The Global Control-Flow Graph

Optimizing an Event-Driven Real-Time System Across Kernel Boundaries

Christian Dietrich, Martin Hoffmann, Daniel Lohmann

{dietrich,hoffmann,lohmann}@cs.fau.de

Friedrich-Alexander University
Erlangen-Nuremberg

12. November 2015

supported by DFG

Compiler Optimization on Function Level

```c
void compute(int a[], int len, int val) {
  for (int i = 0; i < len; i++) {
    a[i] = a[i] + val + 1000;
  }
}
```

Compiler Optimization on Function Level

```
void compute(int a[], int len, int val) {
  for (int i = 0; i < len; i++) {
    a[i] = a[i] + val + 1000;
  }
}
```

Calculated in each Iteration (val + 1000)

Compiler Optimization on Function Level

```c
void compute(int a[], int len, int val) {
  for (int i = 0; i < len; i++) {
    a[i] = a[i] + val + 1000;
  }
}
```

Calculated in each Iteration (`val + 1000`)

Loop-Invariant Code Motion

```c
void compute(int a[], int len, int val) {
  int temp = val + 1000;
  for (int i = 0; i < len; i++) {
    a[i] = a[i] + temp;
  }
}
```

Compiler Optimization on Program Level

```c
int lastVal, data[2];
void compute(int a[], int len, int val) {
    int temp = val + 1000;
    ...
}
void Task1() {
   compute(data, 2, lastVal);
}
```

Compiler Optimization on Program Level

```
int lastVal, data[2];
void compute(int a[], int len, int val) {
    int temp = val + 1000;
    ...
}
void Task1() {
   compute(data, 2, lastVal);
}
```

Compiler Optimization on Program Level

```c
int lastVal, data[2];
void compute(int a[], int len, int val) {
    int temp = val + 1000;
    ...
}
void Task1() {
   compute(data, 2, lastVal);
}
```

Inlining and Loop Unrolling

```c
int lastVal, data[2];
void Task1(){
  int temp = lastVal + 1000;
  data[0] = data[0] + temp;
  data[1] = data[1] + temp;
}
```

Compiler Optimization on System Level (potential)

```
void Task1(){
  int temp = lastVal + 1000;
  data[0] = data[0] + temp;
  data[1] = data[1] + temp;
}
void Task2() {
  lastVal = 23;
  ActivateTask(Task1); // System Call
}
```

Compiler Optimization on System Level (potential)

```
void Task1(){
  int temp = lastVal + 1000;
  data[0] = data[0] + temp;
  data[1] = data[1] + temp;
}
void Task2() {
  lastVal = 23;
  ActivateTask(Task1); // System Call
}
```

Compiler Optimization on System Level (potential)

```
void Task1(){
  int temp = lastVal + 1000;
  data[0] = data[0] + temp;
  data[1] = data[1] + temp;
}
void Task2() {
  lastVal = 23;
  ActivateTask(Task1); // System Call
}
```

Constant Propagation across Kernel Boundaries

```
void Task1(){
  data[0] = data[0] + 1023;
  data[1] = data[1] + 1023;
}
void Task2() {/* unchanged */}
```

A System Model for the Compiler

- **Problem:** System-Calls are not transparent for the compiler
  - Compilers stay only within the language level
  - Possible operating-system decisions are not taken into account

A System Model for the Compiler

- **Problem:** System-Calls are not transparent for the compiler
  - Compilers stay only within the language level
  - Possible operating-system decisions are not taken into account
- **Solution:** We supply an OS execution model
  - Knowledge about application–OS interaction
  - Execution model includes possible scheduling decision
  - System calls become more transparent for the compiler

A System Model for the Compiler

- **Problem:** System-Calls are not transparent for the compiler
  - Compilers stay only within the language level
  - Possible operating-system decisions are not taken into account
- **Solution:** We supply an OS execution model
  - Knowledge about application–OS interaction
  - Execution model includes possible scheduling decision
  - System calls become more transparent for the compiler
- Especially useful for embedded real-time systems
  - Application and kernel are often statically combined
  - Precise OS execution model through determinism

Outline

- **Question 1:**
  How to gather OS exeuction model for a static real-time systems?

- **Question 2:**
  How to utilize the gathered information?

