McGill University
School of Computer Science
COMP 621

Optimizing Simulink® Models

Report No. 2014-05

Bentley James Oakes
bentley.oakes@mail.mcgill.ca

April 27, 2014

Contents

List of Figures

List of Tables

Abstract

The Simulink$^{®}$ modelling tool is used to diagram and study cyber-physical systems. One advantage of modelling the systems in this way is that embeddable code can generated from the models directly. However, this process means that inefficiencies in the model may be propagated to the code. Code generation optimizations are available, but may lead to an unacceptable loss of traceability in determining which parts of the model were modified or removed during code generation.

In our work, we focus on defining model-to-model optimizations. This means that the optimized model can be loaded back into Simulink for further development or analysis, improving traceability and allowing model specialization for different platforms. An analysis framework has been created, based on dataflow analysis from the compiler optimization domain. This allows fast and accurate definition of new optimizations. As well, an initial optimization classification was developed to aid in the discovery of new optimizations. At the present time, we have implemented three optimizations in our framework: constant folding, dead-block removal, and hierarchy flattening. These optimizations are intended to simplify the model and potentially increase model performance when simulated.

Our framework allows us to communicate directly with a Simulink instance to import and export models, allowing us to test these optimizations on a number of sample Simulink models. In order to test the performance benefits of our optimizations, our experiments generated simulation code for the model before and after optimization. Examining the run-time of these simulations indicated that the constant folding optimization decreased the run-time on all applicable models when code generation optimizations were not used. This decrease in run-time is well above the fraction of a second that the analysis and transformation took for this optimization. Thus, a net gain in performance was obtained. Other optimizations did not show performance results, but did produce a simplified model, which is also desirable for a modeller.

1 Introduction and Motivation

Model-driven software engineering promotes models as being the primary artifact in the software design process. In the cyber-physical system domain, these models are executable representations of the system under design [3]. For example, the Simulink$^{®}$ modelling tool allows the representation of such systems as the dynamics of an inverted pendulum [7]. Models created may also be synthesized directly into code to be embedded into hardware, allowing rapid design and deployment of cyber-physical systems. However, this code generation process could propagate inefficiencies in the model representation to the synthesized code and thus the final system. Another use for models is in the direct simulation of a system. Inefficiencies are not desirable in this workflow either, as it is advantageous to execute the simulation as quickly as possible. Any inefficiencies should therefore be detected and resolved.

Our work presents a method of defining optimizations that operate on causal-block models with the aim to increase the performance of the model when simulated or synthesized, and/or to simplify the structure of the model. We then use this definition process to create a number of optimizations composed of analyses and transformations. The analysis step provides information about the semantics of the model, which is then used to transform the model in a second step. This produces a second model which should be more desirable than the original.

The optimizations presented are inspired by known compiler techniques, as well as some optimizations performed by Simulink. Of note is that the optimizations performed in Simulink are performed when the code for a model is generated. This reduces traceability, as the modeller cannot easily see the differences between the original model and the generated code. Our framework adds to the traceability of the optimization process, as the same modelling formalism is retained for both the original and transformed model. Our method also allows for model verification tools to operate on the transformed model and guarantee correctness.

Our work has produced an analysis and transformation framework to better understand the optimizations that can be performed on causal-block models. We have implemented three optimizations and executed them on a number of sample models. Analyzing the results allows us to see the performance and simplicity gains from these optimizations.

Our specific contributions include:

- Definition and classification of optimizations applicable to causal-block models
- Formulation of structured approach to defining optimization analyses
- Framework for performing analyses and transformations on Simulink models
- Implementation of three optimizations within framework, with example models and timing information

Section 2 contains an overview of the causal-block domain, the Simulink tool, and a review of related work. Following that, section 3 discusses our classifications of model optimizations, while section 4 describes the process of defining an optimization, using constant folding as an example. The optimizations implemented are presented in section 5 along with sample models. Our framework and results will be presented in section 6, and section 7 will conclude with our final remarks and a brief discussion of future work to be performed.

2 Background and Related Work

This section will provide a brief introduction to the causal-block formalism, as well as the Simulink tool. Related work on the optimization of causal-block models will also be presented.

2.1 Causal-block Models

A causal-block model is a typed graph, where the nodes are termed 'blocks', and the edges termed 'links' [11]. Each block in the model represents a function such as addition or integration. These blocks operate on data in either scalar and vector form, which is passed along the links between blocks. During the execution of a model, a block will receive some data as input, perform a function, and then produce some output value. Note that this implies a causal ordering to the block's execution, where some blocks must execute before other blocks in order to produce the required values. Figure 1 shows a functional description of arithmetic, constant, and delay blocks. The arithmetic block takes multiple inputs and produces an output based on the operator selected. A constant block produces a constant output at every time-step. The delay block produces an initial value at the first time-step, and then echoes the input from the previous time-step on all other time-steps.

