Survivable Software

AFOSR FA0550-09-1-0481, Start Date: 6-1-09

David R. Luginbuhl, Ph.D.
Program Manager, Systems and Software

Radu Grosu, Scott A. Smolka,
Scott D. Stoller, Erez Zadok
Stony Brook University

Klaus Havelund
NASA JPL

Safe & Secure Systems & Software Symposium – S5
June 16, 2010

STONY BROOK
STATE UNIVERSITY OF NEW YORK

NASA

Mars Rover Sets Endurance Record

Opportunity has been operating on Mars for six years and 140 days!

Human(s) in the Loop

JPL Control Room

Project Overview

- ***Objective***: develop theory & techniques for ***Survivable Software***, new breed of software for embedded & mission-critical applications
- ***Scientific Approach***:
  - ***InterAspect:*** More flexible AOP framework for specification, instrumentation, and recovery
  - ***SMCO:*** Feedback control of monitoring overhead
  - ***HSR:*** Hierarchical Simplified Redundancy
- ***Breakthrough Opportunity***: *equip systems with software that continues to function in presence of residual defects*

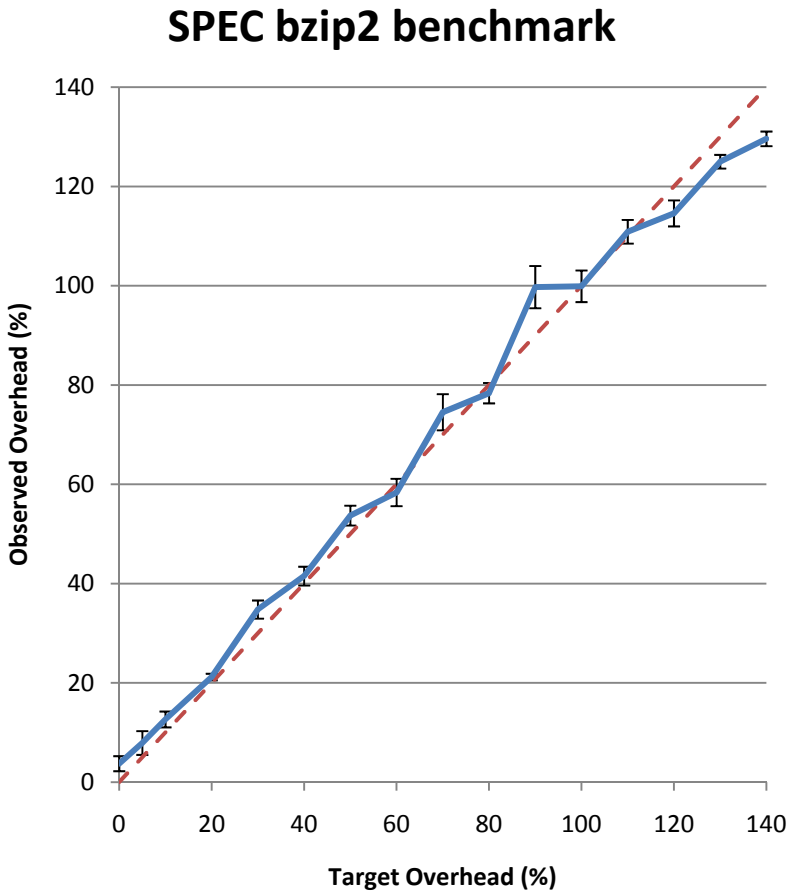
Project Overview

- *Objective*: develop theory & techniques for *Survivable Software*, new breed of software for embedded & mission-critical applications
- ***Scientific Approach***:
  - ***InterAspect:*** More flexible AOP framework for specification, instrumentation, and recovery
  - ***SMCO:*** Feedback control of monitoring overhead
  - *HSR:* Hierarchical Simplified Redundancy
- *Breakthrough Opportunity*: *equip systems with software that continues to function in presence of residual defects*

This Talk

- ***SMCO***: Software Monitoring with Controllable Overhead
  - ***Cascade*** and ***Global*** controllers
  - ***Base overhead*** and ***overhead tracking*** optimizations
  - ***Integer range analysis*** and ***leak detection***
- ***InterAspect:*** Aspect-oriented instrumentation for GCC
  - ***Plug-in architecture*** for GCC
  - InterAspect ***API***
  - ***Function body duplication***

SMCO: Software Monitoring with Controllable Overhead

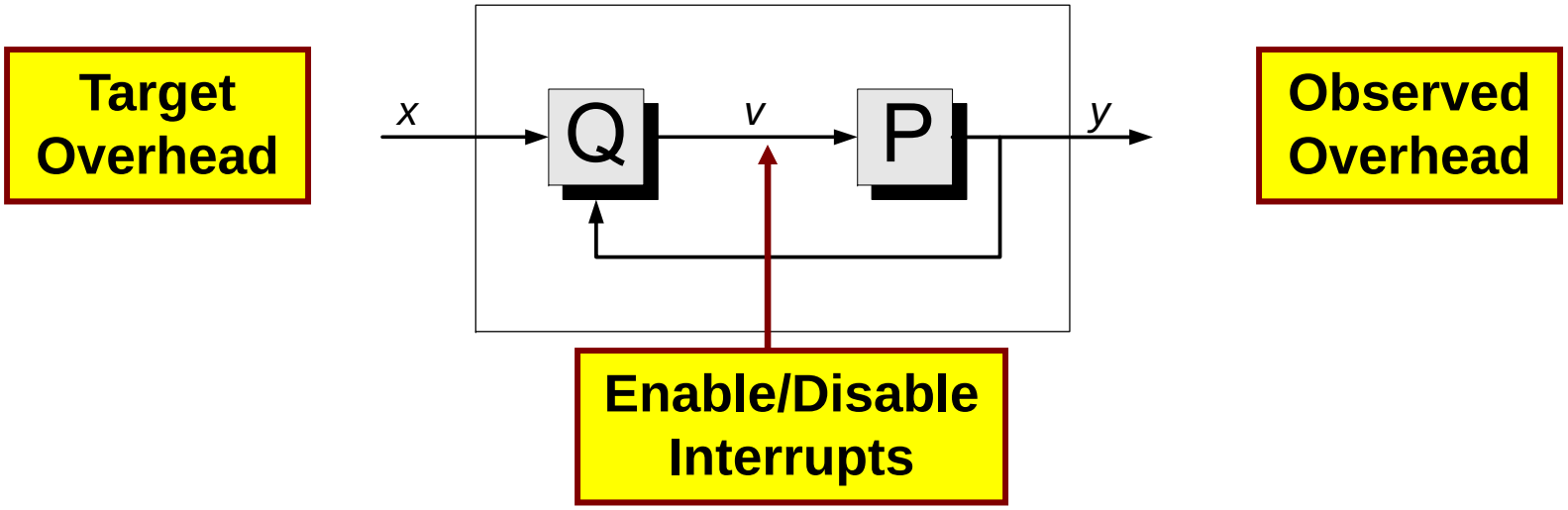


- User chooses ***target overhead***
- Target is a tradeoff between overhead and monitoring coverage
- Control-theoretic approach maximizes utility of allowed overhead

SMCO: Control Theoretic Approach

- Runtime monitoring can be beneficially stated as a **controller design problem**
  - **Controller** is the runtime monitor
  - **Plant** is instrumented software application
  - **Reference input** is **target overhead** O_t
- **Overhead control realized by disabling monitoring of events** generated by plant, and hence avoid overhead associated with processing these events
- **Interrupts disabled for as short a time as possible** so that number of events monitored, under the constraint of O_t , is maximized

Global Controller Architecture



Controller **Q** regulates *input* **v** to *plant* **P** in such a way that P's *output* **y** adheres to a *reference input* **x** with good dynamic response and small error

Preliminaries

- **Monitoring Overhead**

  If an unmodified and ***unmonitored*** program executes in time ***R***, and the ***monitored*** version executes in total time ***R+P***, we say that the monitoring has ***overhead P/R***