Outline

- **Question 1:**
  How to gather OS exeuction model for a static real-time systems?
  ↳ Global Control-Flow Graph

- **Question 2:**
  How to utilize the gathered information?

Event-Triggered Static Real-Time Systems

- Basic assumptions for our system-level analysis
  - Event-triggered real-time systems: execution threads, interrupts, etc.
  - Static system design: fixed number of threads, fixed priority
  - Deterministic system-call semantic and scheduling
  - System-calls are fixed in location and arguments

Event-Triggered Static Real-Time Systems

- Basic assumptions for our system-level analysis
  - Event-triggered real-time systems: execution threads, interrupts, etc.
  - Static system design: fixed number of threads, fixed priority
  - Deterministic system-call semantic and scheduling
  - System-calls are fixed in location and arguments

- Assumption apply to a wide range of systems: OSEK, AUTOSAR
  - Industry standard widely employed in the automotive industry
  - Static configuration at compile-time

Example Application

Static System Configuration

```
TASK TaskA {
   PRIORITY = 0;
   AUTOSTART = TRUE;
};
```

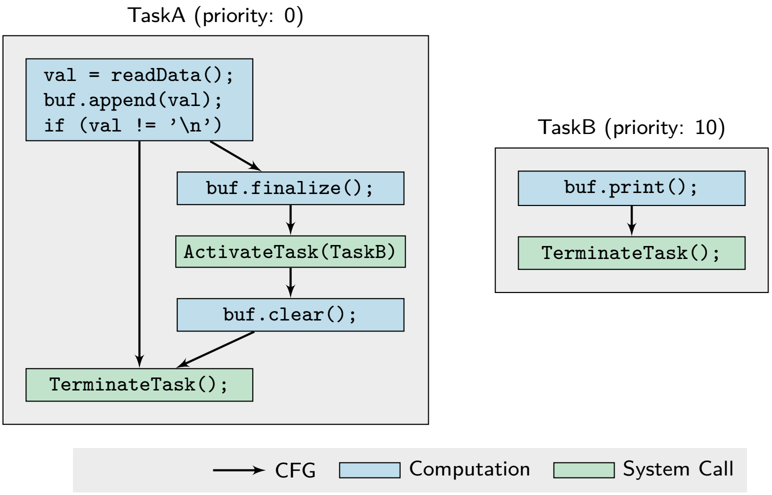
```
TASK TaskB {
   PRIORITY = 10;
};
```

Application Code

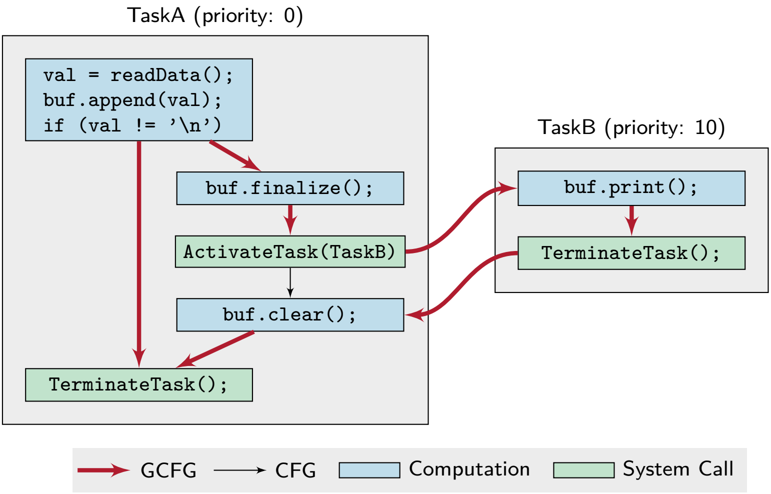
```
void TaskA() {
  int val = readData();
  buf.append(val);
  if (val != '\n') {
     buf.finalize();
     ActivateTask(TaskB);
     buf.clear();
  }
  TerminateTask();
}
```

```
void TaskB() {
   buf.print();
   TerminateTask();
}
```

Control-Flow Graph



Global Control-Flow Graph (GCFG)



GCFG and System State Enumeration

- The GCFG contains all possible scheduling decisions
  - GCFG is OS specific
  - GCFG is application specific

