효율적인 모델 기반 설계를 위한 최적화 코드 생성
김학범, MathWorks Korea
The new XC 90 is built on the SPA platform utilizing Model-Based Design and AUTOSAR in Volvo.

Alenia Aermacchi develops autopilot software for DO-178B level A certification.

ITK engineering develops IEC 62304 compliant controller for a dental drill motor with MBD.

Stem accelerates development of power electronics control system with MBD.

IDNEO develops embedded computer vision and machine learning algorithms for interpreting blood type results.
Code Generation Connects Model-Based Design Workflows

- Accelerate development process
- Reduce translation error
- Enable rapid iterative workflows
Design an Embedded Controller

How to optimize embedded software?

Plant or Environment

Controller or Application

Embedded System
Approach to Code Efficiency with Model-Based Design

Model Analysis
- Model level & Algorithm level analysis
- Application-aware optimizations (modeling pattern)

Code Generation
- Implementation level analysis
- Target-aware optimizations (resources)
Embedded Coder® Quick Start helps you generate production code for 'rtwdemo_roll'.

The Quick Start tool:
- Asks a few questions about your code generation goals and your target hardware.
- Validates your model against your selections.
- Shows you the recommended configuration changes.
- Applies the configuration changes and generates code.

No changes are made to your configuration until you choose to generate code. After successful code generation, the Quick Start tool presents possible next steps.
RAM, ROM and Execution Performance

- Data copy reduction
- Buffer reuse
- Minimize global accesses, Line Of Code

- Execution Speed
- ROM Efficiency
- RAM Efficiency
RAM, ROM and Execution Performance

- ROM Efficiency
- RAM Efficiency
- Execution Speed
Challenge: Maintaining Large and Complex Systems

- Size and complexity of systems are increasing
  - "Typical ECU contains 2000 function components that are each developed by a different person" Automotive customer

- "Enforcement of low complexity" required for model standards
  - ISO 26262-6 "Product Development at the Software Level", Table 1
Challenge: Maintaining Large and Complex Systems

- Studies estimate 13-20% of code in large systems are cloned *
- Old fashion modeling patterns appear:

* Source: Roy and Cordy A Survey on Software Clone Detection Research, Sept 2007
Clone Detection
• Find duplicate model content in your design
• Locate opportunities to optimize with a library

Refactoring
• Replace exact clones with library blocks
• Improve reuse and maintainability
DEMO: Detect Clone in Model
Review Generated Code

Before

After

ROM Efficiency

MATLAB EXPO

Refactoring

SS3

SS5

SS3,5
Library-Based Subsystem Code Generation

- System complexity → Unit testing
  - Models ✓
  - Subsystems

- Library-based Subsystem Code Generation
  - Lock down function interfaces
  - Generate small reusable sub-functions
  - Verify usage within a model using SIL/PIL
  - **SIL/PIL unit test in library with code coverage**

Normal SIL PIL

ROM Efficiency

\[ R_{2019a} \]

\[ R_{2020a} \]

MATLAB EXPO

MathWorks
RAM, ROM and Execution Performance

- Data copy reduction
- Buffer reuse
- Minimize global accesses, line of code

Execution Speed
ROM Efficiency
RAM Efficiency
RAM, ROM and Execution Performance

Execution Speed

ROM Efficiency

RAM Efficiency
Reuse Local Block Output

Pulse Generator

Model_B.Add

Model_B.Subsystem1

Model_B.Subsystem2

Model_B.y

1 buffer

2 buffers

3 buffers

RAM Efficiency
Reuse Local Block Output

- Reuse buffers with different sizes and/or shapes (dimensions)
  - Different buffers collapse to one, the biggest size is kept
Reuse Local Block Output