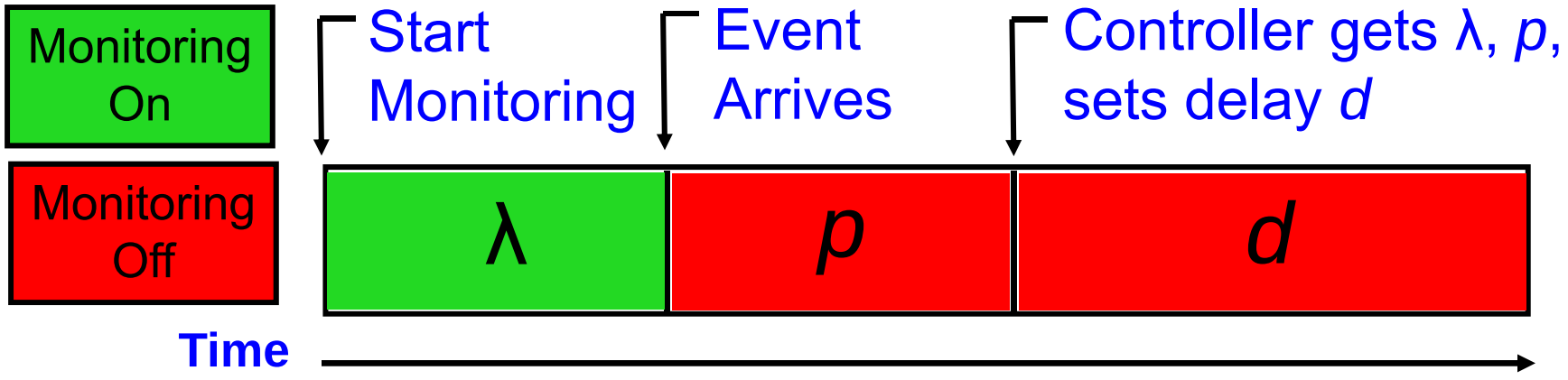
- **Monitoring Percentage** $m = P/(R + P)$

- Monitoring percentage m related to traditional definition of overhead o by $m = o/(1 + o)$

- **User-specified target monitoring percentage** m_t derived from o_t similarly: $m_t = o_t/(1 + o_t)$

SMCO Control Loop

- Controller spends $c = \lambda + p + d$ **total time** in each cycle (iteration of its control loop):

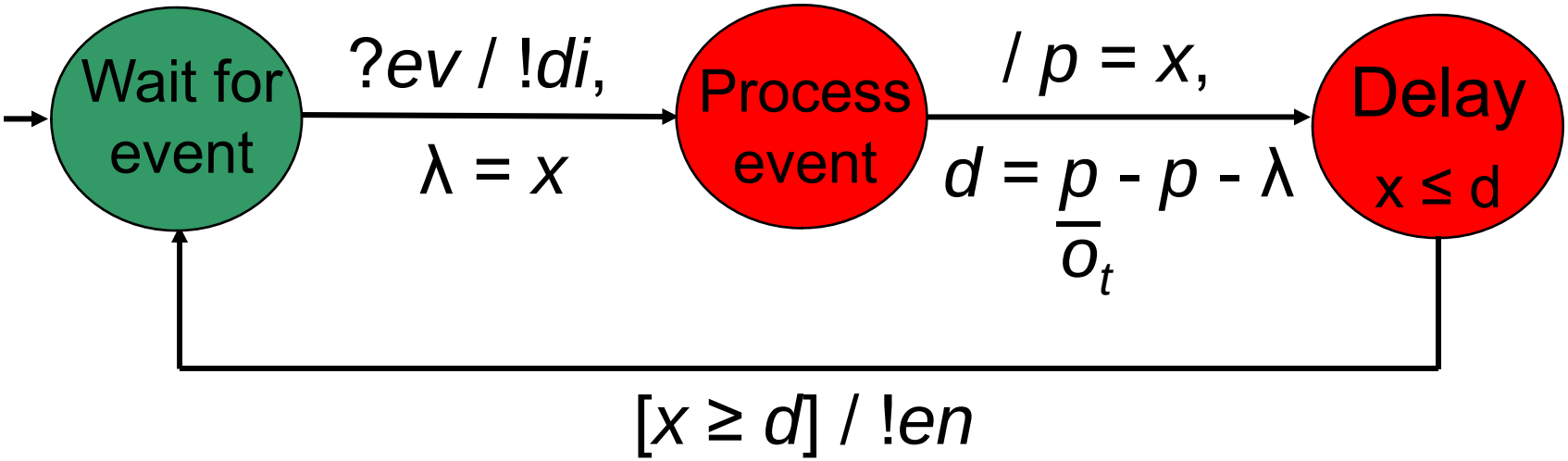


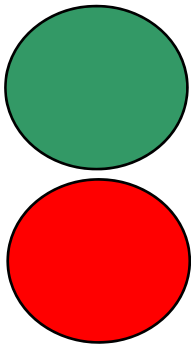
- Per-cycle **monitoring percentage** $m = p/c$
- Set $d = p/m_t - p - \lambda$ so that $m = m_t$

Controller adjusts ***sampling rate*** $1/d$ to be inversely proportional to object's ***access rate*** $1/\lambda$

