Hardware Support for Reliability and Security: *Looking at the Future*

Ravishankar Iyer
(with Nithin Nakka, Prateek Patel, Karthik Pattabiraman, Zbigniew Kalbarczyk)

Center for Reliable and High Performance Computing
Coordinated Science Laboratory and Information Trust Institute
University of Illinois at Urbana-Champaign

Dependability Techniques: A Little Bit of History

- Testing and failure recovery in the ILLIAC machine at Illinois in the early 1950s.
  - fault diagnosis using battery of programs that exercised different sections of the machine
- Space-borne computing systems
  - JPL-STAR (Self-Testing and Repair) computer (1971)
- Aviation
  - Fly-by wire F-16 (?), Airbus, Boeing
- Research Machines: c.mmp, FTMP, SIFT
- Commercial systems
  - AT&T No.5 ESS
  - IBM S/360 and IBM S/370
  - Tandem Integrity S2

Evolution of Fault Sources, Levels of Integration, Users, and User Sophistication (Siewiorek, Kabalczyk, Chillerege)

Decade	1970s	1980s	1990s	2000s
Typical systems	Mainframes	Workstations	Personal computers	Mobile devices;,, e.g., cellphones, PDAs
Fault/error sources	Hardware	Hardware, network	Hardware, network, software, human errors	Hardware, software, wireline/wireless networks, environment; e.g., frequent connectivity loss, malicious faults
Integration/ complexity	Close systems; highly custom designs, where both hardware and OS are fully controlled by the vendor	Mostly close systems; network connectivity; standard interfaces exposed to users	Open systems; wide access to network; COTS operating systems; third-party hardware and software	Open systems; proprietary and COTS operating systems; highly integrated PC-like systems
People/users	Tens of thousands	Millions	10s of millions	100s of millions
Level of user sophistication/ training	BS in engineering; 5000 hours	Basic knowledge in computing; 500 hours	Basic computing literacy; 50-100 hours	Training at the time of a purchase of a device; Hours

Growing Cost of Commodity Systems

- Successful and cost effective use of *Parity, ECC, and RAID* in commodity systems
- Use of Significant redundancy in hardware and software led to high overheads in
  - performance cost,
  - hardware components and software developments cost, e.g.,
    - IBM MVS operating system devotes 50% of its software code base to fault management,
    - IBM G5 processor dedicates 35% of its processor silicon area to fault detection and tolerance hardware
  - validation becomes increasingly complex and difficult
- **One-size-fits-all architectures**
  - **OK** for high-end, high-cost systems, e.g., military, telecommunication, and financial (Wall Street) applications
  - **NOT OK** for commodity environments

What Changed?

- Explosion of computing devices, e.g., mobile/hand-held devices in a wide variety of applications,
- Computing has become a social enterprise
- Massive computing data centers servicing networked entities from telecom to internet service providers to banks
- New computing paradigms, e.g., cloud
- Ubiquitous computing, present in everyday appliances, e.g., microwaves, vehicles, e-commerce and health monitoring,
- With computing as a major enabling enterprise, outages cannot longer be ignored or brushed aside with a marginal or cursory solution

