Throughput-optimizing Compilation of Dataflow Applications for Multi-Cores using Quasi-Static Scheduling

Tobias Schwarzer¹, Joachim Falk¹, Michael Glaß¹, Jürgen Teich¹, Christian Zebelein², Christian Haubelt²

¹Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
²University of Rostock, Germany
Advantages:
- Exposes all degrees of available parallelism
- High flexibility of mapping actors to resources

Disadvantages:
- Scheduling overhead at runtime
- May sacrifice the code-generation efficiency of the compiler
Coarse-granular Modeling

- Advantages:
  - High code-generation efficiency
  - Low scheduling overhead at runtime

- Disadvantages:
  - May sacrifice parallelism significantly
  - Reduces mapping flexibility
Proposed Solution

● Input:
  ● Fine-grained (data flow) modelling of applications
  ● Model of the target platform

● Given: a clustering technique [FKHTB08] to automatically combine fine-grained actors to composite actors using Quasi-Static Scheduling (QSS)

● Key Idea: Perform a Design Space Exploration (DSE) to find the best combinations of clusters and their mapping to available resources in terms of application throughput

● Contributions:
  ● A genetic representation for the exploration of clusters
  ● A repair strategy to efficiently restrict the search to feasible clusters only
  ● Synthesis-in-the-Loop: Automatic synthesis of each implementation candidate to the target platform to determine quality numbers, capturing the effects of (a) scheduling overhead, (b) memory accesses, and (c) compiler optimizations

Proposed System Compilation Flow

Automatic extraction of the dataflow graph

Explore design space
Step 1: Determine clustering
Step 2: Perform binding
Architecture graph

Non-dominated design points

Input

Dataflow application

Read in executable specification

Synthesis-in-the-loop

DSE

Evaluate implementation on target hardware

Run executable
Obtain throughput

Output

Non-dominated design points

Schwarzer et al. | Throughput-optimizing Compilation of Dataflow Applications for Multi-Cores using Quasi-Static Scheduling | SCOPES 2015
Fundamentals
Classic System Synthesis

Problem graph \( g_p = (A, C) \):

Architecture graph \( g_r = (R, L) \):

- Dynamic actors: red
- Static actors: blue

Problem graph:

- \( a_1 \) connected to \( a_2 \), \( a_3 \), \( a_4 \), \( a_5 \), \( a_6 \), \( a_7 \), \( a_8 \), \( a_9 \)

Architecture graph:

- \( r_{CPU1} \) connected to \( r_{CPU2} \), \( r_{CPU3} \), \( r_{CPU4} \)
Classic System Synthesis

Problem graph $g_p = (A, C)$:

Architecture graph $g_r = (R, L)$:

Binding $\beta$

- Dynamic actors
- Static actors
Clustering-aware System Synthesis

- Clustering:
  Replace a connected subgraph containing only static actors of a DFG by a single composite actor.
Clustering-aware System Synthesis

- **Clustering:**
  Replace a connected subgraph containing only static actors of a DFG by a single composite actor

- **Each composite actor contains a Quasi-Static Schedule (QSS)**
  executing sequences of statically scheduled actor firings

- **QSS is represented by a Finite State Machine (FSM)**

- **Details in [FKHTB08]**

Clustering-aware System Synthesis

- **Clustering:**
  Replace a connected subgraph containing only static actors of a DFG by a single composite actor

- **Each composite actor contains a Quasi-Static Schedule (QSS)**
  executing sequences of statically scheduled actor firings

- **QSS is represented by a Finite State Machine (FSM)**

- **Details in [FKHTB08]**

Clustering-aware System Synthesis

- Clustering:
  Replace a connected subgraph containing only static actors of a DFG by a single composite actor
- Each composite actor contains a Quasi-Static Schedule (QSS) executing sequences of statically scheduled actor firings
- QSS is represented by a Finite State Machine (FSM)
- Details in [FKHTB08]

Clustering-aware System Synthesis

Cluster 1
- a_1
- a_2
- a_3
- a_4
- a_5
- a_6
- a_7
- a_8
- a_9

Cluster 2
- a_1

Binding \( \beta \)

Dynamic actors: Red
Static actors: Blue

CPU 1
- r_{CPU1}

CPU 2
- r_{CPU2}

CPU 3
- r_{CPU3}

CPU 4
- r_{CPU4}
Design Space Exploration
Design Space Exploration - Opt4J

Custom optimization problems are typically defined by providing three main classes:

- **Creator**: Generate random genotype objects
- **Decoder**: Convert the genotype into a phenotype
- **Evaluator**: Determine the quality of one phenotype

![Diagram of the optimization process](http://opt4j.sourceforge.net/documentation/3.1/img/problem.png)
Design Space Exploration - Creator

Generate random genotype objects:

- Associate each channel of the problem graph with a Boolean value using the clusterization function $\gamma$:

  $\gamma = C_S \rightarrow \{\text{true, false}\}$

  - “true” $\rightarrow$ channel should be contained in a cluster
  - “false” $\rightarrow$ channel should not be contained in a cluster

- Exclude channels connected to a dynamic actor from genotype
Design Space Exploration - Decoder

- How to convert the genotype (true/false edges) to a phenotype (clusters)?