Control Loop as a Timed A

x is a ***clock variable*** that is reset after every transition





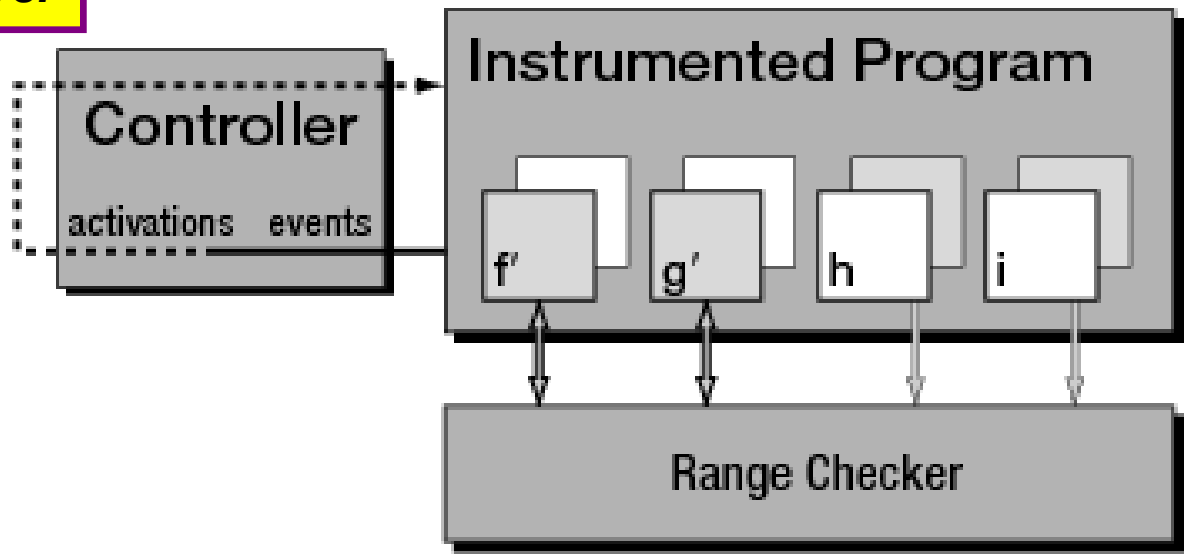
= Monitoring On

= Monitoring Off

ev = event
di = disable interrupts
en = enable interrupts

Integer Range Analysis

Range Solver

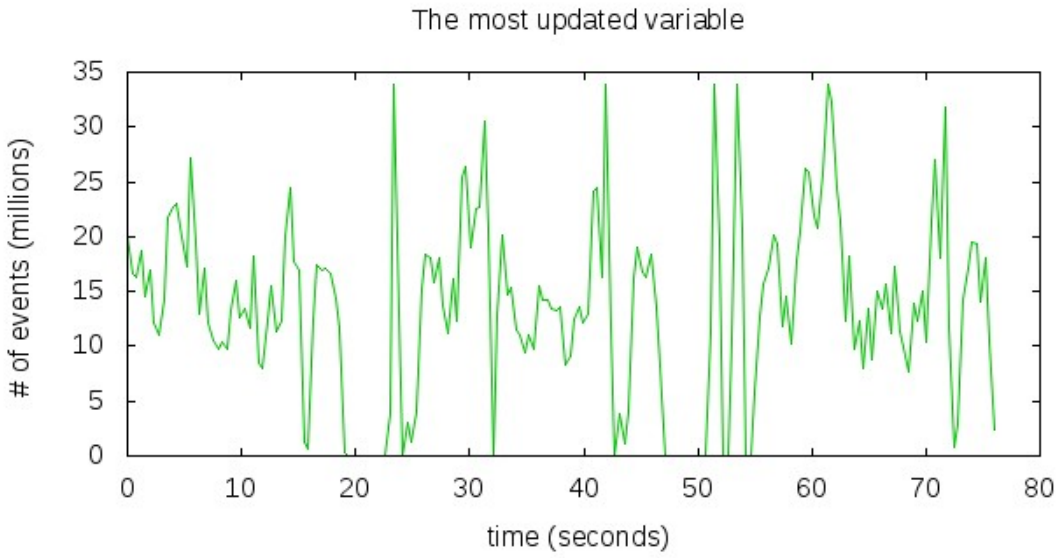


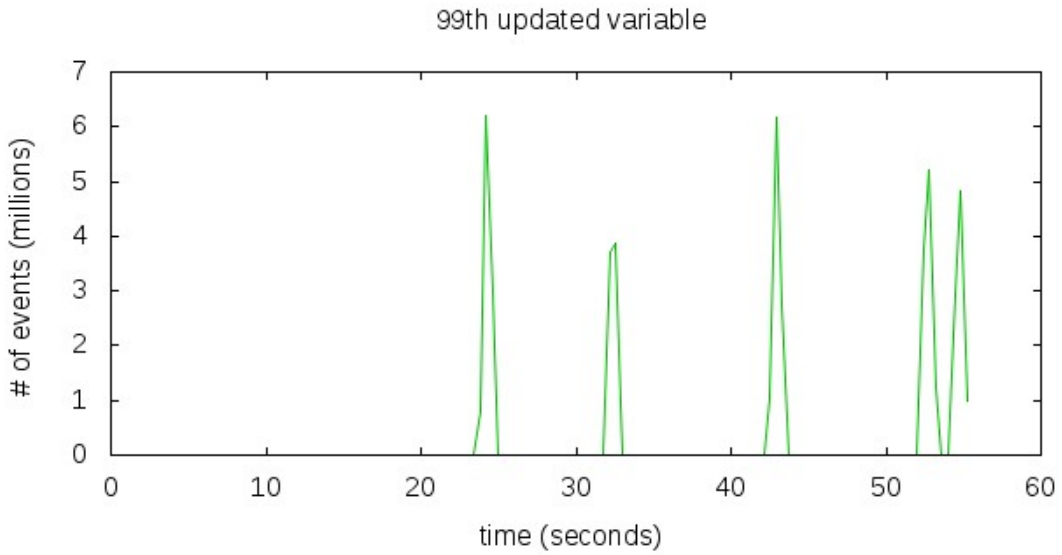
- Track range (min and max values) for every integer variable
- Values outside usual range are incriminating
- E.g., out-of-range index could indicate ***array-bounds violation***

Range Solver Workload

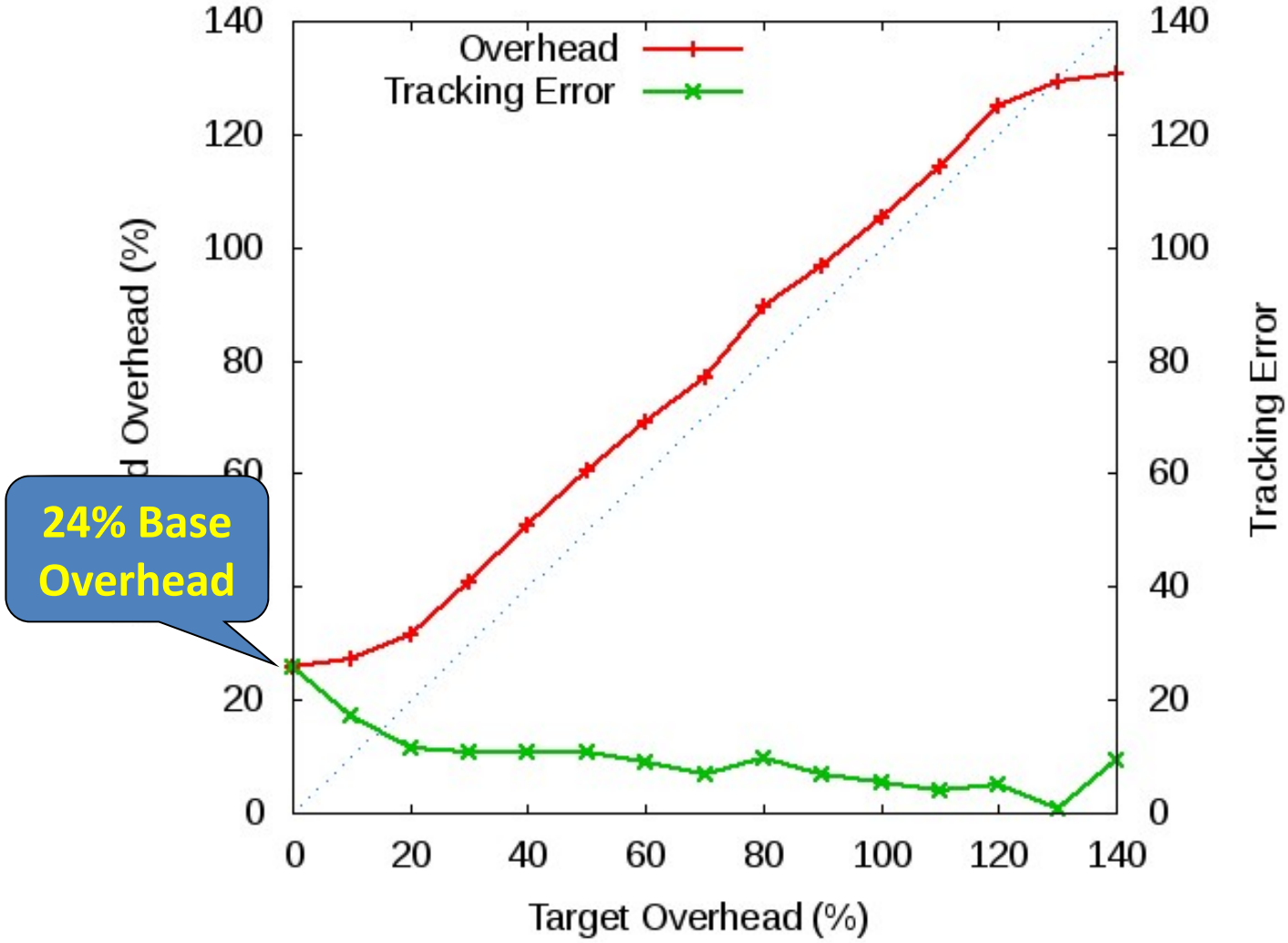
- ***bzip2*** data-compression utility from SPEC CPU2006 benchmark suite, run on ***53 MB*** "standard" data file
- bzip2 has 80 ***functions***, 61 of which contained integer assignments
- 445 ***integer variables***, 242 of which were modified during execution.
- ***Least-updated variables*** assigned only 1-2 times during a run
- ***Most updated variable*** assigned 2.5 billion times