Model.Switch1

Model.Switch2

Reuse signal storage
Reuse Local Block Output

- Review Code Generation Report

Use only 3 variables for 9 blocks - Reduce 6 variables
Reuse Local Block Output

- Review Static Code Metrics Report

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>33</td>
<td>30</td>
</tr>
<tr>
<td>Total Lines</td>
<td>89</td>
<td>78</td>
</tr>
</tbody>
</table>
Reuse Buffer Using Signal Labels

- Using Signal Labels to Guide Buffer Reuse
  - Case#1: Same variable for the Atomic Subsystem and Saturation block outputs
  - Case#2: Same variable for the Atomic Subsystem and Gain block outputs
Reuse Buffer Using Signal Labels

- Using Signal Labels to Guide Buffer Reuse
  - Same variable for the Atomic Subsystem and Saturation block outputs

![Diagram of buffer reuse with signal labels](image)
Reduce Code Complexity by Refactoring Subsystem
Reduce Code Complexity by Refactoring Subsystem

- Review Code Generation Report

Create subsystem function
Reduce Code Complexity by Refactoring Subsystem

- Review Static Code Metrics Report

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>33</td>
<td>11</td>
</tr>
<tr>
<td>Total Lines</td>
<td>78</td>
<td>58</td>
</tr>
</tbody>
</table>
Optimize Generate Code in Reusable Subsystem

- Passing Reusable Subsystem Outputs as *Structure Reference*
Optimize Generate Code in Reusable Subsystem

- Passing Reusable Subsystem Outputs as *Individual Argument*

- Save buffer storage
- Reduce data copy
- RAM efficiency
- Execution speed
Optimize Generate Code in Reusable Subsystem

RAM Efficiency
Optimize Generate Code in Reusable Subsystem

- Review Code Generation Report

- 3 times data copy from subsystem structure

- 1 times data passing using arguments

[Structure reference]

[Individual argument]
Optimize Generate Code in Reusable Subsystem

- Review Static Code Metrics Report

![Global Variables Table]

Global variables: -24 bytes
Read/Write: -4 times
Easy to Configure Options for Optimizing Code

Automatically checked options

MATLAB EXPO
RAM, ROM and Execution Performance

- Minimize global Accesses, line of code
- Data copy reduction
- Buffer reuse
- Execution Speed
- ROM Efficiency
- RAM Efficiency
RAM, ROM and Execution Performance

Execution Speed

ROM Efficiency

RAM Efficiency
Row-Major vs. Column-Major

- **Row-Major layout**
  - Elements of the rows are contiguous
  - C and C++ use row-major layout

  \[
  X = \begin{bmatrix}
  x_1 & x_2 & x_3 \\
  x_4 & x_5 & x_6 \\
  x_7 & x_8 & x_9 
  \end{bmatrix}
  \]

  The elements of the array are stored: \( x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9 \)

- **Column-Major layout**
  - Elements of the columns are contiguous
  - MATLAB® and Fortran use column-major layout

  \[
  X = \begin{bmatrix}
  x_1 & x_2 & x_3 \\
  x_4 & x_5 & x_6 \\
  x_7 & x_8 & x_9 
  \end{bmatrix}
  \]

  The elements of the array are stored: \( x_1, x_4, x_7, x_2, x_5, x_8, x_3, x_6, x_9 \)
MATLAB:

M =

11  12
21  22
31  32

Column-major code generation:
M[] = {11, 21, 31, 12, 22, 32};
M[2] = 31

Row-major code generation:
M[] = {11, 12, 21, 22, 31, 32};
M[2] = 21
Row-Major vs. Column-Major