GCFG and System State Enumeration

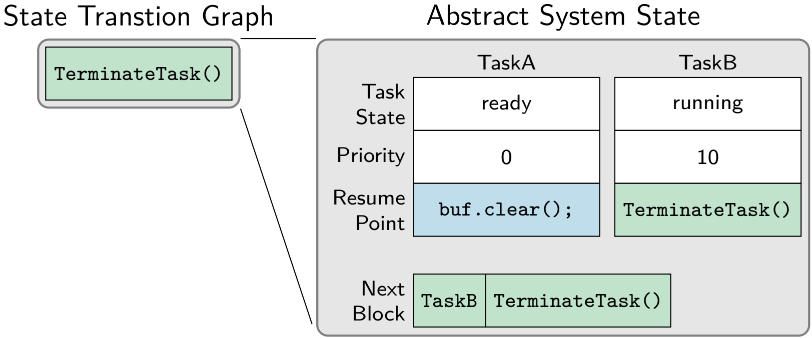
- The GCFG contains all possible scheduling decisions
  - GCFG is OS specific
  - GCFG is application specific

- Combine three information sources in System-State Enumeration
  - System specification
  - Static system configuration
  - Application structure from control-flow graphs

GCFG and System State Enumeration

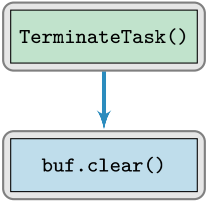
- The GCFG contains all possible scheduling decisions
  - GCFG is OS specific
  - GCFG is application specific
- Combine three information sources in System-State Enumeration
  - System specification
  - Static system configuration
  - Application structure from control-flow graphs
- Basic principle of system-state enumeration
  - Instantiate abstract OS model with system configuration
  - Simulate the application structure on top of the OS model
  - Discover all possible system states

System-State Enumeration and the Transition Graph



System-State Enumeration and the Transition Graph

State Transtion Graph



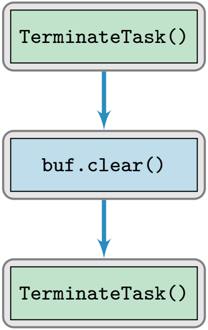
Abstract System State

	TaskA	TaskB
Task State	running	suspended
Priority	0	10
Resume Point	`buf.clear();`	–

Next Block	`TaskA`	`buf.clear()`

System-State Enumeration and the Transition Graph

State Transtion Graph



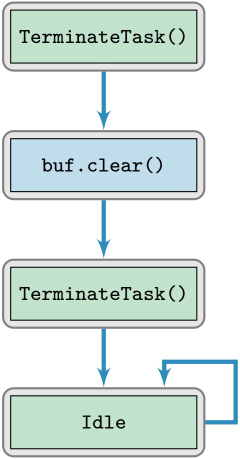
Abstract System State

	TaskA	TaskB
Task State	running	suspended
Priority	0	10
Resume Point	`TerminateTask()`	–

