The Global Control-Flow Graph
Optimizing an Event-Driven Real-Time System Across Kernel Boundaries

Christian Dietrich, Martin Hoffmann, Daniel Lohmann
{dietrich,hoffmann,loehmann}@cs.fau.de
Friedrich-Alexander University
Erlangen-Nuremberg

12. November 2015
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

Loop-Invariant Code Motion
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
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

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;
    }
}
```
int lastVal, data[2];
void compute(int a[], int len, int val) {
    int temp = val + 1000;
    ...
}
void Task1() {
    compute(data, 2, lastVal);
}
You are working on a program that performs a specific task. The program consists of two main functions: `compute` and `Task1`. The `compute` function takes an array `a`, its length `len`, and a value `val`. It calculates a temporary value `temp` by adding `val` to 1000, and then assigns this value to two array elements `data[0]` and `data[1]`. The `Task1` function calls `compute` with the current value of `lastVal` and an array of length 2.

```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

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

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

void Task2() {
    lastVal = 23;
    ActivateTask(Task1); // System Call
}

void Task1(){
    data[0] = data[0] + 1023;
}

void Task2() {/* unchanged */}
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
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 execution model for a static real-time systems?

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

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

Global Control-Flow Graph

Question 2:
How to utilize the gathered information?
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

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

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

Application Code

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

```cpp
void TaskB () {
    buf.print();
    TerminateTask();
}
```
Control-Flow Graph

TaskA (priority: 0)

val = readData();
buf.append(val);
if (val != '\n')
    buf.finalize();

ActivateTask(TaskB)

buf.clear();
TerminateTask();

TaskB (priority: 10)

buf.print();

TerminateTask();

---

The Global Control-Flow Graph
Global Control-Flow Graph (GCFG)

TaskA (priority: 0)

val = readData();
buf.append(val);
if (val != '\n')
buf.finalize();
ActivateTask(TaskB)
buf.clear();
TerminateTask();

GCFG

TaskB (priority: 10)

buf.print();
TerminateTask();

buf.print();
TerminateTask();

Computation
System Call
GCFG and System State Enumeration

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

**State Transition Graph**

TerminateTask() → Idle

**Abstract System State**

<table>
<thead>
<tr>
<th>Task State</th>
<th>Priority</th>
<th>Resume Point</th>
<th>Next Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>ready</td>
<td>buf.clear();</td>
<td>TaskB</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>TaskB</td>
</tr>
<tr>
<td>TaskB</td>
<td>running</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FAU The Global Control-Flow Graph
System-State Enumeration and the Transition Graph

State Transition Graph

Abstract System State

<table>
<thead>
<tr>
<th>Task State</th>
<th>Priority</th>
<th>Resume Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>running</td>
<td>0</td>
<td>buf.clear()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TaskA</th>
<th>TaskB</th>
</tr>
</thead>
<tbody>
<tr>
<td>suspended</td>
<td>10</td>
</tr>
</tbody>
</table>

Next Block

TaskA | buf.clear()
System-State Enumeration and the Transition Graph

State Transition Graph

- `TerminateTask()`
- `buf.clear()`
- `TerminateTask()`

Abstract System State

<table>
<thead>
<tr>
<th>Task</th>
<th>Task State</th>
<th>Priority</th>
<th>Resume Point</th>
<th>Next Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>running</td>
<td>0</td>
<td><code>TerminateTask()</code></td>
<td>TaskA, <code>TerminateTask()</code></td>
</tr>
<tr>
<td>TaskB</td>
<td>suspended</td>
<td>10</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

FAU The Global Control-Flow Graph
System-State Enumeration and the Transition Graph

State Transition Graph:
- TerminateTask()
- buf.clear()
- TerminateTask()
- Idle

Abstract System State:

<table>
<thead>
<tr>
<th>Task</th>
<th>Task State</th>
<th>Priority</th>
<th>Resume Point</th>
<th>Next Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>suspended</td>
<td>0</td>
<td>-</td>
<td>Idle</td>
</tr>
<tr>
<td>TaskB</td>
<td>suspended</td>
<td>10</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- FAU The Global Control-Flow Graph
Transforming the State-Transition Graph