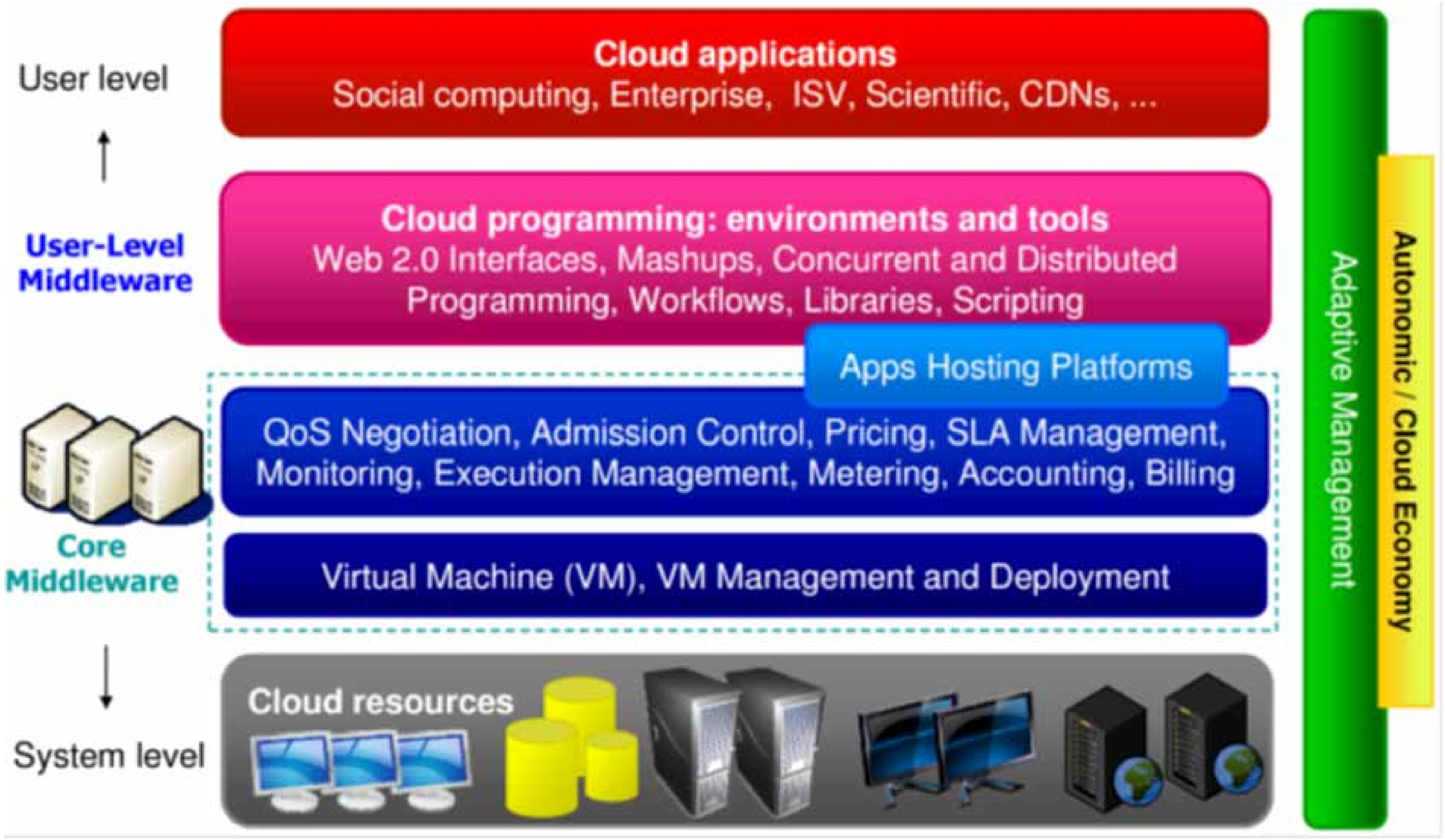
Hardware Level Issues

- A 10 petaflop supercomputer with ~300K cores has a very substantial error rate
  - estimated MTBF is 100 min (hard and soft errors) and checkpointing overhead is about 25%)
- Decreasing feature sizes, bring reliability concerns at the device level
  - e.g., recent bug in Intel’s Cougar Point SATA (Serial Advanced Technology Attachment) port on the 6-Series Chipset
  - “in some cases, the Serial-ATA (SATA) ports within the chipsets may degrade over time, potentially impacting the performance or functionality of SATA-linked devices such as hard disk drives and DVD-drives.”
  - The recall may reduce Intel's revenue by around $300 million and cost around $700 million to completely repair and replace affected systems

