Deep Optimization: Eliminating Common Implementation Inefficiencies

Chris Hibner
October 2015

Intro to Pi

Pi Innovo specializes in vehicle electronics innovation, ECU hardware and software, electronics design engineering, automotive systems engineering, emissions, chassis, suspension, prototyping tools, IC, fuel cell, electric, hybrid powertrain, in-vehicle Infotainment (IVI).

OpenECU is a wide range of adaptable, field-ready products and intellectual property designed to accelerate electronics system development.

Our Capability

World Class Engineering Consultancy

- Broad domain expertise
- System design
- Control design
- Electronics design
- Custom products
- SEI CMMI Level 3

Field Proven Standard ECU Production

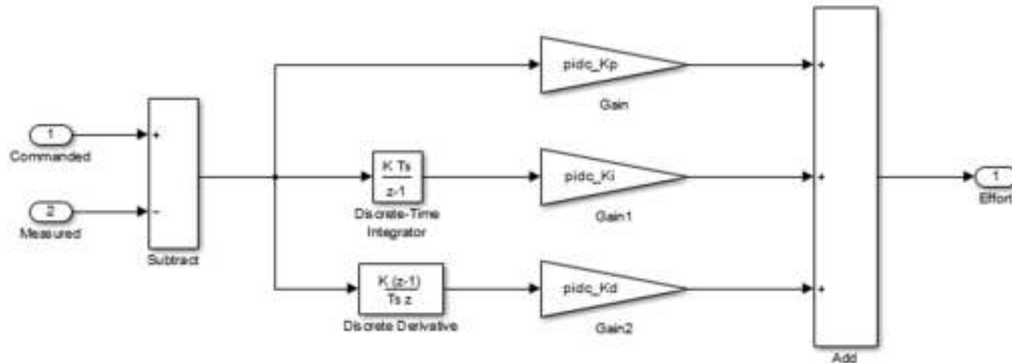
- Large hardware family
- Application software
- Rapid prototyping tools
- Industry tools integration
- Semicustom options

Manufacturing Partners

- Tier 1 & 2
- Automotive
- Contract Manufacture
- TS16949, ISO 14001

Background

Simulink is a popular environment for developing and testing controls software



- Simulink is a popular environment for developing and testing controls software

- Non-coders feel comfortable using this tool

    Graphical interface

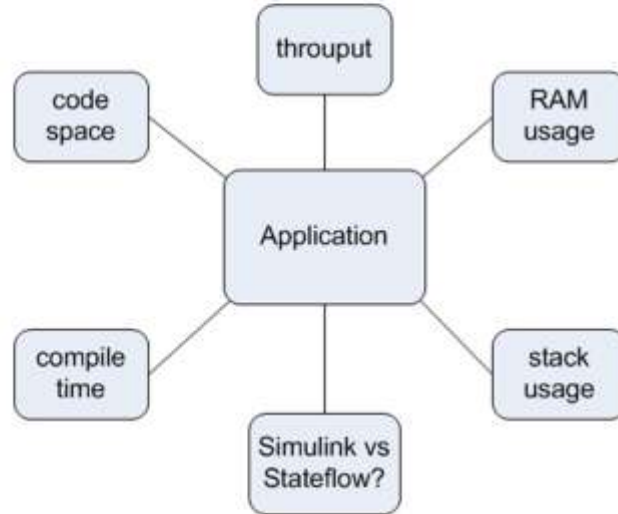
    Built in libraries

    Auto code generation

    Auto managed hardware resources

    Many simulation and debug tools available

Challenges



- Is the auto generated code efficient and when run on target, will it consume only the minimum amount of resource needed?
- Are the functions being executed in the correct sequence?
- Are the algorithms using the latest data or is there some unnecessary latency?
- Are the tasks being executed in the allocated loop time or taking longer than intended?
- Would using Stateflow instead of Simulink or vice versa result in a more efficient code?

-These questions need to be answered when the model wont build, the ECU resets, tasks overrun, or other anomalous behavior is seen

In reality...

The previous challenges do not matter if the micro is over-sized.