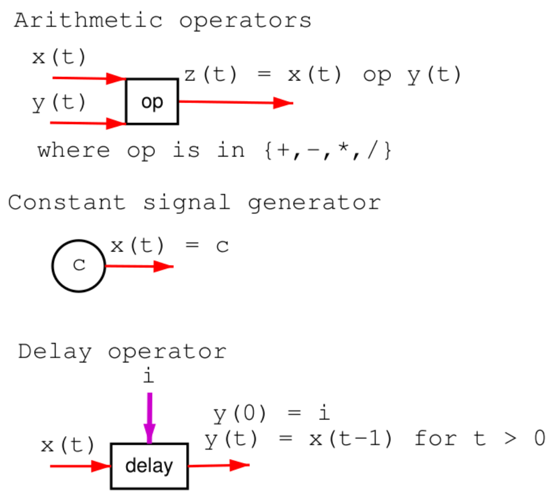
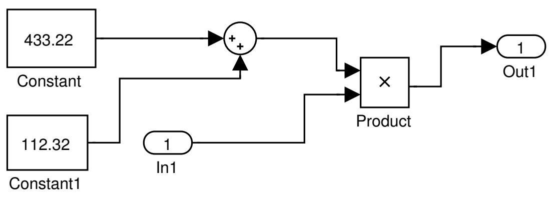


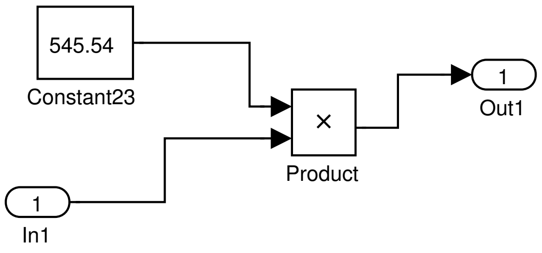
Figure 1: Functional description of three blocks (from [11])

When a model is simulated, or code is generated, a scheduler must ensure that blocks are executed in the correct ordering to ensure data dependencies are preserved. Each block is then be executed in turn to produce data for the next block. Optimizations can speed up this execution by reducing the number of blocks to execute, or otherwise making blocks less computationally expensive.

For instance, the model in Figure 2a has two constant blocks, whose result is summed together. This data is then multiplied by an incoming signal to the system, before being output out of the system. This system is inefficient, as the summation result will always be a constant but will still be calculated every time-step. Therefore, those blocks can be replaced by a constant in the constant folding optimization, further explained below. The transformed model can be seen in Figure 2b. Note that the transformed model contains less blocks and edges, which may offer a performance benefit when the model is simulated.



(a) Before



(b) After

Figure 2: Causal-block model before and after constant folding

2.1.1 Simulink

The Simulink tool from The MathWorks Inc. is a modelling environment that provides a visual workflow to create causal-block models [8]. The focus of Simulink is to create industrial cyber-physical system models, which can then be simulated, or generated into code to be directly embedded in a device.

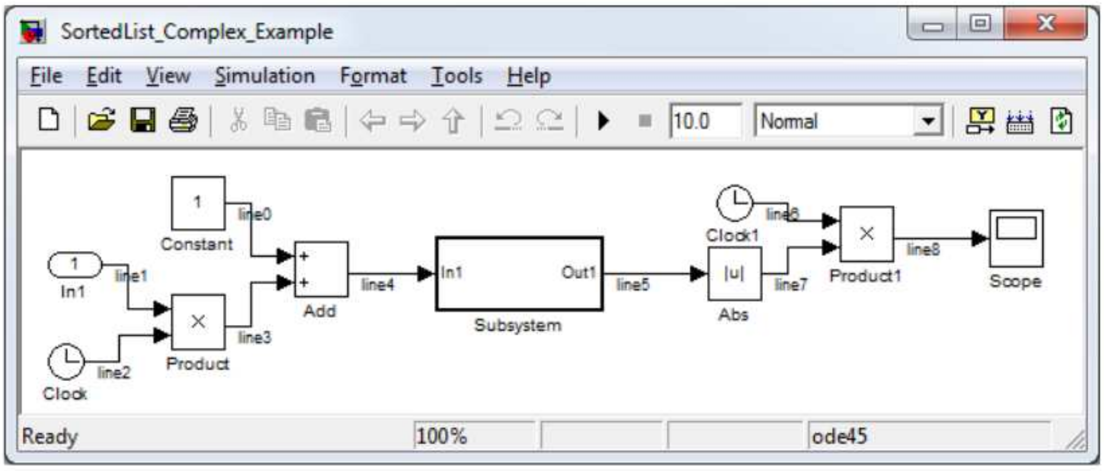


Figure 3: A Simulink model (from [4])

An example of a Simulink model within the Simulink tool is seen in Figure 3. All blocks in our work will be described as needed. Note the in port, which represents a component to communicate with other parts of the system or the environment. Data can be sampled from these sensors at a defined sample rate, which defines the rate of calculation in the model.