Issues at the platform level

- Use of virtual machine-based systems transforms the system view by introducing the Hypervisor
  - new set of interactions and consequent failure modes in the system
- Non-uniform, dynamic geographic distribution of the nodes in the cloud
  - violation of assumptions of traditional distributed systems regarding communication overheads
  - legacy checkpointing techniques may incur significant overhead and cannot be applied naïvely in the new scenario without investigation
  - non-deterministic costs due to the dynamic nature of the distributed system

Cloud Computing layered architecture


User level
User-Level Middleware
Core Middleware
System level
Cloud applications
Social computing, Enterprise, ISV, Scientific, CDNs, ...
Cloud programming: environments and tools
Web 2.0 Interfaces, Mashups, Concurrent and Distributed Programming, Workflows, Libraries, Scripting
Apps Hosting Platforms
QoS Negotiation, Admission Control, Pricing, SLA Management, Monitoring, Execution Management, Metering, Accounting, Billing
Virtual Machine (VM), VM Management and Deployment
Cloud resources
Adaptive Management
Autonomic / Cloud Economy

Example CNN Cloud Failures

- Providing a higher level of reliability and availability remains a major a challenges of Cloud computing
- Amazon S3 failure
  - 8 hour outage of Amazon services on July 20, 2008
  - caused by a single bit error in messages communicated (using a gossip protocol) between the servers
  - data corrupted before being sent on the network using checksum
- Google AppEngine’s partial outage (6/17/2008) due to a programming error
- Microsoft Azure’s outage (3/17/2009) for 22 hours due to the malfunction in the hypervisor

Early Warning of Such Failures

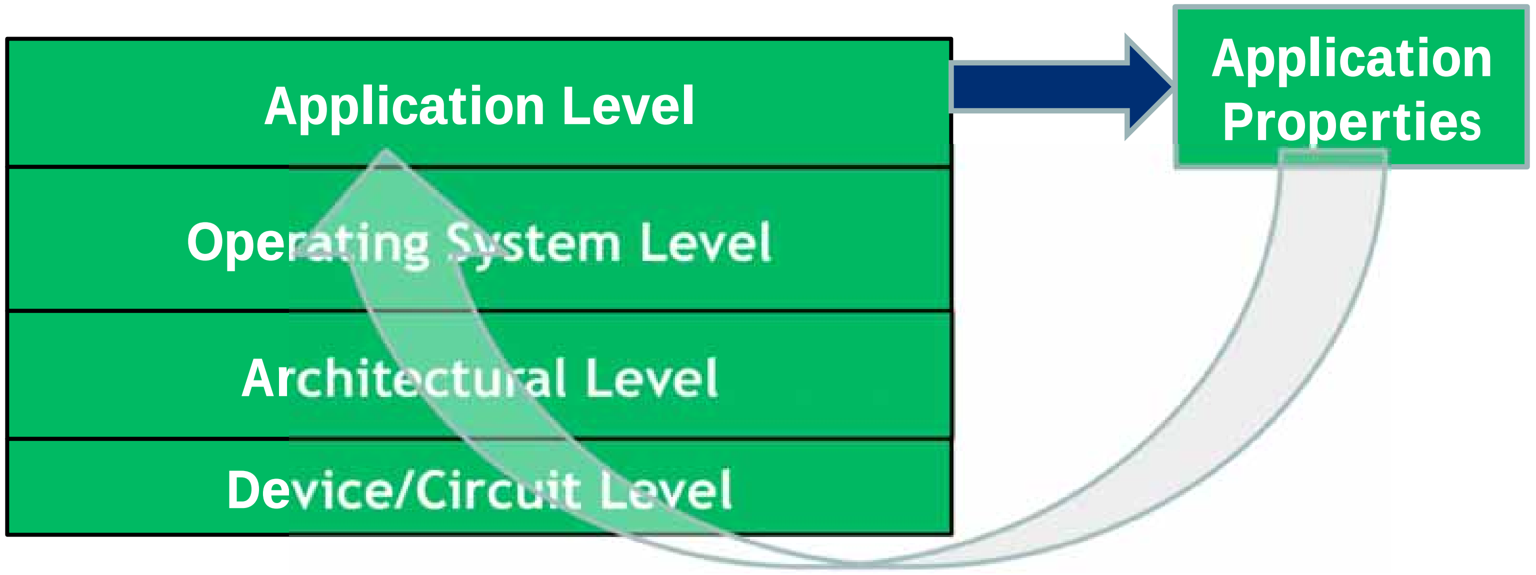
- Similar failure patterns demonstrated in an error-injection based experimental analysis of the Ensemble GCS – Group Communication System (done at Illinois)
  - GCS formally specified and verified, but it constitutes only about 5% of the entire code base
  - Additionally, 5–6% of application failures are due to an error that escapes the GCS error-containment mechanism and manifests as silent data corruption

