Boosting Job-Level Migration by Static Analysis

Workshop on Operating Systems Platforms for Embedded Real-Time Applications
July 09, 2019

Tobias Klaus, Peter Ulbrich, Phillip Raffeck, Benjamin Frank,
Lisa Wernet, Maxim Ritter von Onciul, Wolfgang Schröder-Preikschat

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Multi-Core Scheduling

Multi-Core Systems

• Static allocation of tasks to cores

Boosting job-level migration by static analysis
Multi-Core Systems

- Static allocation of tasks to cores
  \[ \rightarrow \text{Poor utilization and schedulability} \]
Multi-Core Scheduling

Multi-Core Systems

- Static allocation of tasks to cores
  - Poor utilization and schedulability

Solution: Full Migration

- Dynamic (re)allocation of tasks
- Good utilization and schedulability
Multi-Core Scheduling

Multi-Core Systems

- Static allocation of tasks to cores
  → Poor utilization and schedulability

Solution: Full Migration?

- Dynamic (re)allocation of tasks
- Good utilization and schedulability
Multi-Core Systems

- Static allocation of tasks to cores
  → Poor utilization and schedulability

Solution: Full Migration?

- Dynamic (re)allocation of tasks
- Good utilization and schedulability
  → Impractical in real-time systems

Multi-Core Scheduling
Multi-Core Scheduling

Multi-Core Systems

• Static allocation of tasks to cores
  → Poor utilization and schedulability

Solution: Full Migration?

• Dynamic (re)allocation of tasks
  • Good utilization and schedulability
  → Impractical in real-time systems

Static Allocation Again?

• Split tasks to appropriate size

Boosting job-level migration by static analysis
Find Appropriate Split Points

```c
1  int32_t x = 0;
2  uint16_t y = foo();
3  for (uint8_t i = 0; i < 5; i++) {
4    x += y * bar[i];
5  }
6  int64_t z = x * 4711;
7  for (uint8_t j = 0; j < 5; j++) {
8    z += baz[j];
9  }
10  return z;
```
Splitting the Execution

```c
int32_t x = 0;
uint16_t y = foo();
for (uint8_t i = 0; i < 5; i++) {
    x += y * bar[i];
}
int64_t z = x * 4711;
for (uint8_t j = 0; j < 5; j++) {
    z += baz[j];
}
return z;
```

Find Appropriate Split Points
• Static analysis

Lifespan:

Boosting job-level migration by static analysis
Splitting the Execution

```
1 int32_t x = 0;
2 uint16_t y = foo();
3 for (uint8_t i = 0; i < 5; i++) {
4    x += y * bar[i];
5 }
6 int64_t z = x * 4711;
7 for (uint8_t j = 0; j < 5; j++) {
8    z += baz[j];
9 }
10 return z;
```

Find Appropriate Split Points
- Static analysis
- Consider WCET

Boosting job-level migration by static analysis
Splitting the Execution

1. **int32_t** x = 0;
2. **uint16_t** y = foo();
3. for (uint8_t i = 0; i < 5; i++) {
   x += y * bar[i];
}
4. **int64_t** z = x * 4711;
5. for (uint8_t j = 0; j < 5; j++) {
   z += baz[j];
}
6. return z;

---

Find Appropriate Split Points

- Static analysis
- Consider WCET
- Minimize migration cost

Boosting job-level migration by static analysis
Migration

Challenges

• Split tasks to target WCET
Challenges

• Split tasks to target WCET
• Reduce migration cost
Challenges
- Split tasks to target WCET
- Reduce migration cost

Approach
- Job-Level Migration
- Static Analysis
- Optimization within two dimensions

Boosting job-level migration by static analysis
Overview

Randomly sized scheduling units ➔ Static analysis ➔ Split point graph

Sequential

Uniformly sized scheduling units

Optimization within WCET and migration cost

Branches

Loops

Boosting job-level migration by static analysis
Overview

Randomly sized scheduling units → Static analysis → Split point graph

Sequential

Uniformly sized scheduling units

Branches

Optimization within WCET and migration cost

Loops

Boosting job-level migration by static analysis
Static analysis

Basic Procedure

1. Create control-flow graph
2. WCET analysis
3. Lifespan analysis
Basic Procedure

1. Create control-flow graph
2. WCET analysis
3. Lifespan analysis

Split-point candidates

Boosting job-level migration by static analysis
Split-Point Graphs

Randomly sized scheduling units → Static analysis → Split-point graph

Sequential

Uniformly sized scheduling units

Branches

Loops

Optimization within WCET and migration cost

Boosting job-level migration by static analysis
General Concept: Split-Point Graphs