2.2 Related Work

Our work is built upon that of Pussig *et al.* [13]. In their work, they explicitly model the computational semantics and solvers for the simulation of Functional Mock-up Units. For the models to be simulated, this involves the replacement of integrator blocks with discrete-time approximations. As well, they also execute a simple constant-folding transformation on their benchmark model. Their results indicate that these transformations can offer significant speedup, with the constant folding optimization increasing the performance of their simulated model by 10 percent. In our work, we have implemented constant folding using our analysis and transformation framework, and seen similar performance gains.

Denil *et al.* have graciously provided their code to communicate with Simulink instances. In their work, they demonstrate the use of model transformations on Simulink blocks, in order to discretize and approximate Simulink models. This was done in order to discretize and optimize a model for code generation, similar to the optimizations we describe in this work.

The work of Barroca *et al.* investigates how model transformations themselves can be used in order to perform an analysis in the causal-block diagram domain [2]. Their transformations operate on blocks, and generate reaching definition information for each block. Their analysis procedure is similar to the one we have developed for causal-block models.

Transformations have also been used in order to remove hierarchical constructs in Simulink models [5]. This is done in order to explicitly model a flattening process which is done during simulation and code generation. The authors state that explicitly modelling this flattening process offers the possibility of transformation reuse, as well as enabling verification techniques to be performed. Another work by the same authors employs transformations in order to replicate data type analysis on Simulink models [4]. The Simulink tool performs this operation in code, but the authors indicate that there are benefits to explicitly modelling the logic that inferences data-type. We have implemented the flattening optimization in our framework, and are working towards an implementation in model transformations.

3 Optimization Classification

We propose a classification for causal-block models, based upon the platform-dependence of the optimization, as well as the optimization intent. We believe that defining classifications for optimizations will create stronger theoretical connections to the compiler domain, as well as to transformations from the model transformation field.

Similar to the use of 'optimization' in the compiler domain, note that we are overloading the term in this work as well. Not all optimizations will produce a performance benefit, and some may even degrade performance in certain cases. We are using 'optimization' to suggest that the modeller will experience some benefit from the transformed model, such as increased readability or platform suitability.

3.1 Platform Dependence

An important characteristic of code optimizations performed by a compiler is whether the optimization is applicable for all computer architectures or some subset. This categorization can also be applied to model optimizations. We distinguish and provide brief examples for three 'levels' of generality for optimizations. A later section will classify the optimizations implemented in our framework.

3.1.1 Model-level

The model-level optimizations are those that are not dependent in any way on the target architecture. These optimizations focus on changing the structure itself of the model. One example is dead-block removal, where blocks that are not needed for calculation in the system should be removed. Other optimizations include mathematical operations such as algebraically simplifying models to avoid extra calculations.

Examples of model-level characteristics to optimize:

Constant blocks Blocks that produce a constant output at all time-steps of model execution could be replaced by a constant block

Dead-blocks Blocks that do not affect the output of the system should be removed

Flattening Hierarchical systems-within-systems may be flattened to unlock further optimizations

Algebraic simplification Mathematical blocks may be simplified. An example is collapsing two gain blocks into one

Single-/Multi-rate sampling A system may have multiple inputs that are sampled at different periods. A model layout that is aware of these sample rates may be more efficient

3.1.2 Platform-independent

A second level of optimization generality can be formulated, where optimizations are specific to a general class of target architecture. These general classes may include whether the target is single- or multi- core, or the target programming language of code generation. Example optimizations of this level include transforming float calculations into integer representations, or specializing blocks into structures appropriate for a given target language.

Examples of general architecture characteristics affecting optimizations:

Target language The semantics of declarative versus functional versus circuit languages may influence the model design

Number of target cores Parallel optimizations may be identified or performed on the model

Processor features Calculations could be optimized for a SIMD processor or GPU

Floating-/fixed- point calculations Precision of calculations could enable or prevent some optimizations

3.1.3 Platform-dependent

Lastly, the platform-dependent optimizations can be characterized by their dependence on a particular target architecture. Examples include rearranging the model in order to take advantage of a particular machine's caching strategy.