Competing Forces

- HIGH dependability requirements for commodity systems
  - comparable with legacy systems that extensively used redundancy
- SMALL cost margins for high availability
  - preclude use of traditional techniques, as-is, for these commodity systems
- New low-cost techniques that are tailored to the specific needs of the application are required

Application-aware Detection

- **App-aware: Use application properties to derive error and attack detectors (runtime checks)**
  - Achieve high-detection coverage with low overheads
  - Detect only attacks and errors that matter to the application
  - Ensure that attack and error is detected before propagation



Runtime Checks (Detectors)

Challenges: Application-aware

- **How do we identify app. properties to check ?**
  - Compiler-based static and dynamic analysis
- **How do we validate the approach ?**
  - **Experimental Methods:** Fault-injection, modeling
  - **Formal Methods:** Model-Checking
- **How do we check/monitor the application ?**
  - Software or hardware (programmable)

Unified Design Framework

Reliability

Identify critical variables and their location within a program

Security

Apply heuristics, e.g., *fanouts*, to identify critical variables.

Use application semantics to identify security critical variables, e.g., a password

Critical Variable Recomputation

Static program analysis: Extract backward slices of critical variables

Information-flow signatures (IFS)

Generate correctness checks for data values in critical program locations

Generate checks to verify that value is produced by legitimate instructions.

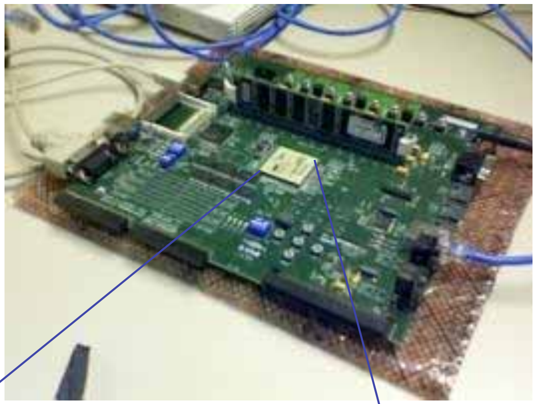
Runtime checking to ensure integrity of critical variables according to the slice