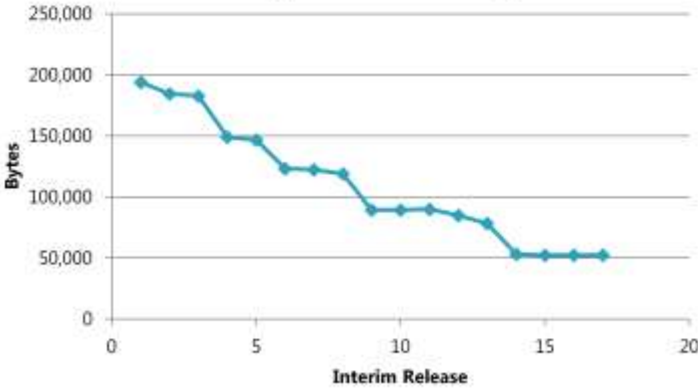
Oversized micro is fine for prototyping phase.

Harsh reality…

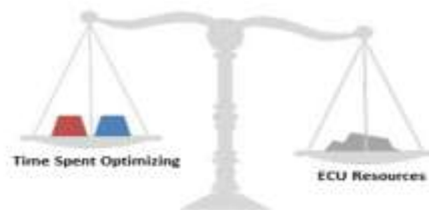
Using an oversized micro means too much cost.

Too much cost => will not be acceptable in production

Complex systems will rapidly consume micro resources as the model grows



Migration from prototype phase to production can bring inefficient code and architecture



-Consider application architecture before prototyping.

But what about style guides?

- MISRA style guide for Simulink/Stateflow applications
  - Focus is primarily on readability and maintaining style consistency
  - Other style guides available, but most don't provide sufficient guidance on creating efficient code
- Mathworks does have guidance: "Modeling Guidelines for Code Generation"

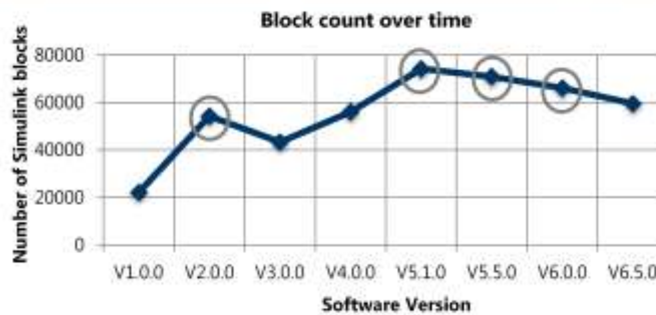
  (cn.mathworks.com/help/pdf_doc/simulink/cg_guidelines.pdf)
- Most efficient modeling methods are tribal knowledge and usually remains within companies
  - Comes from experience, and learned the hard way

Case study – brake based stability control

-ABS, TCS, ESC and lots of diagnostic features
- Features were slowly rolled out into the parent model
- Diagnostic features were about 50% of the total code
- ECU used an older processor – resources were not abundant
- Large team size: 10 engineers working in parallel
- Circles indicate when optimization exercise was performed

Large production project == efficient code needed

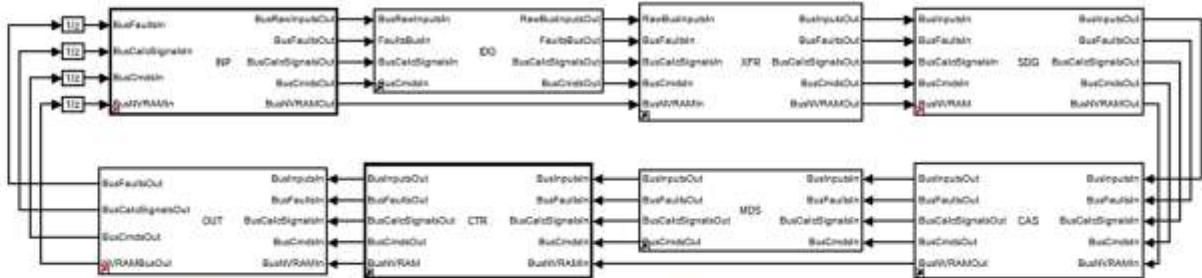
- In spite of all the planning and preparations, multiple rounds of optimization had to be performed
  - Too large for compiler to handle
  - Required more memory than what was available
  - Very computationally intensive – tasks often overran the designated loop time



-Feature creep caused model content to grow => in 5.5.0, there was more content but optimizations resulted in fewer blocks.
- v1.0.0: Base brakes
- v2.0.0: ABS added
- v3.0.0: TCS added and software optimized
- v4.0.0: ESC added
- remaining versions: additional small feature content, improvements, and optimizations.