Bursty and Variable Behavior

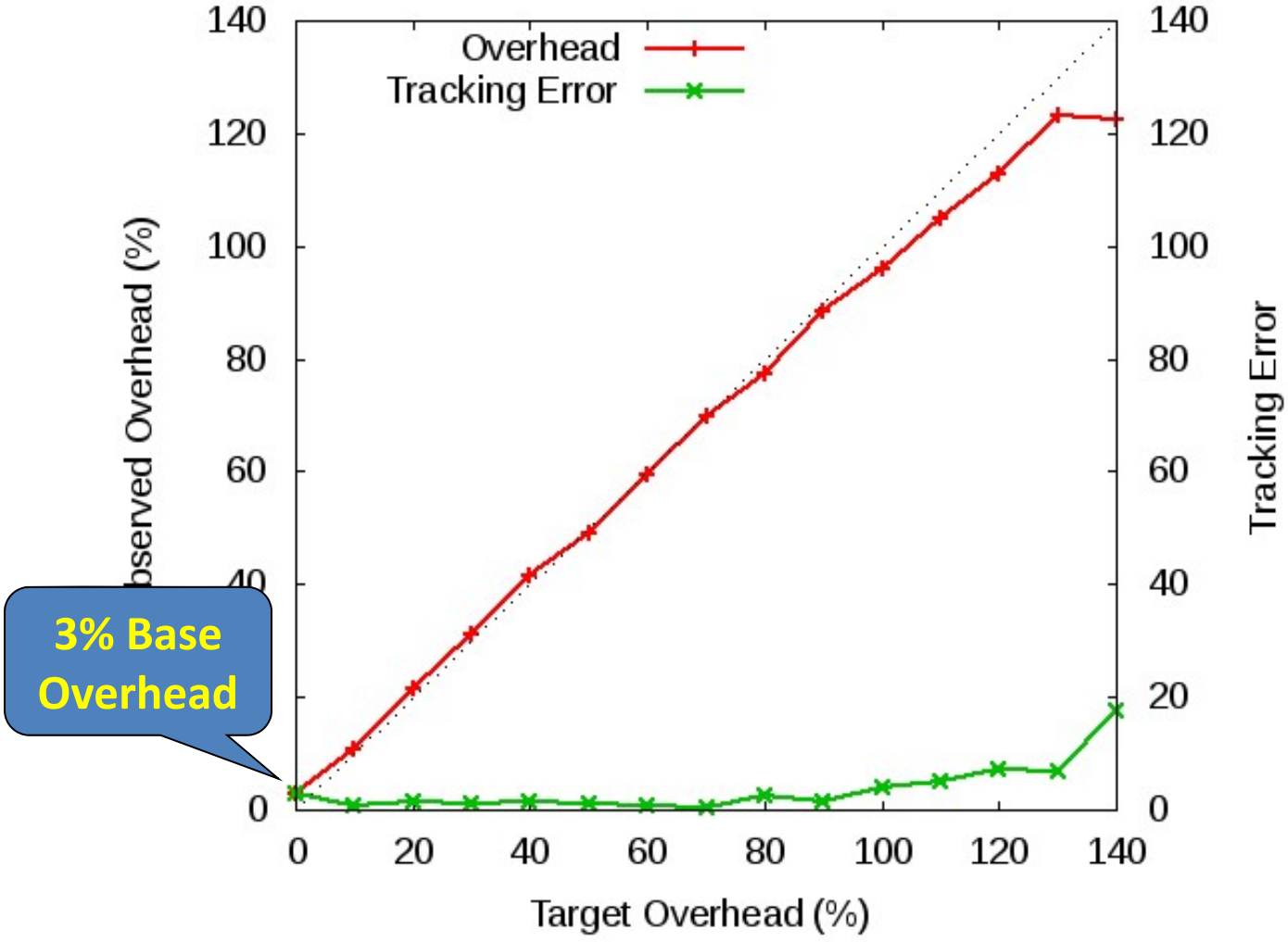




Range-Solver Benchmark *Before* Base-Overhead Optimizations on bzip2



Range-Solver Benchmark *After* Base-Overhead Optimizations on bzip2



Lowering Base Overhead: Clock Thread

Source of overhead:

- Frequent ***rdtsc*** instruction calls (45 CPU cycles per call on average) brings large base overhead

Our solution: ***Clock Thread***

- Spawn separate ***clock thread*** to handle timekeeping
- Clock thread periodically (usually 10Hz is enough) calls ***rdtsc*** and stores result in in-memory variable
- Main thread uses this variable as an ***inaccurate clock***

Lowering Base OH: Function Duplication

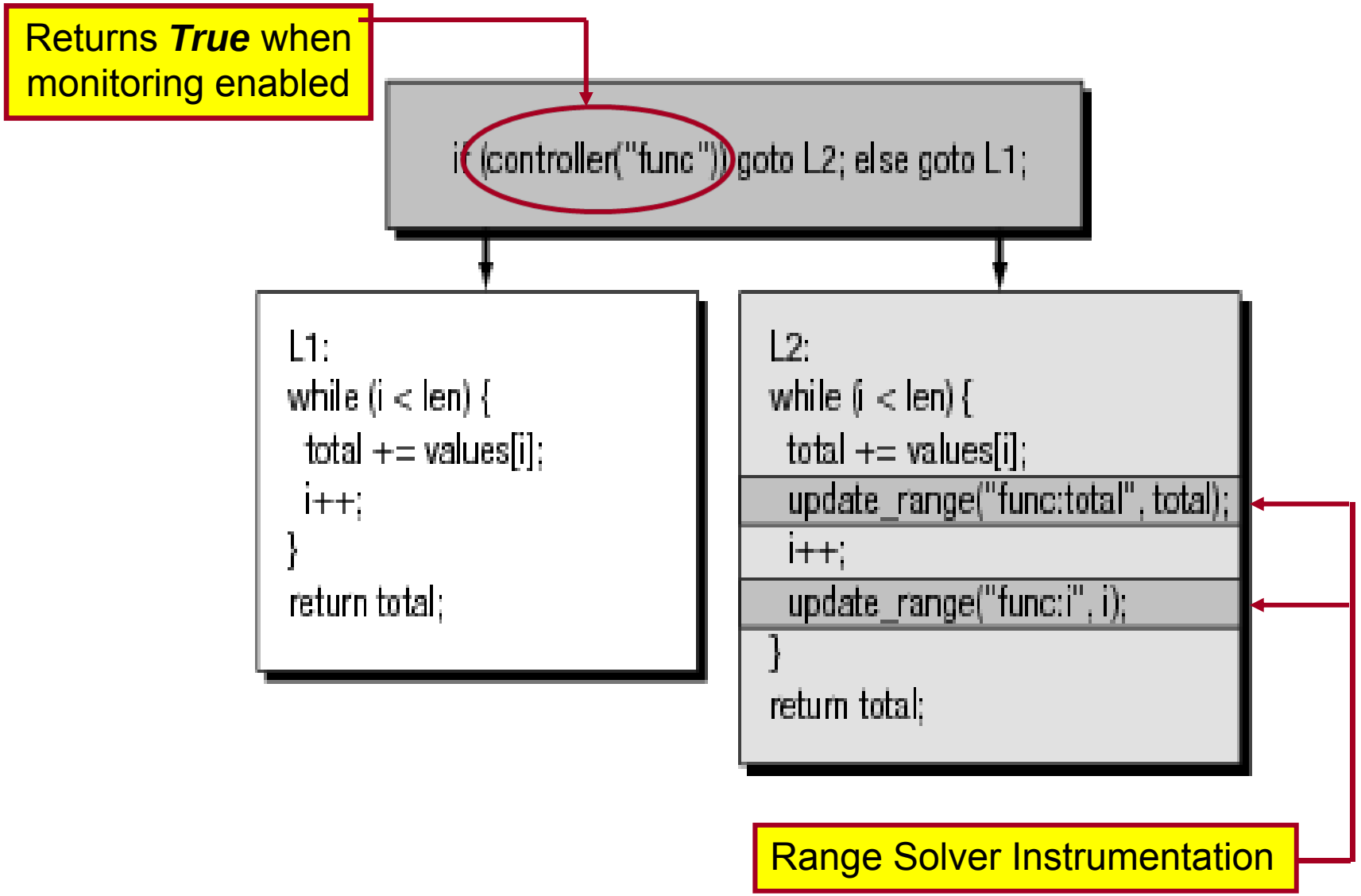
Source of overhead:

- In-lined monitoring statements waste time when monitoring is turned off

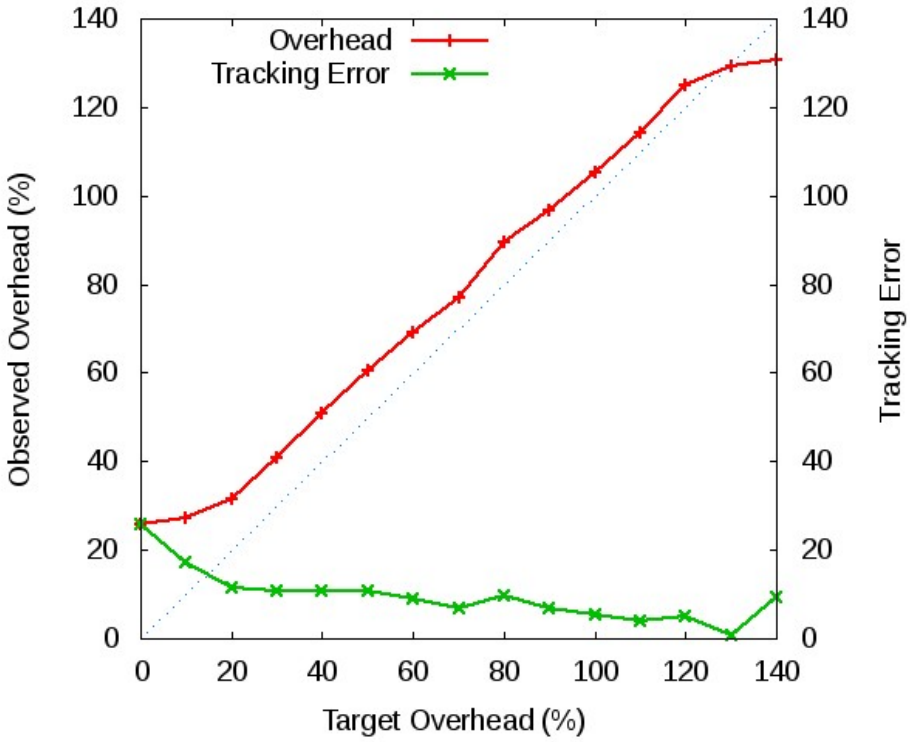
Our solution: ***Function body duplication***

- Duplicate function body of every function that needs to be instrumented
- Only duplicate copy is instrumented
- ***Distributor block*** passes control to original or instrumented copy of function body

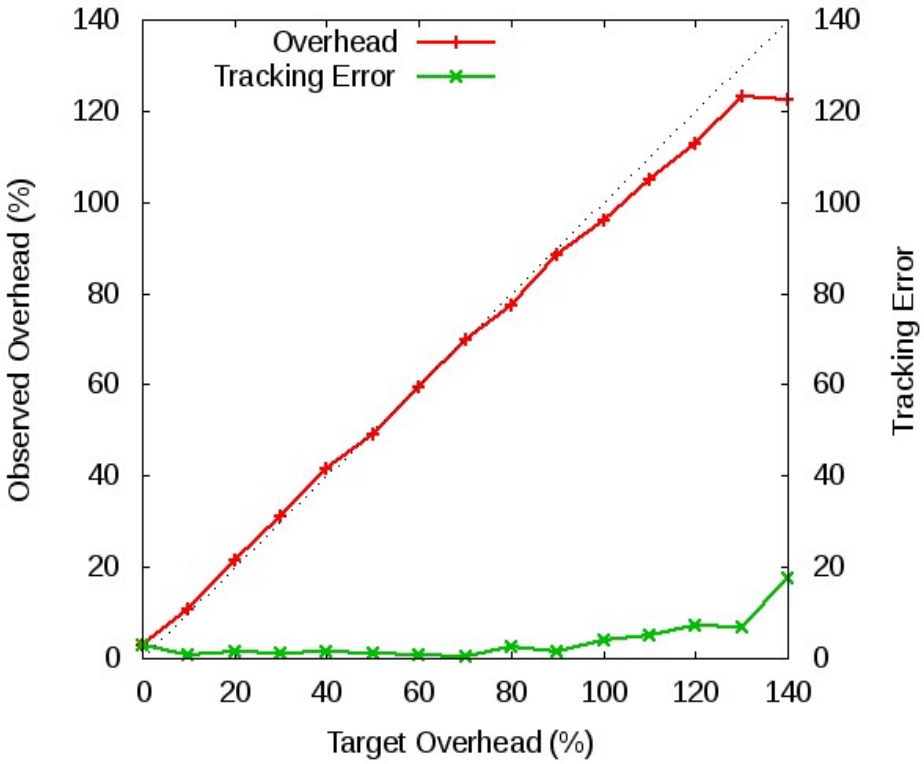
Function Body Duplication



Range-Solver Benchmark Before/After



Benchmark before Base-Overhead Optimizations



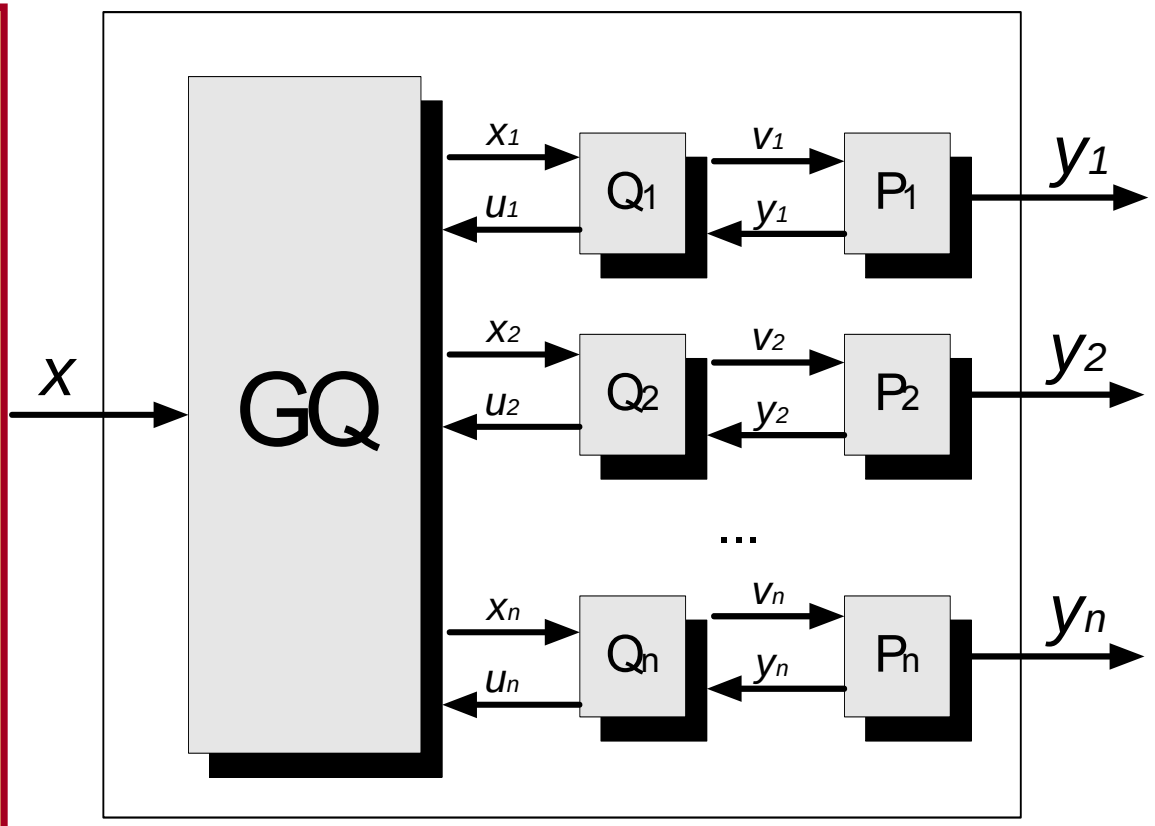
Benchmark with Clock-Thread and Function Duplication

