs with 100-1000 lines of code

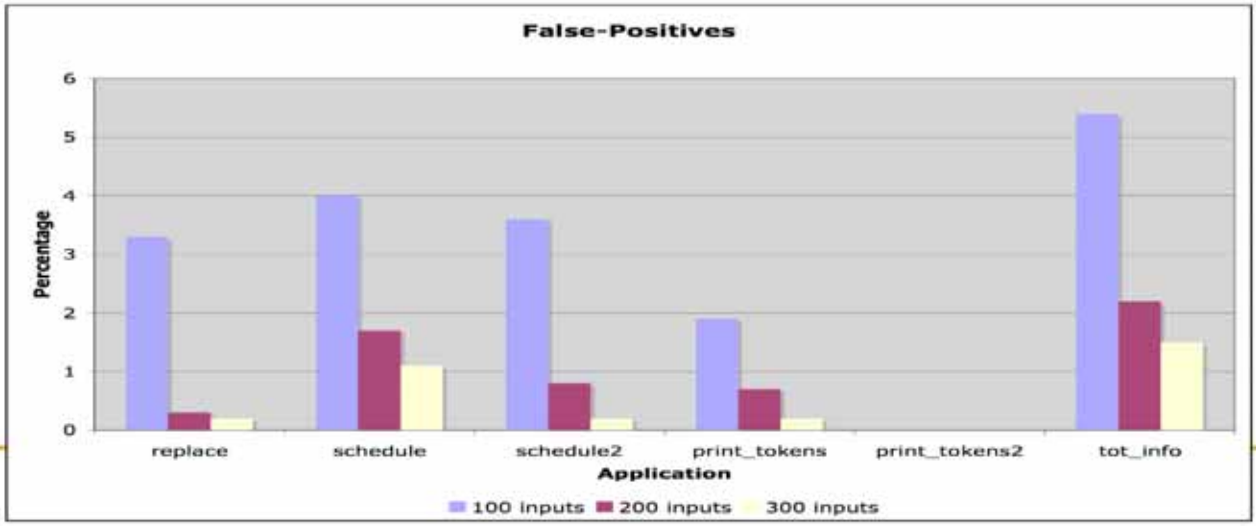
Dynamic Detector Results



Type of Failure	**Minimum Coverage**	**Maximum Coverage**
Manifested Errors	50%	75%
Program Crashes	45% (*print_tokens)*	65% (*tot_info*)
Fail-Silent Violations	25% (*schedule2*)	75% (*tot_info*)