Control-Flow Graph

Boosting job-level migration by static analysis
General Concept: Split-Point Graphs

Control-Flow Graph

Intermediate Graph

Boosting job-level migration by static analysis
General Concept: Split-Point Graphs

Control-Flow Graph

Intermediate Graph

Split-Point Graph

Boosting job-level migration by static analysis
General Concept: Split-Point Graphs

Control-Flow Graph

Intermediate Graph

Split-Point Graph

Boosting Job-Level Migration

- Static analysis of tasks w.r.t. WCET and resident-set size
- Split-point graphs capture split-point candidates
- Horizontal cuts: finding split points with low migration cost

Boosting job-level migration by static analysis
Overview

Randomly sized scheduling units → Static analysis → Split point graph

Sequential

Uniformly sized scheduling units → Branches → Optimization within WCET and migration cost

Loops

Boosting job-level migration by static analysis
Splitting Loops

Let the body untouched!

Original Loop

```c
1   LOOP_Bound(x:10);
2   for(int i = 0; i < x; ++i)
3   { .... }
```

- Splitting the loop body?
- # of iterations dominates WCET

General Approach

- Compute number of iterations to $/f_it$ target WCET
- Derive upper bound for the number of cuts
- Duplicate body and adjust loop condition
Splitting Loops

Let the body untouched!

Original Loop

1) LOOP_Bound(x:10);
2) for(int i = 0; i < x; ++i)
3) { .... }

• Splitting the loop body?
• # of iterations dominates WCET

→ Split by number of iterations!
Splitting Loops

Let the body untouched!

Original Loop

1  LOOP_Bound(x:10);
2  for(int i = 0; i < x; ++i)
3  { .... }

• Splitting the loop body?
• # of iterations dominates WCET

→ Split by number of iterations!

Loop after Splitting

1  int i = 0, C = 5;
2  for(; i < x && C; ++i)
3  { --C; .... }
4  ....
5  C = 5;
6  for(; i < x && C; ++i)
7  { --C; .... }

General Approach

• Compute number of iterations to fit target WCET
• Derive upper bound for the number of cuts
• Duplicate body and adjust loop condition

Boosting job-level migration by static analysis
Splitting Branches

The problem with conditional load ...

Scheduling Unit (SU)

true

false

\( C_{\text{TRUE}} = 160 \)

\( C_{\text{FALSE}} = 205 \)

exit

true

false

\( C_{\text{TRUE}} = 205 \)

\( C_{\text{FALSE}} = 205 \)

exit

Additional Pessimism Caused by Naive Splitting

- Local optimization may lead to unbalanced cuts in branches
- Condition is unknown at compile time

→ Overapproximation in timing analysis

Boosting job-level migration by static analysis
Splitting Branches

**Global vs. Local Optimization**

- Find suitable points locally
- Global alignment between branches
  → Minimize size differences

Boosting job-level migration by static analysis
Splitting Branches

**Original if-then-else**

- SU\(_A\)
  - BB\(_1\)
  - BB\(_2\)
  - BB\(_3\)
  - BB\(_4\)

**Subdivided if-then-else**

- SU\(_A\)
  - BB\(_1\)
  - BB\(_2\)\(_a\)
  - BB\(_3\)\(_a\)
  - BB\(_5\)

- SU\(_B\)
  - BB\(_6\)
  - BB\(_2\)\(_b\)
  - BB\(_3\)\(_b\)
  - BB\(_4\)

---

**Global vs. Local Optimization**

- Find suitable points locally
- Global alignment between branches
  - Minimize size differences

**General Approach**

- Add jump
- Additional logic

---

Boosting job-level migration by static analysis
Overheads per Cut

Sequential Code

\[ i_{seq}^+ = 1 \]
Overheads per Cut

How much is the fun?

Sequential Code

\[ i_{seq}^+ = 1 \]

Branches

\[ i_{if}^+ = n_{branch} \times 2 \]

Marking the active branch

Terminating the first scheduling unit

Proceeding with the correct branch

Low overall overhead

- Only few additional instructions for all different program constructs
  \[ \Rightarrow \] Minor effects on overall execution time

Boosting job-level migration by static analysis
Overheads per Cut

Sequential Code

\[ i_{\text{seq}}^+ = 1 \]

Branches

\[ i_{\text{if}}^+ = n_{\text{branch}} \times 2 \]

Marking the active branch

Terminating the first scheduling unit

Proceeding with the correct branch

Loops

\[ i_{\text{loop}}^+ = (5 + 1) \]

Counter for planned iterations

Exiting the scheduling unit and resetting the iteration counter

Executing the following part of the loop

\[ i^+ \] # additional instructions

\[ n_{\text{branch}} \] # branches, affected by a horizontal cut

Boosting job-level migration by static analysis
Overheads per Cut

Sequential Code

\[ i_{seq}^+ = 1 \]