Next Block	`TaskA`	`TerminateTask()`

System-State Enumeration and the Transition Graph

State Transtion Graph

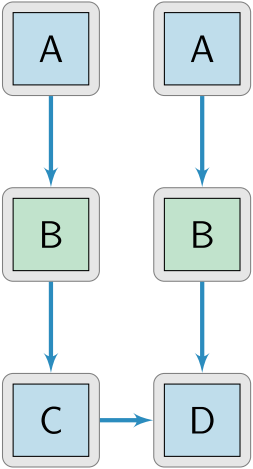


Abstract System State

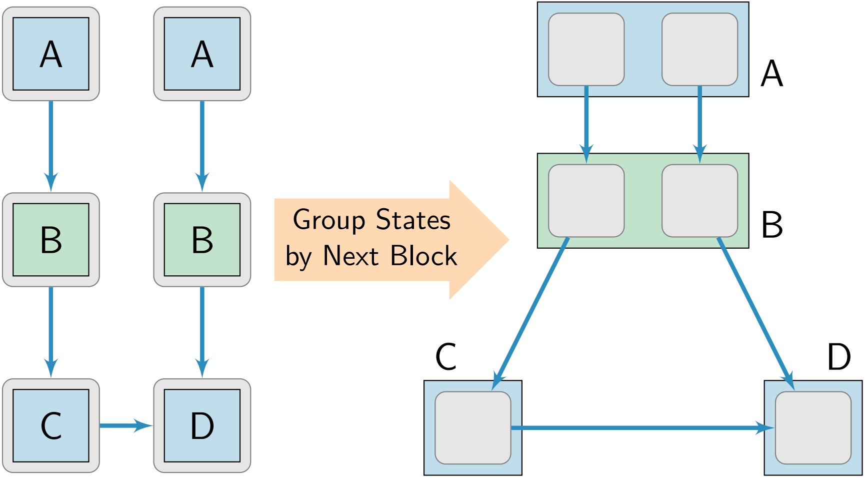
	TaskA	TaskB
Task State	suspended	suspended
Priority	0	10
Resume Point	–	–

Next Block	-	`Idle`

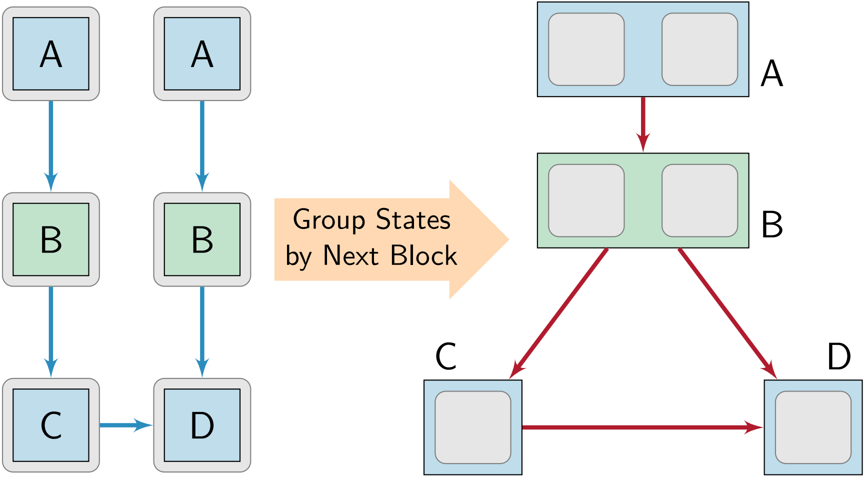
Transforming the State-Transition Graph



Transforming the State-Transition Graph



Transforming the State-Transition Graph



Outline

- **Question 1:**
  How to gather OS exeuction model for a static real-time systems?
  ↳ Global Control-Flow Graph ✓

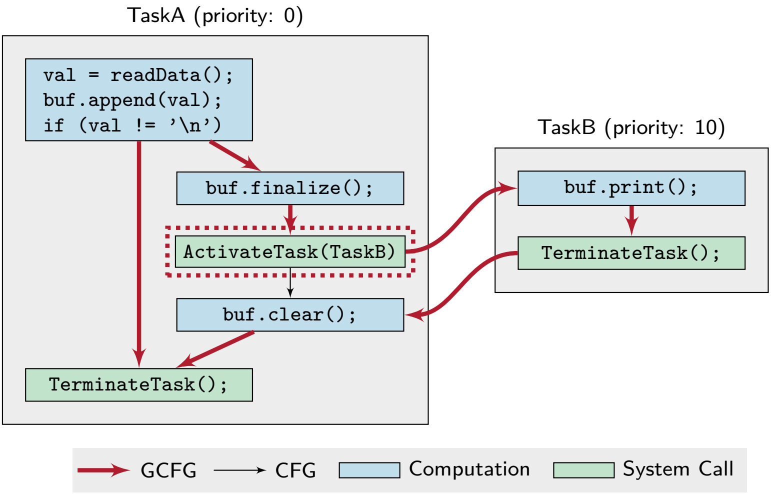
- **Question 2:**
  How to utilize the gathered information?