Cascade Controller

Q_i's may not achieve target overhead if P_i's not sufficiently active

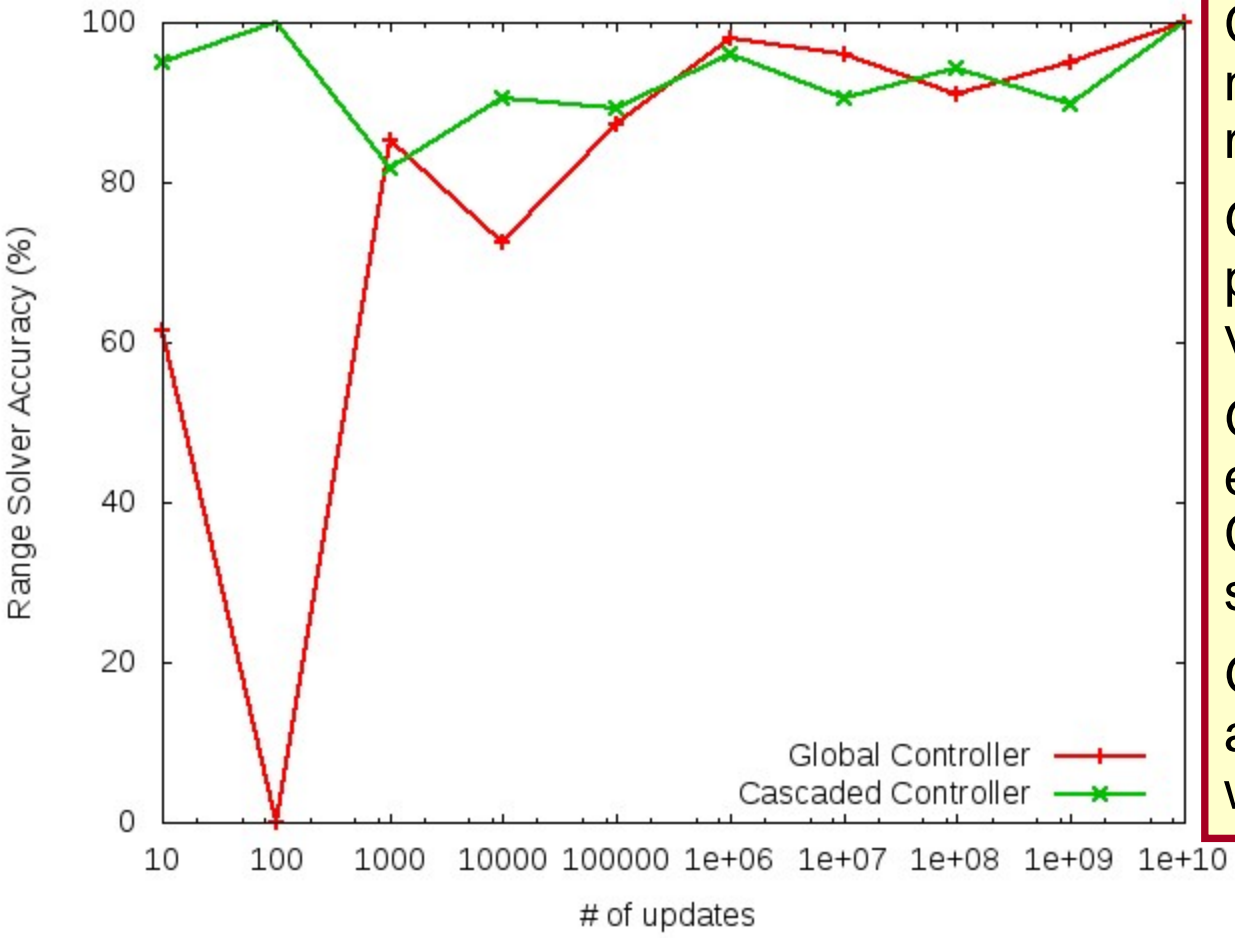
GQ periodically adjusts local target overhead x_i of Q_i's

Adjustment interval of **T** secs



Primary CTLR | **Secondary CTLRs** | **Software Plants**

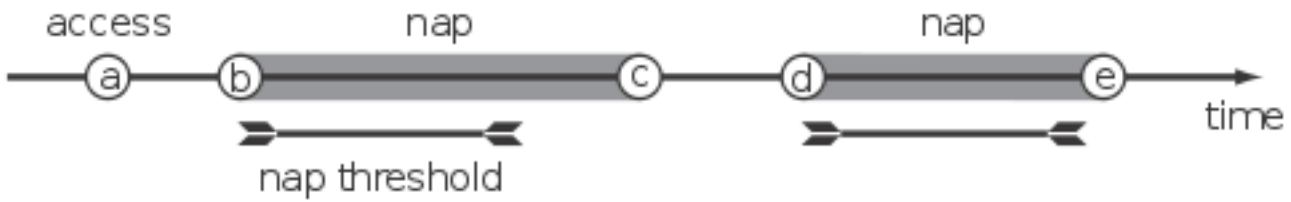
Range Solver *Accuracy* for Both Controllers with bzip2 Workload



- Cascade ctlr has notion of fairness (favors rare event sources)
- Global ctlr may perform poorly for rarely updated variables
- Global ctlr monitors more events in total than Cascade CTLR with same target overhead
- Global ctlr has better accuracy for variables with > 10^6 updates

NAP Detector

- ***NAP*** (Non-Accessed Period): memory allocations not accessed for periods of time whose duration equals or exceeds a ***user-specified threshold***



NAPs can vary in length, and multiple NAPs can be reported for the same allocation

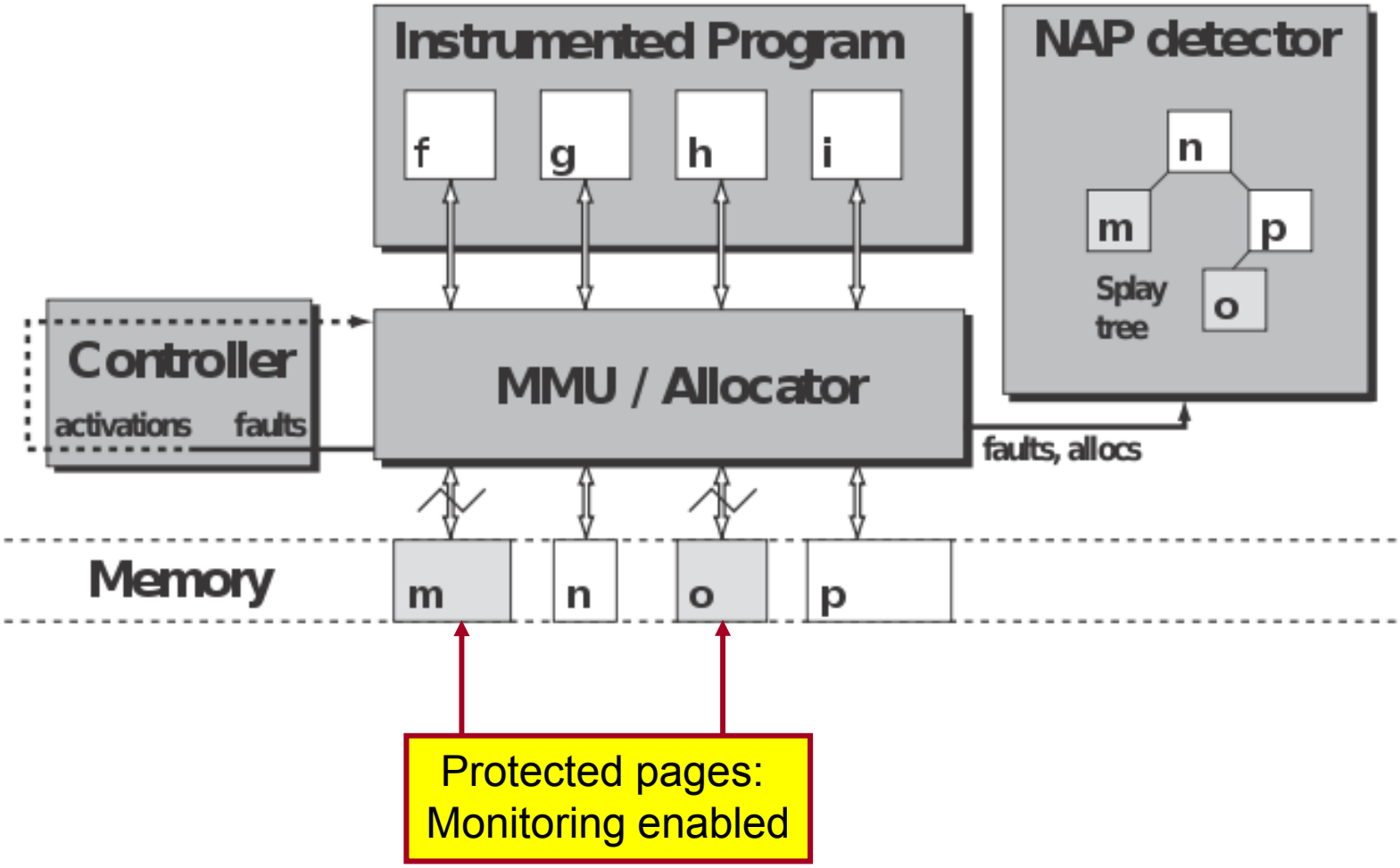
NAP Detector Controller

- Cascade controller used for ***per-allocation*** control of monitoring
- To enable monitoring for an allocation, controller calls ***mprotect*** to turn on read and write protections for the allocation
- Any access will result in a ***page fault***, which will be intercepted by NAP Detector and interpreted as an ***access event***

NAP Detector Controller (cont.)