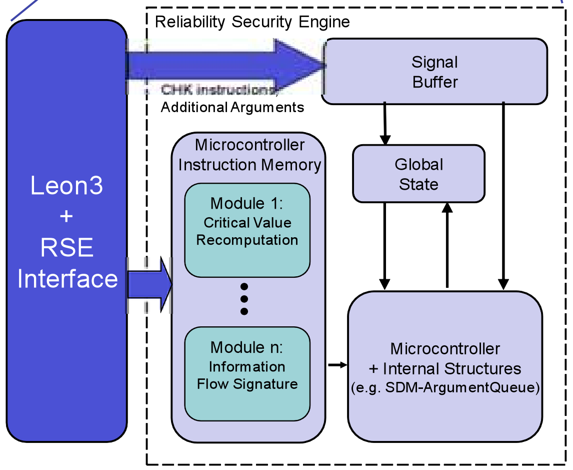
Techniques and Attack/Error Models

- **Selectively enforce source-level properties of writes to critical data at runtime**
- **Techniques:**
  - IFS (information flow signatures) – protects critical data integrity
  - CVR (critical value re-computation) – verifies correctness of critical data computation
- **Attack Models**
  - Memory corruption attacks (e.g. buffer overflows)
  - Insider attacks (malicious libraries, 3rd party plugins)
  - Program binary modifications after compilation
- **Fault Models**
  - Soft errors
  - Memory corruption errors
  - Race conditions and/or atomicity violations

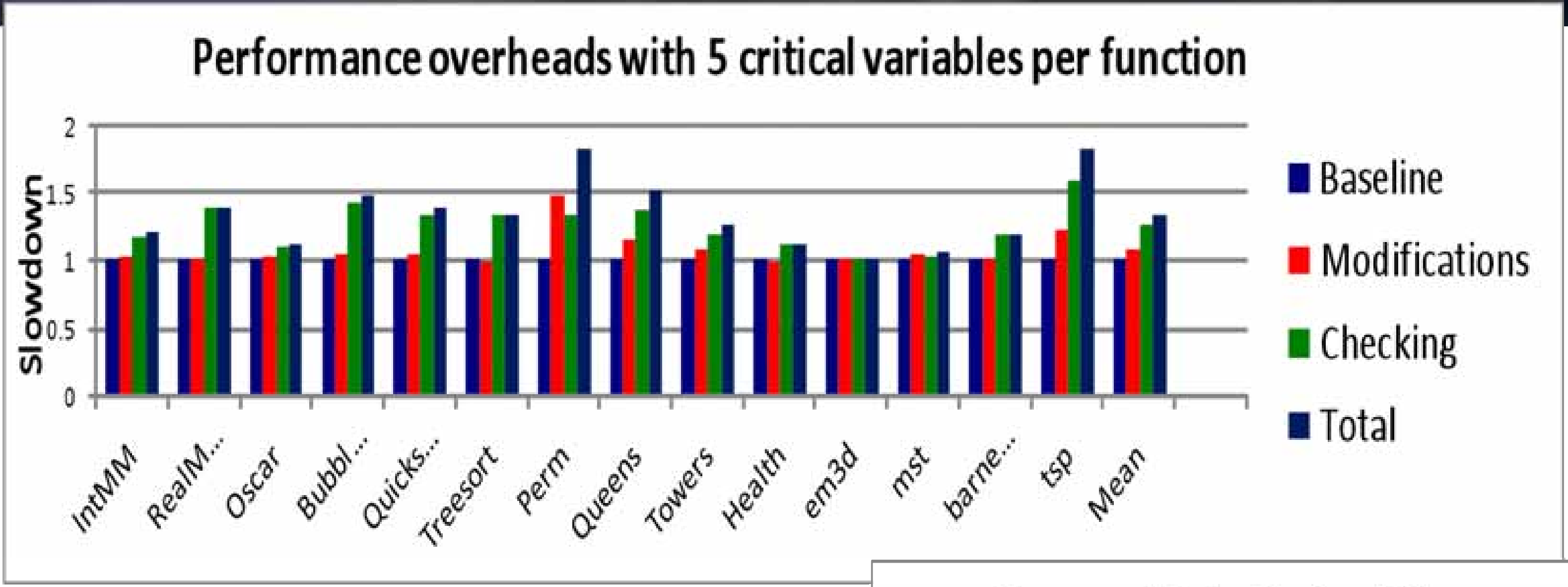
Hybrid Implementation (hw + sw)