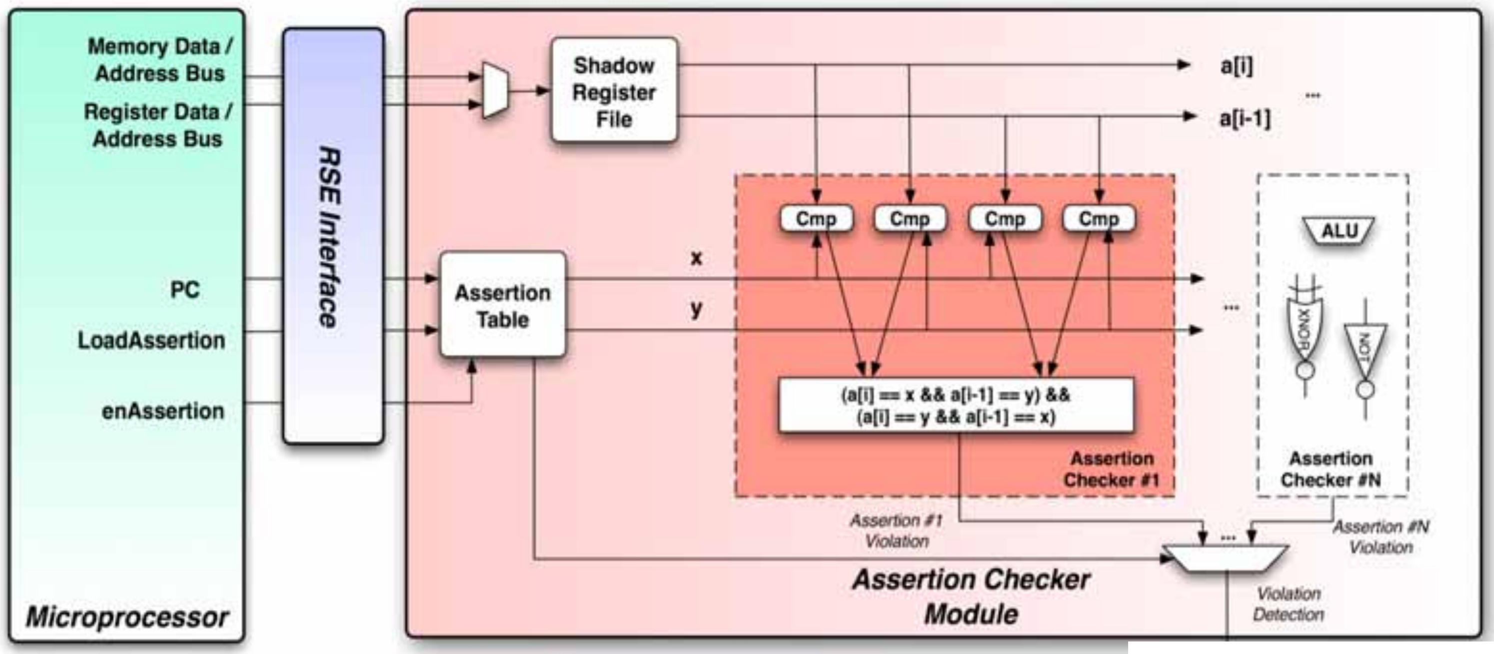
False-Positives





Hardware Implementation

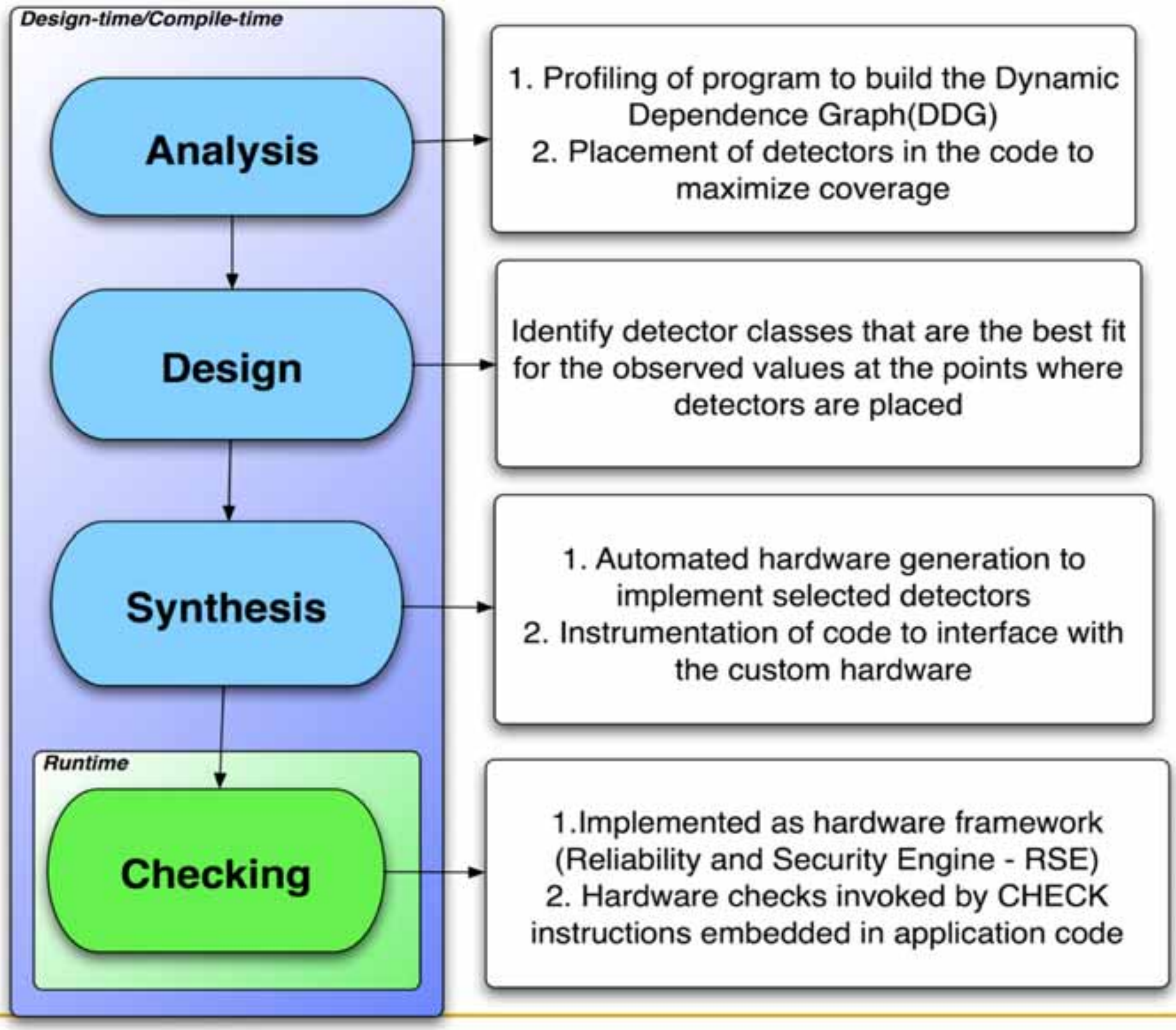
- Reliability and security Engine (RSE)
  - reconfigurable processor-level framework for reliability and security
- Detectors implemented as an RSE module consisting of:
  - Shadow Register File - holds the state of the checked location
  - Assertion Table - stores the assertions' parameters
  - Data-path - check assertions independently from processor



Area overhead **30 %**

Performance Overhead= **5.6 %**

Approach Summary



Ongoing and Future Work

- Dynamic Analysis: Extension to larger programs and multi-valued detectors
- Static Analysis: Concise representation of checking expressions and compiling to H/W
- Extension to Security: Signatures based on Information-flow in a program
- Formal methods of verification of derived detectors: Model Checking/Theorem Proving
- Integerated Hardware/Software framework with support from the OS