Outline

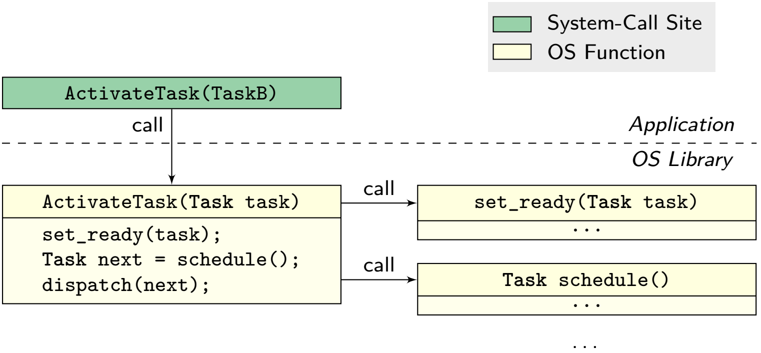
- **Question 1:**
  How to gather OS exeuction model for a static real-time systems?
  ↳ Global Control-Flow Graph ✓

- **Question 2:**
  How to utilize the gathered information?
  → Specialized System Calls
  → Assertions on the System State
  → Kernel as a Statemachine
  → ...

Control-Flow Graph

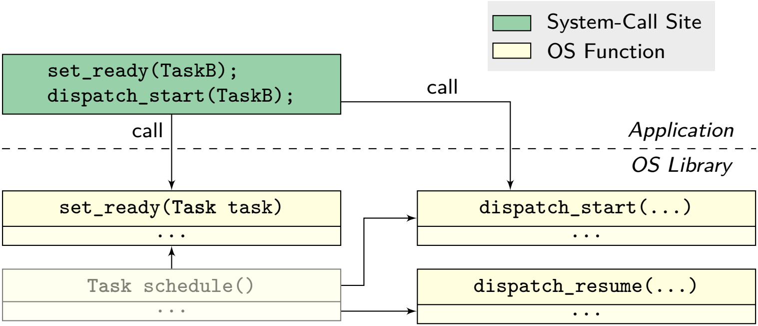


Traditional Library Operating System



- Dictate on generality: "One size fits all"
  - One system-call implementation for all system-call sites
  - System-call must be callable from anywhere
  - Code reuse saves flash memory

Specialized System Calls



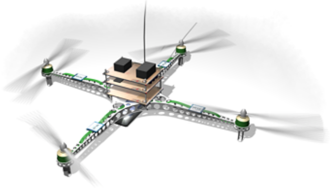
- Specialize each system-call site:
  - Strip out computation steps with predictable outcome
  - Trade-off between run time and code size
  - Outgoing edges in the GCFG are possible `schedule()` results.

Evaluation Scenario

- Evaluation System: *d*OSEK (*dependable* OSEK) *(RTAS'15)*
  - Fault-tolerant OSEK implementation for IA-32
  - Generative Approach

Evaluation Scenario

- Evaluation System: *d*OSEK (*dependable* OSEK) *(RTAS'15)*
  - Fault-tolerant OSEK implementation for IA-32
  - Generative Approach
- Scenario: Quadrotor Flight Control
  - 11 tasks, 3 alarms, 1 ISR
  - 53 system-call sites
  - Execute system for 3 hyperperiods



Outline

- **Question 1:**
  How to gather OS exeuction model for a static real-time systems?
  ↳ Global Control-Flow Graph ✓

- **Question 2:**
  How to utilize the gathered information?
  ↳ Specialized System Calls — Kernel Runtime: -30 %
  ↳ Assertions on the System State — Resilience Against Bitflips: +50 %
  ↳ Kernel as a Statemachine — FSM with 728 States
  ↳ ...

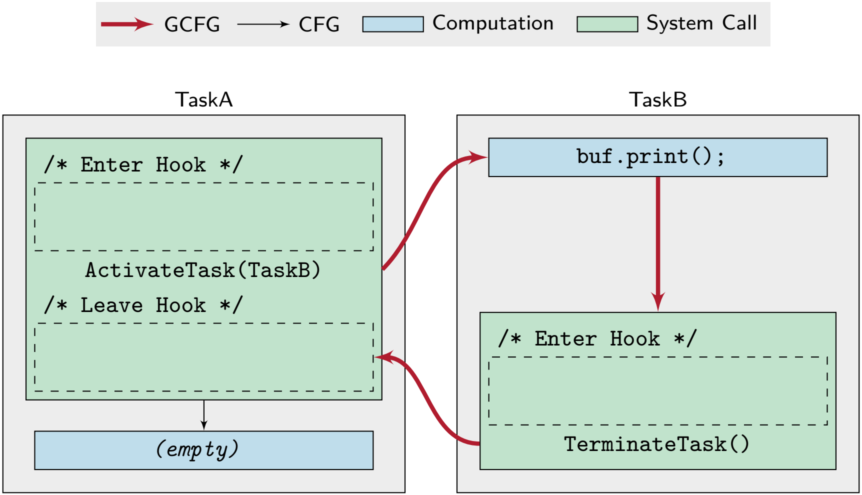
Conclusion and Future Work

> “*With Great Knowledge comes*
> *Great Optimization Potential* .
> – *SpiderGCC* ”