FAU The Global Control-Flow Graph
Transforming the State-Transition Graph

Group States by Next Block
Transforming the State-Transition Graph

Group States by Next Block
Outline

Question 1:
How to gather OS execution 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 execution 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
  - ...
  - Kernel Runtime: -30%
  - Resilience Against Bitflips: +50%
  - FSM with 728 States
Control-Flow Graph

TaskA (priority: 0)

val = readData();
buf.append(val);
if (val != '\n')
  buf.finalize();
  ActivateTask(TaskB)
  buf.clear();
  TerminateTask();

TaskB (priority: 10)

buf.print();
TerminateTask();

GCFG → CFG → Computation
System Call
Traditional Library Operating System

- Activates a task
- Sets a task as ready
- Schedules the next task
- Dispatches the next task

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
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: dOSEK (*dependable* OSEK)
  - Fault-tolerant OSEK implementation for IA-32
  - Generative Approach
Evaluation Scenario

Evaluation System: $dOSEK$ (dependable OSEK)
- 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
  - Assertions on the System State
  - Kernel as a Statemachine
  - ...
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: $\sim 50$ percent smaller SDC rate

- Further Applications
  - Improve worst-case execution time analysis of whole applications
  - Replace Kernel by a State Machine (→ OSPERT’15)

Source code available at https://github.com/danceos/dosek
System State Assertions

TaskA

/* Enter Hook */
ActivateTask(TaskB)
/* Leave Hook */
(empty)

TaskB

buf.print();

/* Enter Hook */
TerminateTask()
System State Assertions

TaskA

/* Enter Hook */
assert ready(TaskA)
assert suspended(TaskB)
ActivateTask(TaskB)

/* Leave Hook */

(empty)

TaskB

buf.print();

/* Enter Hook */

TerminateTask();
System State Assertions

**TaskA**

```c
/* Enter Hook */
assert ready(TaskA)
assert suspended(TaskB)
ActivateTask(TaskB)
/* Leave Hook */
```

**TaskB**

```c
buf.print();
```

```c
/* Enter Hook */
assert ready(TaskA)
assert ready(TaskB)
TerminateTask()
```
System State Assertions

TaskA

/* Enter Hook */
assert ready(TaskA)
assert suspended(TaskB)
ActivateTask(TaskB)

/* Leave Hook */
assert ready(TaskA)
assert suspended(TaskB)

(empty)

TaskB

buf.print();

/* Enter Hook */
assert ready(TaskA)
assert ready(TaskB)

TerminateTask()
Fault Injection of System-State Assertions
Results with 748 Assertions

**Unprotected dOSEK**

- Base: $1,391.51 	imes 10^6$
- ... with Assertions: $685.14 	imes 10^6$

**Protected dOSEK**

- Base: $0.15 	imes 10^6$
- ... with Assertions: $0.08 	imes 10^6$

**Instructions per Syscall**

- Base: 68
- ... with Assertions: 85

- Base: 270
- ... with Assertions: 293
Results with 748 Assertions

Unprotected dOSEK
- Absolute SDC Counts
  - Base: 1,391.51
  - ... with Assertions: 685.14
  - Reduction: $-51\%$

Protected dOSEK
- Absolute SDC Counts
  - Base: 1.15
  - ... with Assertions: 0.08

Instructions per Syscall
- Base: 68
- ... with Assertions: 85
- Increase: $+25\%$

- Base: 270
- ... with Assertions: 293

FAU The Global Control-Flow Graph
Results with 748 Assertions

**Unprotected dOSEK**

- Absolute SDC Counts:
  - Base: 1,391.51
  - ... with Assertions: 685.14
  - Decrease: 51%

- Instructions per Syscall:
  - Base: 68
  - ... with Assertions: 85
  - Increase: 25%

**Protected dOSEK**

- Absolute SDC Counts:
  - Base: 0.15
  - ... with Assertions: 0.08
  - Decrease: 49%

- Instructions per Syscall:
  - Base: 270
  - ... with Assertions: 293
  - Increase: 9%