- Secondary controller processes an event by removing area's read/write protections
- It then computes a delay ***d***, after which time monitoring ***re-enabled***
- ***Background thread*** periodically checks monitored areas for ***NAP condition***
  - Needed for areas that are never accessed!
- ***NAP condition*** also checked by secondary controller during event handling

NAP Detector Architecture



This Talk

- *SMCO*: Software Monitoring with Controllable Overhead
  - *Cascade* and *Global* controllers
  - *Base overhead* and *overhead tracking* optimizations
  - *Integer range analysis* and *leak detection*
- ***InterAspect:*** Aspect-oriented instrumentation for GCC
  - ***Plug-in architecture*** for GCC
  - InterAspect ***API***
  - ***Function body duplication***

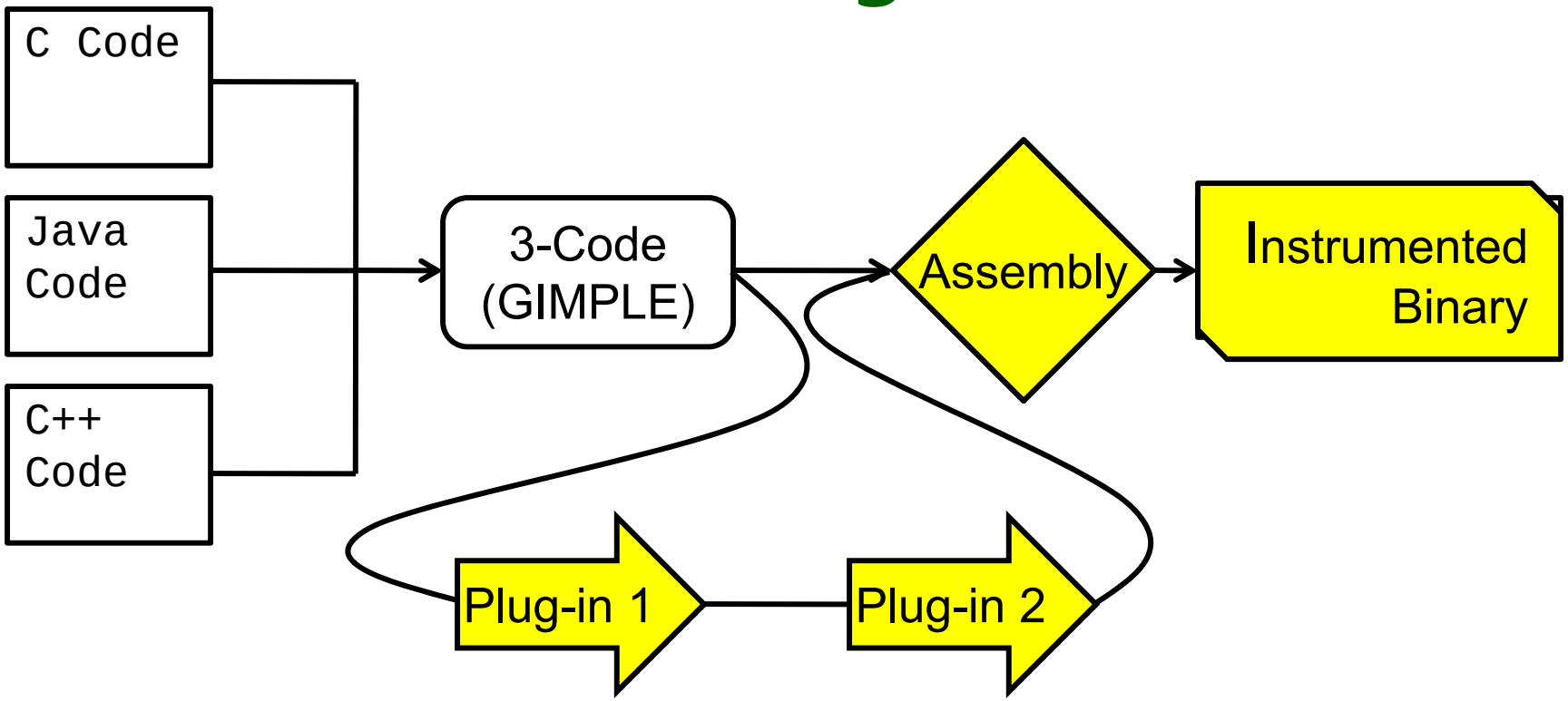
Compiler-Assisted Instrumentation

- Instrumentation occurs at compile time
- Provides direct access to all of compiler's analyses (type information, data-flow analysis)
- But modifying a production compiler is unwieldy
  - Long GCC recompile cycles
  - No way to mix and match compiler transformations from various sources

Plug-In Architecture for GCC

- GCC plug-ins operate on compiler's intermediate representation
- Plug-ins can analyze and instrument optimized production code
- Plug-ins are small and self-contained (fast to compile, easy to deploy)

GCC Plug-ins



Plug-ins ***dynamically loaded*** & can directly modify GCC's internal representation (***GIMPLE***, a three-address code)

Initial GCC plug-in API developed at Stony Brook in 2007

Original GCC Plugin Proposal

In: **2007 GCC Developer Summit**

Extending GCC with Modular GIMPLE Optimizations

Sean Callanan, Daniel J. Dean, and Erez Zadok

Abstract

We present a system of plug-ins for GCC that allows GCC to load GIMPLE transformations at run-time. This system reduces the support effort required for GCC by separating transformations from the core compiler. It also makes it possible for developers not connected with the GCC project to develop and distribute transformations independently. We demonstrate two plug-ins we have developed with this system, one of which …

GCC 4.5.0 released on 4/14/10!

Major new feature: Plugins

- Now possible to extend the GCC compiler without having to modify its source code
- New option *-fplugin=file.so* tells GCC to load the shared object *file.so* and execute it as part of the compiler
- *gcc -fplugin=plugin1.so -fplugin=plugin2.so* …