Conversion consists of two steps:

1. Determine *clustering candidates*
2. *Repair* clustering candidates to obtain a feasible cluster
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 1: Determine Clustering Candidate

Collect reachable actors starting at channel associated with “true”
Step 2: Repair Strategy

Schwarzer et al. | Throughput-optimizing Compilation of Dataflow Applications for Multi-Cores using Quasi-Static Scheduling | SCOPES 2015
Step 2: Repair Strategy
Step 2: Repair Strategy

Schwarzer et al. | Throughput-optimizing Compilation of Dataflow Applications for Multi-Cores using Quasi-Static Scheduling | SCOPES 2015
Step 2: Repair Strategy

Clustering condition:
For each pair of cluster input and output actor, there exists a directed path from the input actor to the output actor.
Step 2: Repair Strategy

Clustering condition:
For each pair of cluster input and output actor, there exists a directed path from the input actor to the output actor.

→ Repair strategy: Transform an infeasible clustering candidate into (multiple) feasible clusters

cluster candidate input actors \{a_2, a_6\}
clustering candidate output actors \{a_3, a_4, a_7\}
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:

cluster candidate input actors \{a2,a6\}
clustering candidate output actors \{a3,a4,a7\}
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:
- Determine set of predecessor input actors and set of successor output actors for each actor

cluster candidate input actors $\{a_2, a_6\}$
cluster candidate output actors $\{a_3, a_4, a_7\}$
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:

- Determine set of predecessor input actors and set of successor output actors for each actor.

\[
I = \{a_2\} \\
O = \{a_3, a_4\}
\]

\[
I = \{a_2, a_6\} \\
O = \{a_3, a_4\}
\]

\[
I = \{a_2, a_6\} \\
O = \{a_3, a_4\}
\]

\[
I = \{a_6\} \\
O = \{a_3, a_4, a_7\}
\]

\[
I = \{a_6\} \\
O = \{a_3, a_4, a_7\}
\]

\[
I = \{a_6\} \\
O = \{a_3, a_4, a_7\}
\]

\[
I = \{a_6\} \\
O = \{a_3, a_4, a_7\}
\]

\[
I = \{a_6\} \\
O = \{a_3, a_4, a_7\}
\]

\[
I = \{a_6\} \\
O = \{a_3, a_4, a_7\}
\]
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:

- Determine set of predecessor input actors and set of successor output actors for each actor

- Create equivalence classes (ECs)

Example:

- EC1: I = \{a_2\}, O = \{a_3, a_4\}
- EC2: I = \{a_6\}, O = \{a_3, a_4, a_7\}
- EC3: I = \{a_6\}, O = \{a_3, a_4\}
- EC4: I = \{a_6\}, O = \{a_3, a_4, a_7\}
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:

- Determine set of predecessor input actors and set of successor output actors for each actor
- Create equivalence classes (ECs)
- Determine clashes between each EC
Step 2: Repair Strategy

Repair clustering candidate to obtain feasible cluster:

- Determine set of predecessor input actors and set of successor output actors for each actor.
- Create equivalence classes (ECs).
- Determine clashes between each EC.

**Clash:** Detect the situation when an output actor is not connected to all input actors, since this may result in the *infinite token accumulation problem*. (Formal details given in the paper)
Step 2: Repair Strategy

Clash exists between EC2 and EC4!
Repair strategy splits the cluster candidate between a9 and a6 or between a9 and a4.
Design Space Exploration - Evaluator

Determine the quality of one phenotype:

- Each implementation candidate is evaluated via synthesis-in-the-loop on the target platform with respect to the design objectives:
  - Throughput [iterations/s] (MAX)
  - Number of used processor cores (MIN)
Experimental Results
Experimental results

● Test suite:
  ● Six synthetic DFGs using the SDF³ tool
  ● Brake-by-wire control application (BBW)

● Target platforms:
  ● Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
  ● Xilinx Zynq-7000 All Programmable SoC with a dual-core ARM Cortex-A9 processor @ 667MHz

● Approaches:
  ● *classic* (DSE without clustering)
  ● *post QSS* (DSE with clustering as a post-processing step)
    ● Direct combination of [FKHTB08] and DSE
  ● Proposed *holistic* DSE flow