Lessons learned

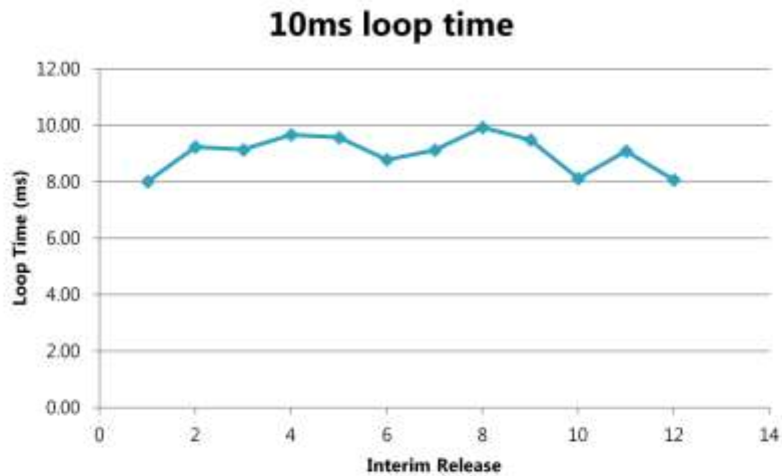
Choosing a model architecture



- Libraries allow multiple engineers to develop in parallel
- Bus architecture makes model clean
- Bus signals grouped by function. Example:
  - Faults bus
  - Commands bus

- Libraries subsystems allow engineers to work on individual features in parallel. When ready, it is merged into the parent model's directory
- Busses reduce clutter, makes debugging models easy – grouped by function
- Grouping them into functions helps find a signal faster. For example, a valve command signal would be on the "Outputs" bus and a valve fault flag would be on the "Faults" bus.
- RTW-EC generates clean, compact, and readable C code and provides many additional optimization options for fine control over the efficiency of the generated code.

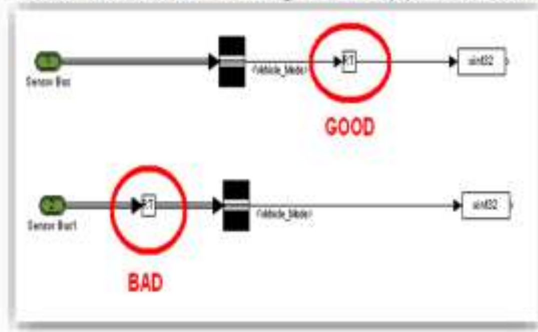
Pay attention to sample times on target ECU



- The faster the sample time, the higher the CPU loading, so choose appropriately for the system
- Use multiple sample times where possible
- But keep the number of sample times in a given model under check. Rate transition makes a copy of the signal or entire bus
    - Put rate transition blocks on the signal as opposed to entire bus

Use multiple sample times where possible

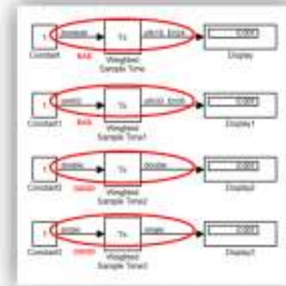
- If an input changes slowly, no reason to process it like fast inputs.
- But keep the number of sample times in a given model under check. Rate transition makes a copy of the signal or entire bus
  - Put rate transition blocks on the signal as opposed to entire bus



Pi Innovo

Work in CPU's native precision

- 64 bit Vs. 32 bit math
  - Some Simulink blocks result in 64 bit floating point operations, which are much slower
    - Example: the square root function used about 1458 clock ticks (double) vs. 268 clock ticks (single). That is ~5x slower
  - If block usage is unavoidable, some custom code can be included to force the compiler to use single precision math



- Avoid usage of certain Simulink inbuilt functions such as Abs(Absolute), Sqrt(Squareroot), Cos(Cosine) etc. that define their input arguments as double (precision) instead of single like the rest of the floating point functions.
- Similarly, a few modeling choices will result in one or more of the following functions that use double precision math to be called. Only a review of the RTW generated C code can confirm that these functions are not being used.

    sin, sinh, asin, cos, cosh, acos, tan, tanh, atan, atan2, floor, round, trunc, ceil, fmod, erf, erfc, frex, modf, hypo, pow, exp, log10, log, gamma, lgamma, ldexp, j0, j1, jn, y0, y1,yn. Weighted Sample Time block as well.