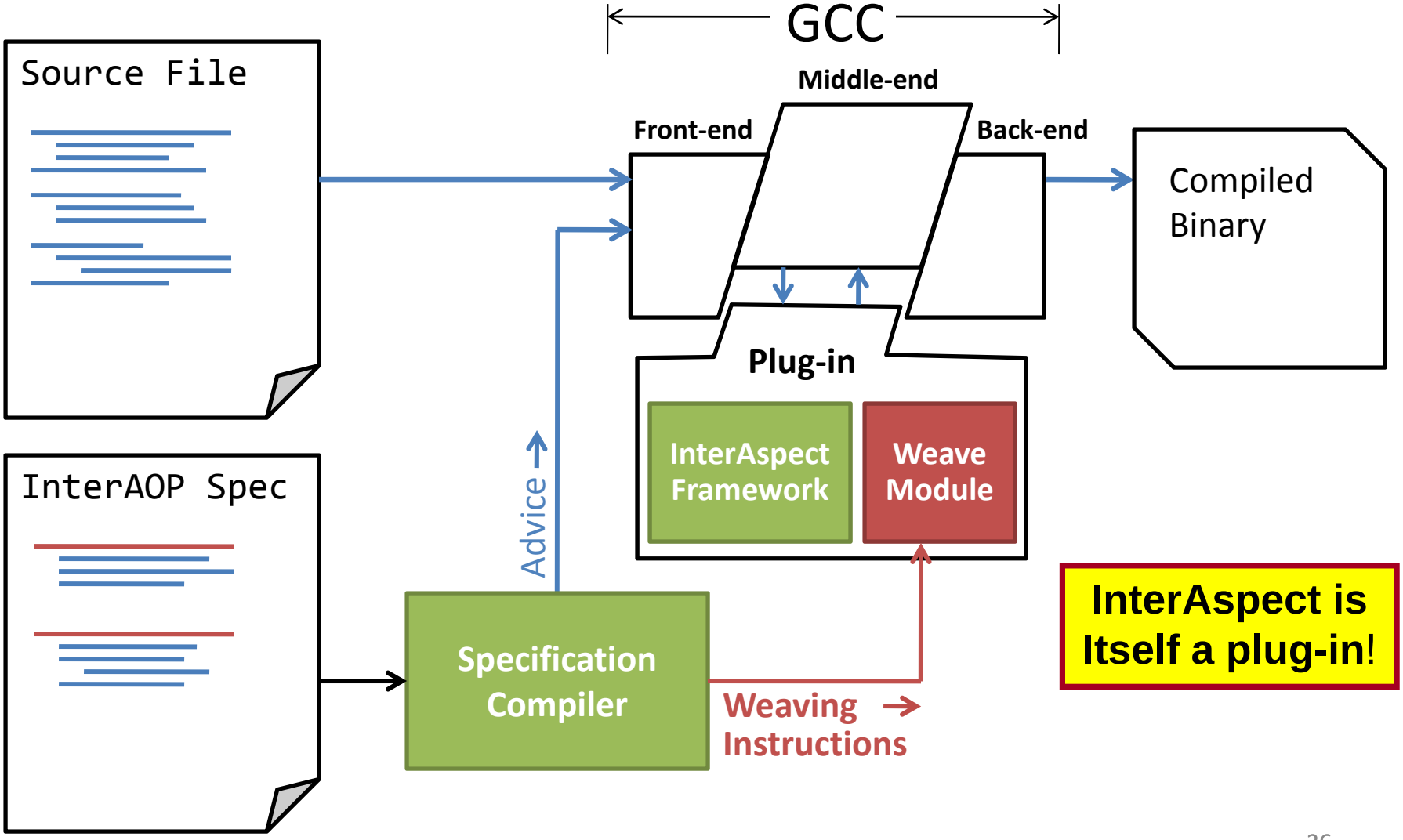
InterAspect

- AOP framework for building GCC plug-ins that perform ***custom instrumentation***
- Operates at level of GCC's intermediate representation (GIMPLE) ***=> Can instrument code written in any language GCC supports***
- Instrumentation sites specified by ***pointcuts*** (function entry/return, function calls, variable assignments) ***=> No specialized knowledge of GCC internals needed!***
- Aspects can include ***weave-time*** code that specializes the code inserted at each site ***=> Existing AOP frameworks do not support this***

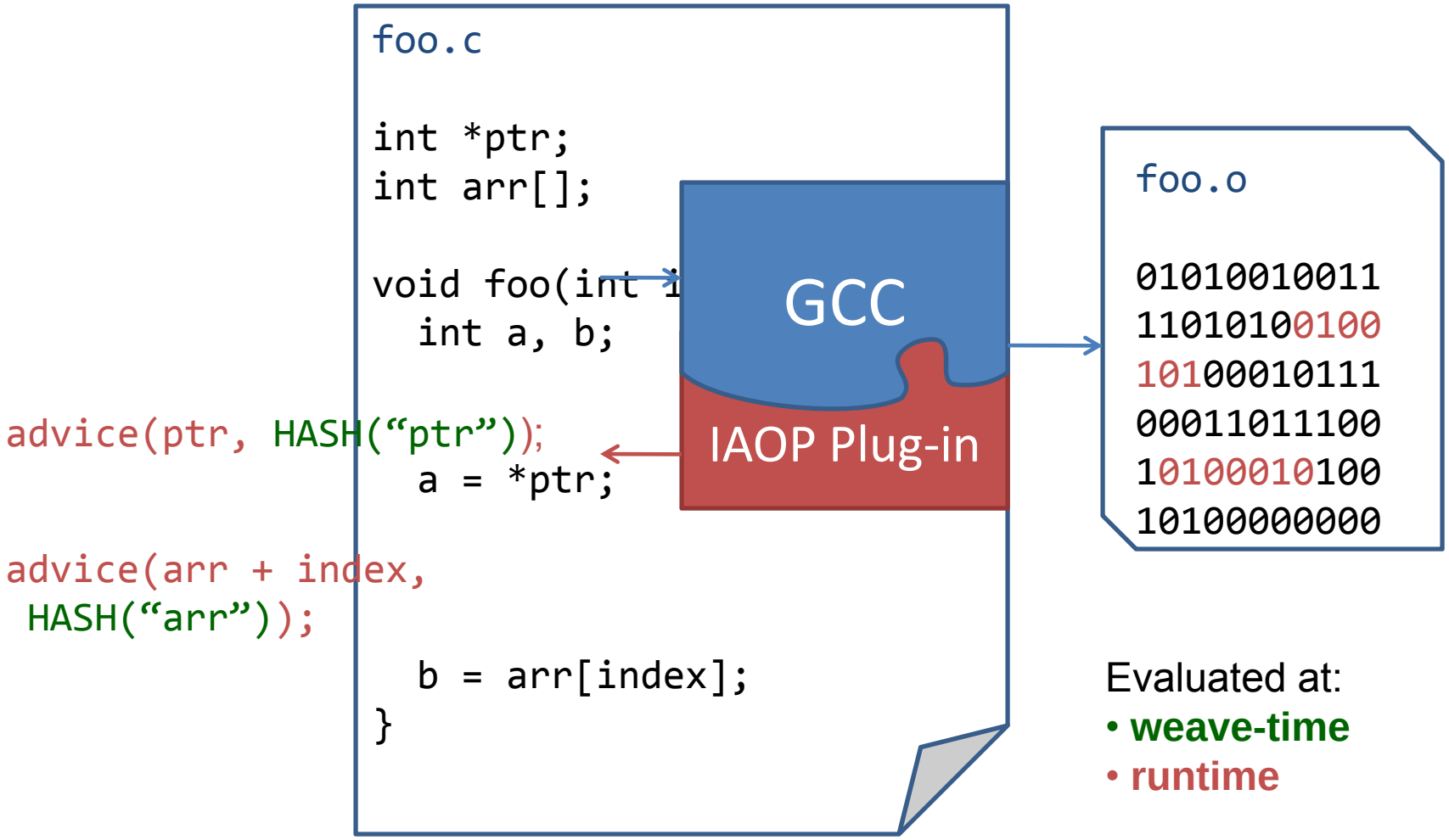
Programming with InterAspect

- Pointcut = set of ***joinpoints***
- ***Customized Instrumentation***: a plug-in iterates through a pointcut by providing a ***callback*** that executes ***once per joinpoint***
- For each joinpoint, callback can choose which ***advice function*** (if any) should be called there and which joinpoint info to pass to advice
  - arguments and return value for a ***function-call JP***
  - address of assigned variable for an ***assignment JP***

InterAspect Architecture



GCC Customizable Aspect Weaving



Pointcuts & Callbacks

- Use ***callback*** for joinpoint-specific code weaving

```
// Define pointcut and invoke callback on joinpoints
void instrumentation_pass()
{
  struct aop_pointcut pc;
  pc = aop_match_assignment_by_type(aop_t_all_signed());

   // Invoke callback on all assignments in pointcut
  aop_join_on(pc, assignment_callback);
}
```

Customize by Site

```
void assignment_callba                    oint *jp)
{
  const char *var_name                    lhs_name(jp);

  if (is_suspicious_var(var_name)) {
    // Instrument assignment to suspicious variable
    aop_insert_advice(jp, "advice", ... /* args */);
  }
  else {
    // Leave this assignment alone
  }
}
```

Result of static analysis

Conclusions

- ***InterAspect*** allows one to perform custom instrumentation using GCC plug-ins
- ***SMCO*** allows one to tightly control the overhead such instrumentation generates

Future Work

- Extend SMCO with ***software plant model***
- ***Trace visualization***: instrument Rover code with InterAspect and visualize event traces
- ***Specification learning***: mine event traces

For More Info

- Please see http://www.fsl.cs.sunysb.edu/ssw/