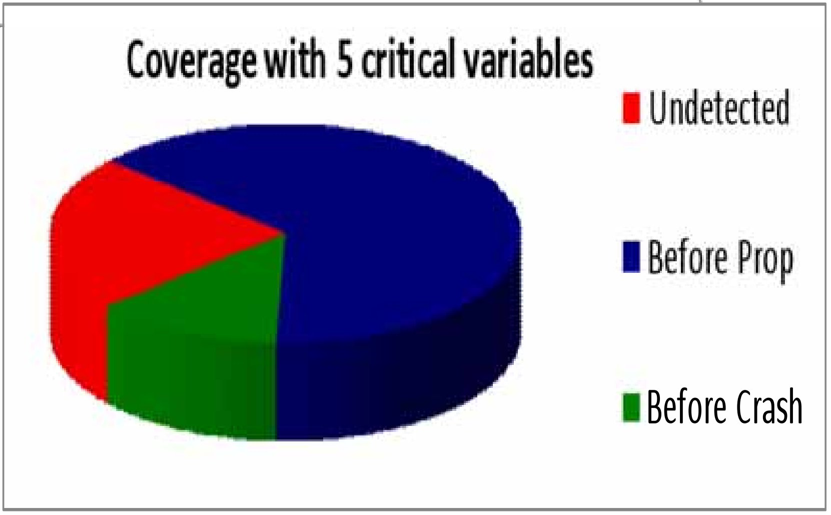
- Runtime enforcement using combination of hardware and software
- Single hardware framework to host modules providing reliability and security protection
  - FPGA-based prototype evaluated on embedded programs and network applications (e.g., OpenSSH)
  - Performance overhead 1% to 70% (depending on the application)
  - Area overhead = 4% to 50 % (relative to Leon3 processor)




Reliability Security Engine
Leon3 + RSE Interface
CHK instructions, Additional Arguments
Signal Buffer
Microcontroller Instruction Memory
Module 1: Critical Value Recomputation
Module n: Information Flow Signature
Global State
Microcontroller + Internal Structures (e.g. SDM-ArgumentQueue)

CVR Results: Coverage and Performance



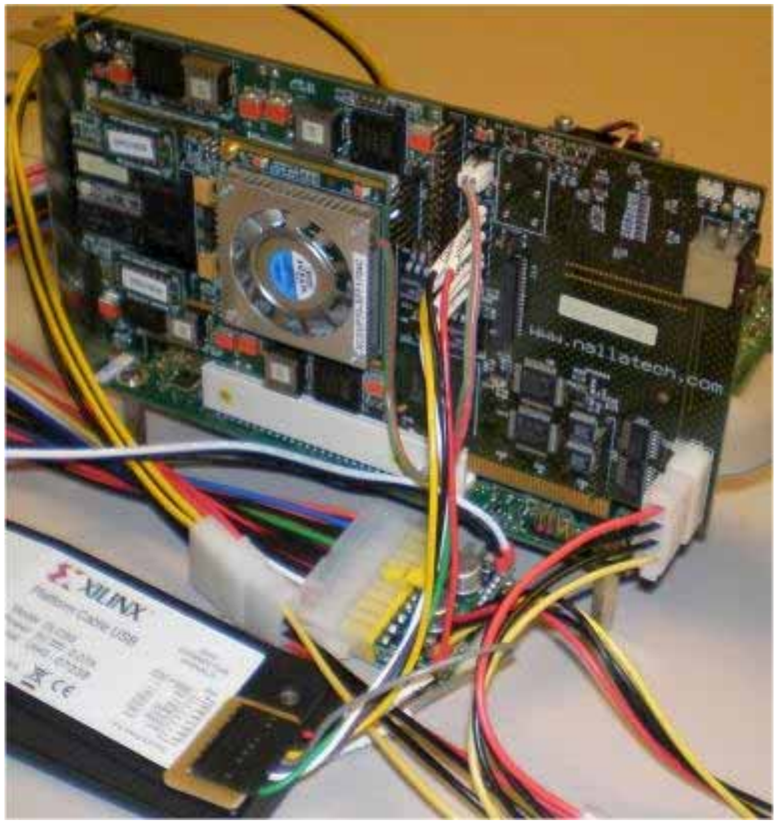


- **Avg. SW Performance Overhead**
  - **Checking = 25%**
  - **Modification = 8%**
  - **Total = 33 %**
- **Avg. Coverage (Crashes)**
  - **Before Prop = 64 %**
  - **Before Crash = 13%**
  - **Total Detected = 77 %**
- **Benign errors detect = 3 %**