- CPUs Process Sequential Data More Efficiently than nonsequential data

\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23}
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>Address</th>
<th>Access</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A[0][0]</td>
<td>(a_{11})</td>
</tr>
<tr>
<td>1</td>
<td>A[0][1]</td>
<td>(a_{12})</td>
</tr>
<tr>
<td>2</td>
<td>A[0][2]</td>
<td>(a_{13})</td>
</tr>
<tr>
<td>3</td>
<td>A[1][0]</td>
<td>(a_{21})</td>
</tr>
<tr>
<td>4</td>
<td>A[1][1]</td>
<td>(a_{22})</td>
</tr>
<tr>
<td>5</td>
<td>A[1][2]</td>
<td>(a_{23})</td>
</tr>
</tbody>
</table>

a_{11}, a_{12}, a_{13}, ……

Sequently data read

Programmed by C

[Raw-major order]
Row-Major vs. Column-Major

- CPUs Process Sequential Data More Efficiently than nonsequential data

\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23}
\end{bmatrix}
\]

- Need data indexing!!

<table>
<thead>
<tr>
<th>Address</th>
<th>Access</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A[0][0]</td>
<td>a_{11}</td>
</tr>
<tr>
<td>1</td>
<td>A[1][0]</td>
<td>a_{21}</td>
</tr>
<tr>
<td>2</td>
<td>A[0][1]</td>
<td>a_{12}</td>
</tr>
<tr>
<td>3</td>
<td>A[1][1]</td>
<td>a_{22}</td>
</tr>
<tr>
<td>4</td>
<td>A[0][2]</td>
<td>a_{13}</td>
</tr>
<tr>
<td>5</td>
<td>A[1][2]</td>
<td>a_{23}</td>
</tr>
</tbody>
</table>

[Column major order]

- Memory access times increase!!
Row-Major vs. Column-Major

\[ P = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \]

Column-major layout

```
P rtP = {
  /* Variable: P */
  /* Referenced by: '<Root>/Constant' */
  { 1.0, 3.0, 5.0, 2.0, 4.0, 6.0 }
};
```

Row-major layout

```
P rtP = {
  /* Variable: P */
  /* Referenced by: '<Root>/Constant' */
  { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 }
};
```
Row-Major and Multi-Dimension Indexing

\[ P = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \]

Row-major layout

Multi-Dimensional layout

\[
\text{P[3][2]} = \{ \{ 1.0, 2.0 \}, \{ 3.0, 4.0 \}, \{ 5.0, 6.0 \} \}
\]
Generating Row-Major Code

File: mycode.c

```c
1 /*
2 * File: mycode.c
3 *
4 * Code generated for Simulink model 'rtwdemo_row_lut2d'.
5 *
6 * Model version : 1.22
7 * Simulink Coder version : 9.1 (R2019a) 23-Nov-2018
8 * C/C++ source code generated on : Tue Apr 16 15:15:58 2019
9 *
10 * Target selection: ert.tlc
11 * Embedded hardware selection: Specified
12 * Code generation objectives: Unspecified
13 * Validation result: Not run
14 */
15
16 #include "rtwdemo_row_lut2d.h"
17
18 /* Exported data definition */
19
20 /* Definition for custom storage class: ExportToFile */
21 real_T P[3][2] = {{ 1.0, 2.0 }, { 3.0, 4.0 }, { 5.0, 6.0 }};
22
23 /*
24 * File trailer for generated code.
25 *
26 * [EOF]
27 */
28```

Execution Speed
Code Execution Profiling with SIL and PIL

- Produce execution time metric for tasks and functions in the generated code
  - Measure execution time, self time, CPU utilization and number of calls
  - Identify tasks that require the most execution time
  - In these tasks, investigate code sections that require the most execution time
Code Execution Profiling with SIL and PIL

- How to Generate Execution-Time Metrics in SIL/PIL Manager
Improving Code and Model Performance

Execution Speed

MATLAB EXPO

View full code execution profiling report
Improving Code and Model Performance

>> out.get('executionProfile').Sections(4)
Key Takeaway

- Improving Modeling Patterns for Efficiency
  - Clone detection, memory efficiency
- RAM and Data Copy Reduction
  - Buffer reuse, reduction data copy
- Execution Speed
  - Row-major and column-major
  - Code execution profiling
Thank You !!