-If the use of these functions cannot be avoided, then a work around can be applied which will force the compiler to replace the functions with appropriate single precision functions. The work around involves, using the Configuration Parameters>Real-Time Workshop>Custom Code
- Use "Data Type Replacement" for embedded coder

Custom code to force single-precision

```
#define sqrt(x) sqrtf(x)
#define cos(x) cosf(x)

etc.
```

- For Embedded Coder, Data Type Replacement must be enabled.

Custom code to force single precision

Configuration Parameters: SAS_Main/Configuration-OpenECU-RTW-EC (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
  - Sample Time
  - Data Validity
  - Type Conversion
  - Connectivity
  - Compatibility
  - Model Referencing
  - Saving
- Hardware Implementation
- Model Referencing
- Simulation Target
  - Symbols
  - Custom Code
- Real-Time Workshop
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
  - Code Style
  - Templates
  - Data Placement
  - Data Type Replacement

Use the same custom code settings as Simulation Target

Include custom c-code in generated:

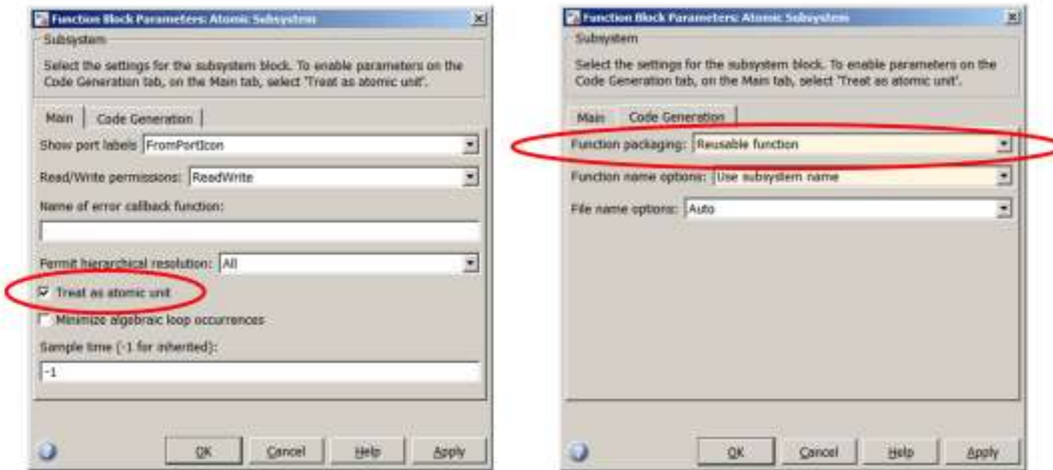
- Source file
- Header file
- Initialize function
- Terminate function

Source file:

```
#define floor(x) floorf(x)
#define sqrt(x) sqrtf(x)
#define pow(x,y) powf(x,y)
#define cos(x) cosf(x)
#define sin(x) sinf(x)
#define tan(x) tanf(x)
#define fabs(x) fabsf(x)
#define asin(x) asinf(x)
#define acos(x) acosf(x)
#define atan(x) atanf(x)
#define atan2(x) atan2f(x)
#define exp(x) expf(x)
#define log10(x) log10f(x)
#define log(x) logf(x)
#define round(x) roundf(x)
#define trunc(x) truncf(x)
#define ceil(x) ceilf(x)
#define fmod(x,y) fmodf(x,y)
#define erf(x) erff(x)
#define erfc(x) erfcf(x)
#define hypot(x) hypotf(x)
#define lgamma(x) lgammaf(x)
#define j0(x) j0f(x)
#define j1(x) j1f(x)
#define jn(x,y) jnf(x,y)
#define y0(x) y0f(x)
#define y1(x) y1f(x)
```