Results (Hardware Checking)

Performance	Cycles	Performance Overhead
No Instrumentation	30,067	-
SW Static-Detector Module	136,607	354%
HW Static-Detector Module	57,411	91%
Static-Detector Module Optimized w/DMA	**30,688**	**2%**

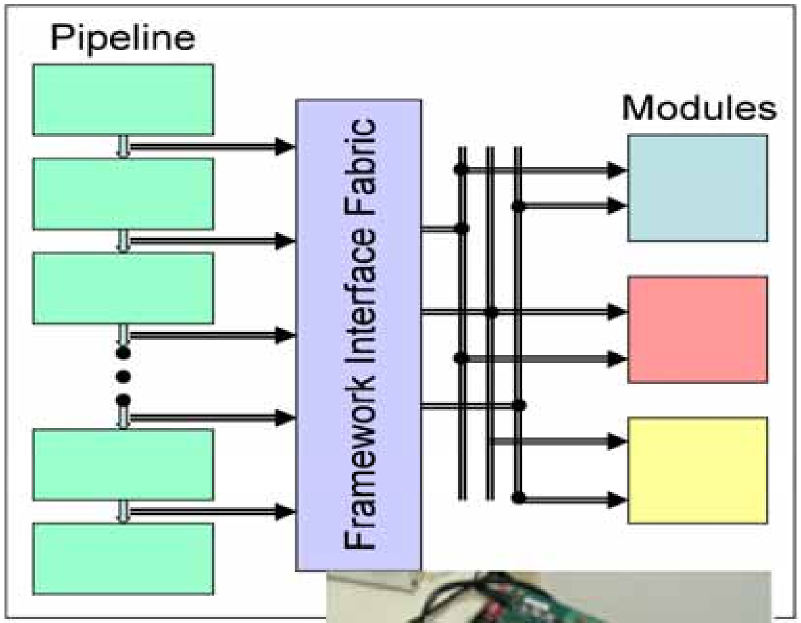
Synthesis	Slices	Max Frequency
DLX	12,262	76 MHz
DLX + Static-Detector Module	12,533	77 MHz



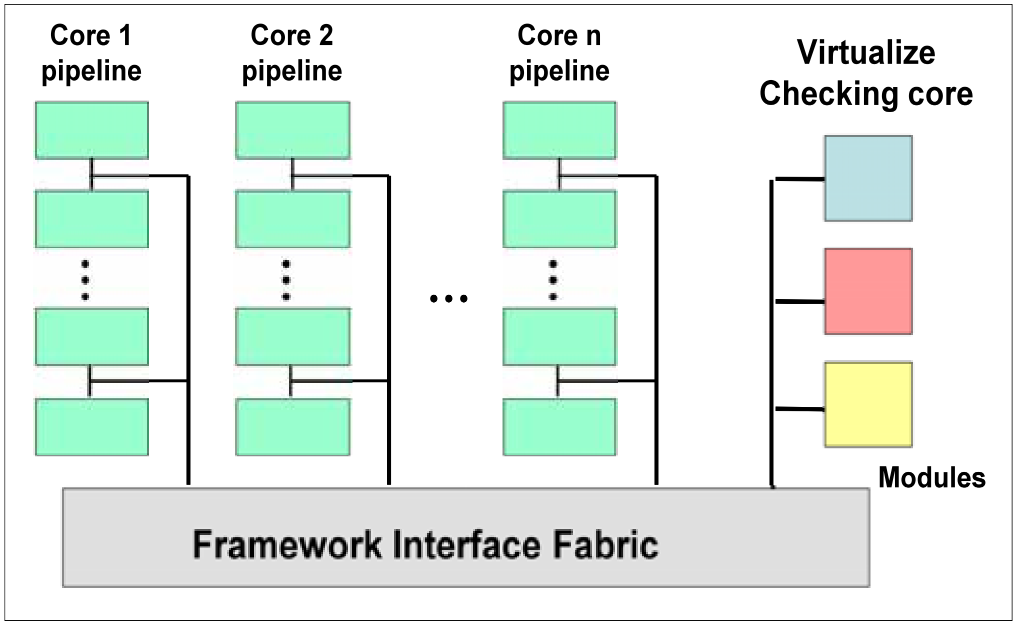
Significant performance gain over software implementation