Results for synthetic DFG $g_P^C$

- ARM Cortex-A9
  - 1 core: Classic 1, Post QSS 1.14, Holistic 2.8, Speedup factor 2.88
  - 2 cores: Classic 1, Post QSS 1.47, Holistic 7.2
- Intel i7-2600
  - 1 core: Classic 1, Post QSS 1.52, Holistic 3.55
  - 2 cores: Classic 1.33, Post QSS 1.84, Holistic 8.92
  - 3 cores: Classic 1.83, Post QSS 2.12, Holistic 8.96
  - 4 cores: Classic 1.98, Post QSS, Holistic 9.91
## Results

<table>
<thead>
<tr>
<th>Approach</th>
<th>Intel i7-2600</th>
<th>Zynq-7000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-Core</td>
<td>2-Core</td>
</tr>
<tr>
<td>$g_P^A$ classic</td>
<td>1 x</td>
<td>—</td>
</tr>
<tr>
<td>$g_P^A$ post QSS</td>
<td>2.33 x</td>
<td>—</td>
</tr>
<tr>
<td>$g_P^A$ holistic</td>
<td><strong>2.60 x</strong></td>
<td>—</td>
</tr>
<tr>
<td>$g_P^B$ classic</td>
<td>1 x</td>
<td>1.06 x</td>
</tr>
<tr>
<td>$g_P^B$ post QSS</td>
<td><strong>1.42 x</strong></td>
<td>1.15 x</td>
</tr>
<tr>
<td>$g_P^B$ holistic</td>
<td>1.40 x</td>
<td><strong>2.51 x</strong></td>
</tr>
<tr>
<td>$g_P^C$ classic</td>
<td>1 x</td>
<td>1.52 x</td>
</tr>
<tr>
<td>$g_P^C$ post QSS</td>
<td>3.55 x</td>
<td>1.33 x</td>
</tr>
<tr>
<td>$g_P^C$ holistic</td>
<td><strong>3.73 x</strong></td>
<td><strong>8.92 x</strong></td>
</tr>
<tr>
<td>$g_P^D$ classic</td>
<td>1 x</td>
<td>1.94 x</td>
</tr>
<tr>
<td>$g_P^D$ post QSS</td>
<td>1.35 x</td>
<td>1.83 x</td>
</tr>
<tr>
<td>$g_P^D$ holistic</td>
<td><strong>1.52 x</strong></td>
<td><strong>3.25 x</strong></td>
</tr>
<tr>
<td>$g_P^E$ classic</td>
<td>1 x</td>
<td>1.66 x</td>
</tr>
<tr>
<td>$g_P^E$ post QSS</td>
<td>1.43 x</td>
<td>1.19 x</td>
</tr>
<tr>
<td>$g_P^E$ holistic</td>
<td><strong>1.77 x</strong></td>
<td><strong>2.63 x</strong></td>
</tr>
<tr>
<td>$g_P^F$ classic</td>
<td>1 x</td>
<td>1.89 x</td>
</tr>
<tr>
<td>$g_P^F$ post QSS</td>
<td><strong>1.57 x</strong></td>
<td>2.05 x</td>
</tr>
<tr>
<td>$g_P^F$ holistic</td>
<td>1.48 x</td>
<td><strong>3.23 x</strong></td>
</tr>
</tbody>
</table>

$BBW$ classic | 1 x | 1.80 x | 2.46 x | 3.16 x | 1 x | 1.78 x |

$BBW$ post QSS | **2.91 x** | 2.24 x | 2.81 x | 3.35 x | **2.06 x** | 2.39 x |

$BBW$ holistic | 2.38 x | **4.10 x** | **5.34 x** | — | 1.79 x | **3.45 x** |
Results

<table>
<thead>
<tr>
<th>Approach</th>
<th>Intel i7-2600</th>
<th>Zynq-7000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-Core</td>
<td>2-Core</td>
</tr>
<tr>
<td>$g^A_D$ classic</td>
<td>1 x</td>
<td>—</td>
</tr>
<tr>
<td>$g^A_D$ post QSS</td>
<td>2.33 x</td>
<td>—</td>
</tr>
</tbody>
</table>

- **Holistic approach**
  - is always superior to classic DSE
  - outperforms post QSS in the multi-processor case

- **Speedups up to**
  - 9.91x for four processor cores of the intel platform
  - 7.2x for two processor cores on the Zynq platform
Conclusion

- Holistic compilation flow combines system synthesis and the use of QSSs in a DSE to maximize application throughput for a given target platform

- DSE explores the clustering of static actors to fine-tune the tradeoff between
  - exploitable degree of parallelism,
  - dynamic scheduling overhead reduction, and
  - compiler code-generation efficiency

- Novel repair strategy ensures the creation of feasible clusters

- Each implementation candidate is compiled to and measured on real target hardware (synthesis-in-the-loop)
Thank you for your attention!

Questions?

Acknowledgements:
This work is supported in part by the German Science Foundation (DFG), HA 4463/3-2 and TE 163/18-2.