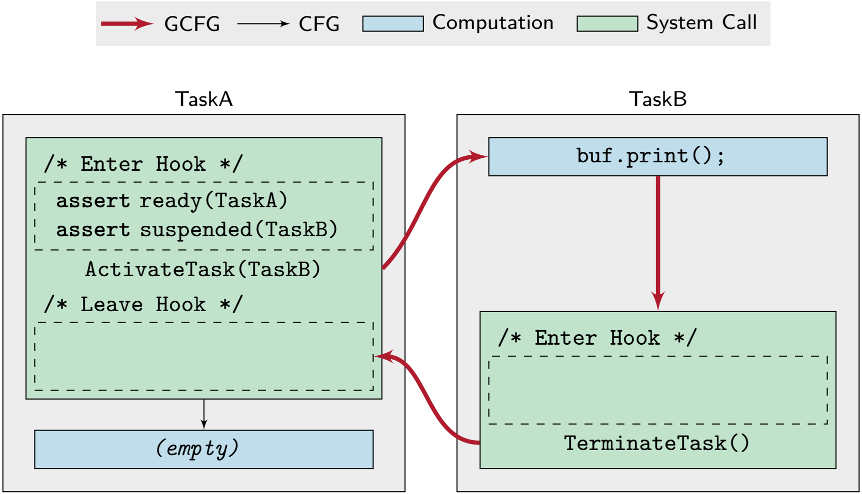
- Fine-Grained Analysis of Event-Triggered, Static Real-Time Systems
  - The Global Control-Flow Graph includes the application–OS interaction
  - Additional static knowledge from the state-transition graph
- Fine-Grained Tailoring of Application and Kernel
  - Reduction of kernel runtime by $\sim$ 30 percent
  - Monitoring of static system properties: $\sim$ 50 pecent smaller SDC rate
- Further Applications
  - Improve worst-case execution time analysis of whole applications
  - Replace Kernel by a State Machine ($\rightarrow$ OSPERT'15)