Where do we go from here?

Reliability and Security Engine (RSE)
Single-core chip architecture

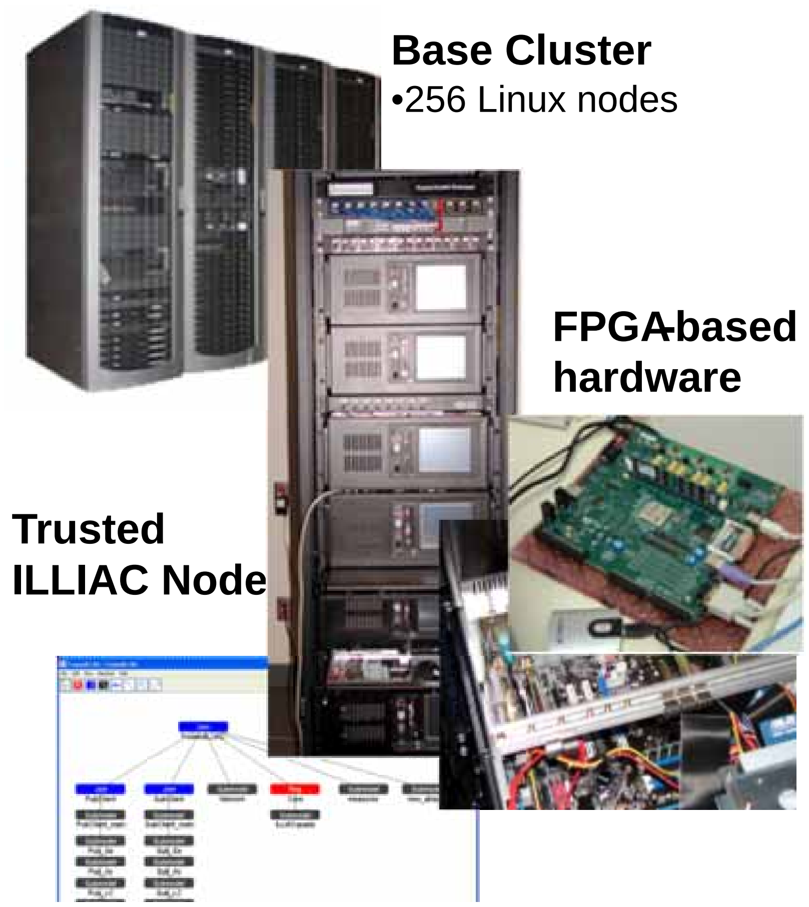


Heterogeneous multi-core chip architecture



FPGA-based prototype

Trusted ILLIAC: A Configurable, Application-Aware, High-Performance Platform for Trustworthy Computing



- Provide application-specific level of reliability and security, while delivering performance.
- Enforce customized levels of trust via an integrated approach involving:
  - configurable hardware,
  - compiler methods to extract applications security and reliability properties,
  - configurable OS and middleware.
- Enable rapid deployment of low-cost application aware engines and processing cores
- Support OS and middleware to facilitate model-driven trust management and oversight in protecting against wide range of attacks and failures.