Branches

\[ i_{if}^+ = n_{branch} \times 2 \]
\[ + 1 \]
\[ + 3 \]

Marking the active branch

Terminating the first scheduling unit

Low overall overhead

- Only few additional instructions for all different program constructs

\[ \Rightarrow \] Minor effects on overall execution time

Loops

\[ i_{loop}^+ = (5 + 1) \]
\[ + 2 \]
\[ + 3 \]

Executing the following part of the loop

\[ i^+ \quad \text{# additional instructions} \]

\[ n_{branch} \quad \text{# branches, affected by a horizontal cut} \]
Effects on the schedulability of systems with high utilization

Experimental Setup
- System with four processor cores
- 12000 synthetic benchmark systems

Goal
- Feasible allocation and schedule for each task set
Effects on the schedulability of systems with high utilization

Experimental Setup

- System with four processor cores
- 12000 synthetic benchmark systems

Goal

- Feasible allocation and schedule for each task set

⇒ 70 percent more schedulable task sets for the highest utilization
Finding split points with low migration cost

Experimental Setup

• Real-world benchmarks taken from the TACLeBench suite
• Creation of OSEK systems: one benchmark task and two load tasks
  • Generate systems which are unschedulable on two cores without migration
  • Only cut benchmark tasks
• Recording of the resident-set size (in LLVM-IR types)
  • Worst-case migration cost observed in all possible split-point candidates
  • Migration cost of the split point chosen by our approach
## Migration Costs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>binarysearch</td>
<td>225</td>
<td>224</td>
<td>1</td>
</tr>
<tr>
<td>bitonic</td>
<td>65</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>complex_update</td>
<td>480</td>
<td>288</td>
<td>192</td>
</tr>
<tr>
<td>countnegative</td>
<td>2176</td>
<td>1568</td>
<td>608</td>
</tr>
<tr>
<td>filterbank</td>
<td>60736</td>
<td>60704</td>
<td>32</td>
</tr>
<tr>
<td>iir</td>
<td>432</td>
<td>400</td>
<td>32</td>
</tr>
<tr>
<td>insertsort</td>
<td>544</td>
<td>128</td>
<td>416</td>
</tr>
<tr>
<td>minver</td>
<td>17568</td>
<td>16800</td>
<td>768</td>
</tr>
<tr>
<td>petrinet</td>
<td>5057</td>
<td>5056</td>
<td>1</td>
</tr>
</tbody>
</table>
## Migration Costs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>binarysearch</td>
<td>225</td>
<td>224</td>
<td>1</td>
</tr>
<tr>
<td>bitonic</td>
<td>65</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>complex_update</td>
<td>480</td>
<td>288</td>
<td>192</td>
</tr>
<tr>
<td>countnegative</td>
<td>2176</td>
<td>1568</td>
<td>608</td>
</tr>
<tr>
<td>filterbank</td>
<td>60736</td>
<td>60704</td>
<td>32</td>
</tr>
<tr>
<td>iir</td>
<td>432</td>
<td>400</td>
<td>32</td>
</tr>
<tr>
<td>insertsort</td>
<td>544</td>
<td>128</td>
<td>416</td>
</tr>
<tr>
<td>minver</td>
<td>17568</td>
<td>16800</td>
<td>768</td>
</tr>
<tr>
<td>petrinet</td>
<td>5057</td>
<td>5056</td>
<td>1</td>
</tr>
</tbody>
</table>

⇒ **Lower worst-case migration overhead**

⇒ **Tighter results from timing analysis**

Boosting job-level migration by static analysis
Conclusion

• Compile time
  • Beneficial size of scheduling units

⇒ Systems with high utilization become schedulable
Conclusion and Outlook

Conclusion

• Compile time
  • Beneficial size of scheduling units
  ⇒ Systems with high utilization become schedulable

• Runtime
  • Migration at beneficial points
  • Only if necessary
  ⇒ Reducing overapproximation in the WCET analysis

Current Work and Outlook

• More accurate WCET estimation
• Adapt an OS to support migration threshold
• Consider the OS and system calls within the analysis

Boosting job-level migration by static analysis
Conclusion and Outlook

Conclusion

• Compile time
  • Beneficial size of scheduling units
  ⇒ Systems with high utilization become schedulable

• Runtime
  • Migration at beneficial points
  • Only if necessary
  ⇒ Reducing overapproximation in the WCET analysis

Current Work and Outlook

• More accurate WCET estimation
• Adapt an OS to support migration threshold
• Consider the OS and system calls within the analysis