Pi Innovo

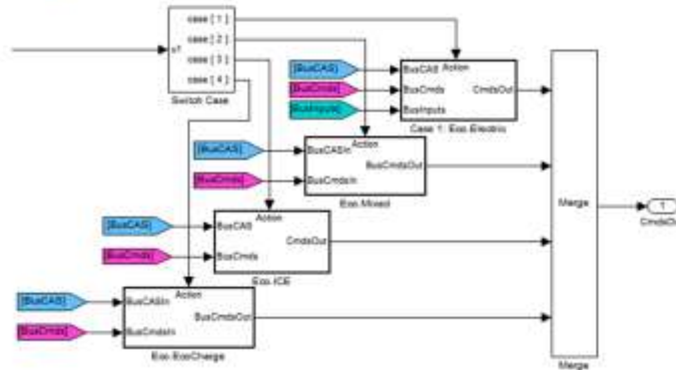
Code size – use atomic reusable functions where possible



- Simulink will code the entire model in a single function if you let it.
- Atomic is a Simulink term which means the blocks inside the subsystem execute together, so you can treat them collectively as one block.
- Convert frequently used function into atomic reusable
    - Results in the atomic subsystem coded as C function instead of code duplication.
    - Breaks up code into smaller chunks which reduces build time
        - Some compilers can’t handle large functions and run out of memory during build time

Avoid data store blocks if possible

- Read and write tasks could preempt each other – race conditions
- Use merge blocks instead for if/else or switch/case



- Cannot guarantee execution order with data store operations
- Data stores can lead to debugging difficulty (race conditions)

RAM usage – choose bus type carefully

Virtual bus

- Virtual bus does not allocate specific memory for the bus
  - The bus just makes the model look cleaner

Pro:

- Uses less memory
- Runs faster
- Multiple sample rates

Con:

- Cannot cross certain model boundaries

Non-virtual bus

- Non-virtual bus allocates specific contiguous memory for the bus
  - Represented in the generated code as a struct

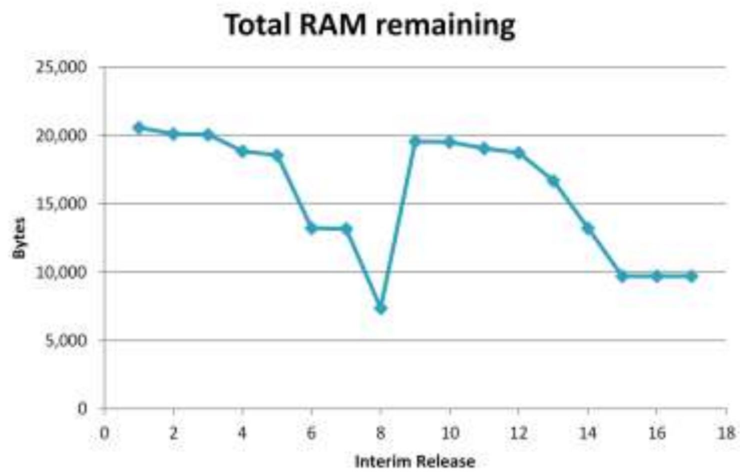
Pro:

- Cleaner code generation (easier to debug)
- Can cross all model boundaries

Con:

- Can result in frequent copies (more memory)

Optimizing with non-virtual bus: reduce bus copies



- We were running out of RAM with the model only 50% complete
- By looking at the generated code, we saw where bus copies were made and worked to eliminate the bus copies

RAM usage – non-virtual bus optimizations

Non-virtual bus considerations

Issue: Non-virtual buses may incur copies when passed across boundaries such as into a Stateflow chart, a model reference block, an atomic subsystem etc.

Fix: Unpack the bus outside the subsystem and pass only the necessary signals

Issue: Non-virtual buses may incur copies during bus assignment

Fix: Only make one bus assignment per atomic subsystem.

General recommendations

1. Spend time creating a robust architecture – even in the prototype stage
2. Create customized guidelines to suit the needs of the project and the team
3. Keep track of build stats (e.g. model build time) and ECU resources usage (e.g. RAM usage, CPU utilization, execution loop time) to catch problems before they happen
4. Perform manual optimization on a regular basis
   - Including reviewing generated code
5. Use Model Advisor

Model Advisor enables you to check models for conditions and configuration settings

What optimization options have been applied

What diagnostics options have been enabled

Those options in turn check for things like

Unconnected blocks in the model

Dead code

Data overflow etc

Pi Innovo

info@pi-innovo.com
Pi-innovo.com

734.656.0140