Source code available at `https://github.com/danceos/dosek`

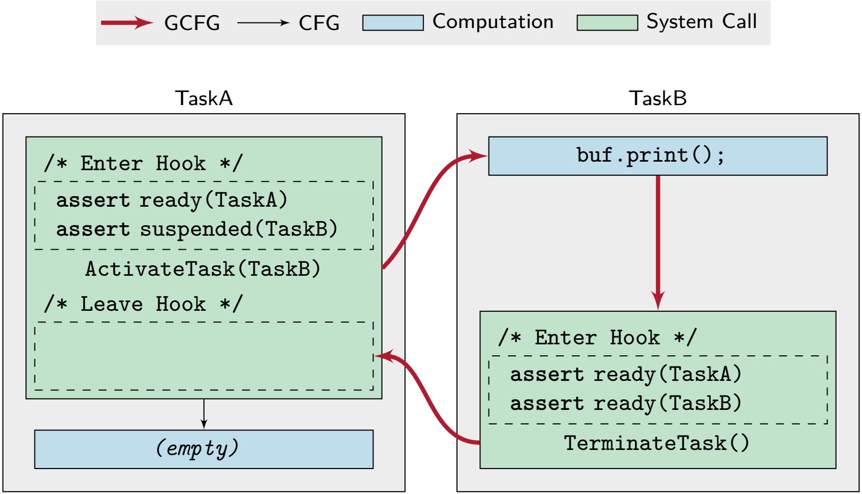
System State Assertions



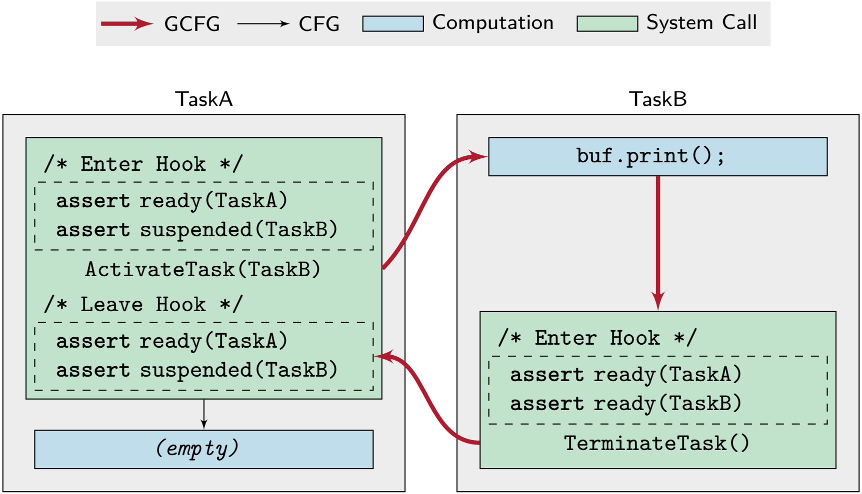
System State Assertions



System State Assertions

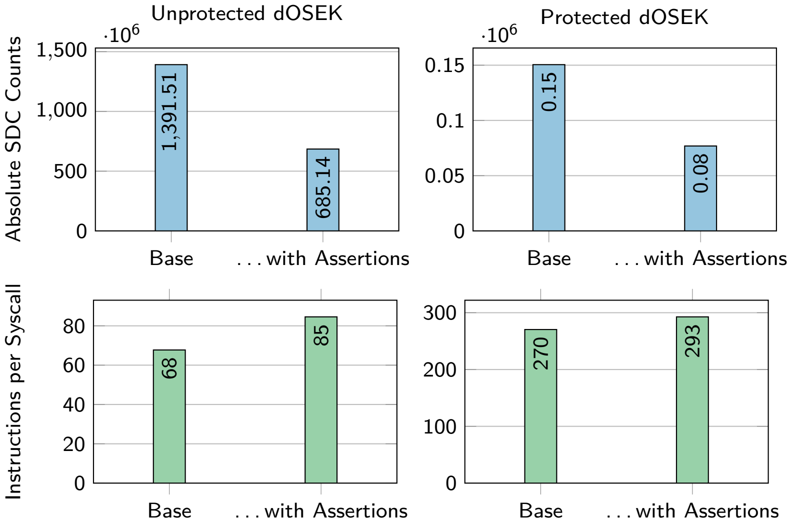


System State Assertions

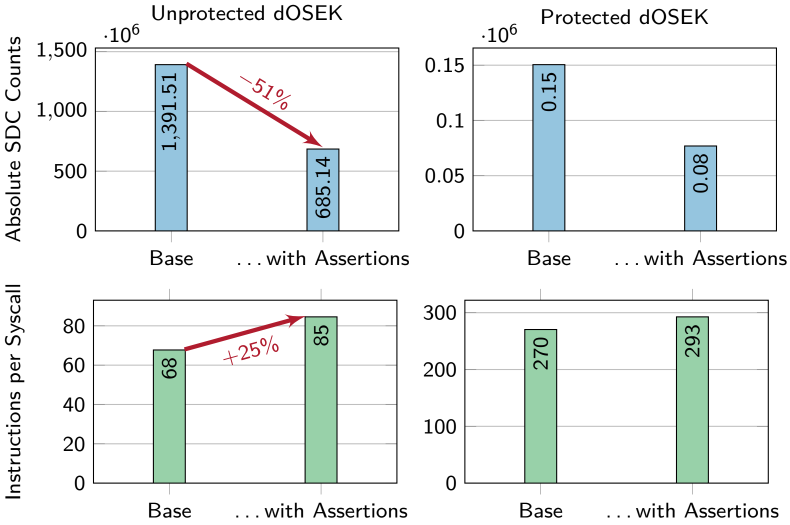


Fault Injection of System-State Assertions

Results with 748 Assertions



Results with 748 Assertions



Results with 748 Assertions