Some platform characteristics affecting optimization:

Cache strategies Data locality optimization depends on cache size and strategies

Processor heterogenity Blocks could be marked to be executed on a particular processor

3.1.4 Level Hierarchy

Note that the classification levels can be placed in a hierarchical relationship as in Figure 4, with optimizations further down the hierarchy more specific to a particular machine. As well, some optimizations may be split up to operate on different levels. For example, a platform-dependent constant folding optimization that is targeting a non-floating point machine may be able to perform a more rigorous analysis of a model and eliminate more blocks than the model-level version.

Another classification of optimizations could be those transformations that specialize a model further down the hierarchy. This is represented in Figure 4 by the inter-level arrows versus the intra-level arrows. These optimizations could be part of a code synthesis workflow, where a general model is specialized further and further until code is generated from a platform-specific model. As well, Pussig *et al.* note that model optimizations may allow the complexity of the code generator or of elements in the simulation environment to be reduced [13].

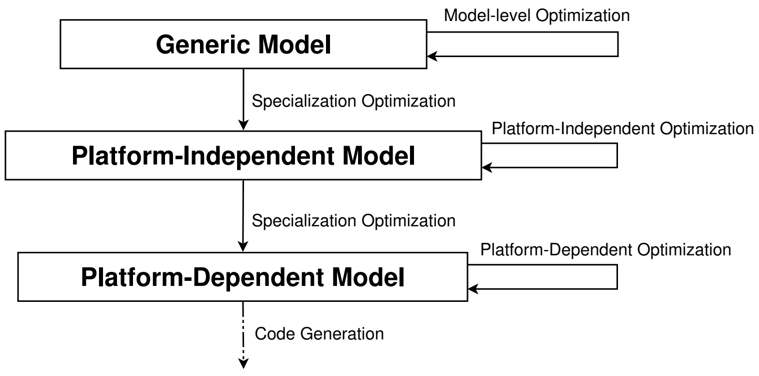

Model-level Optimization
Generic Model
Specialization Optimization
Platform-Independent Optimization
Platform-Independent Model
Specialization Optimization
Platform-Dependent Optimization
Platform-Dependent Model
Code Generation

Figure 4: Hierarchy of optimization generality levels

3.2 Optimization Intent

We also look to model transformation catalogues for inspiration in classifying optimizations. For instance, the work of Amrani *et al* provides categories which can also be applied to model optimizations [1]. Here we discuss four of their transform classifications as applied to model optimizations. Further research may more precisely define this relationship and determine possible advantages of a unified classification.

The *refinement* and *abstraction* transformations of Amrani *et al* act on models to produce more or less platform-specific models. This is equivalent to the inter-level optimizations previously discussed.

Approximation transformations lower the precision of model calculations in order to raise performance. This is also applicable to a causal-block model, as in transforming blocks into integer versions.

The *optimization* transformation raises performance, defined above as a model-level optimization in our classification.

Refactoring transformations may improve 'certain quality characteristics' of the model such as their visual appearance. For causal-block models, we have noted certain optimizations as 'style optimizations'. For example, a model's structure may be transformed from a data-flow orientation design to a control-flow orientated design by rearranging switch blocks. This optimization would allow the modeller to model in their preferred style, potentially reducing errors from unfamiliar model design. Another example of a refactoring optimization would be a 'feedback optimization' to analyze the model and annotate blocks that could perform a potentially erroneous operation such as a divide by zero. This would provide visual feedback to the modeller directly as to where error-handling techniques are required.

4 Defining Optimizations

This section will describe the process of defining an optimization for the causal-block domain. An optimization is considered to be composed of two parts: the analysis, and the actual transformation.

4.1 Analysis Definition

Our research has found that the analysis of some optimizations is extremely similar to those performed in the compiler domain. In this section, we introduce a six-step analysis framework for compilers, and describe how it may be modified for the causal-block domain.

4.1.1 Dataflow Analysis Steps

Dataflow analysis is structured upon divided program code into separate blocks, and examining how data flows through each block. An approximation for the analysis is then given for each of these blocks. The six-step procedure given here is based upon work by Hendren and Thibodeau [6, 14]. Further details on dataflow analysis can also be found in [10].

1. State the problem precisely
   - Stated for a program point p, and concerning paths to and from the start and end blocks
   - Example: Which program variables have been defined above this line of code.
2. Define the approximations to build, as well as a partial ordering for these approximations
   - Example: Sets of variables for live variable analysis. The ordering is based on set inclusion.
3. Determine if the analysis is a forwards or backwards analysis
   - Given a topological sort of the code blocks, does the analysis start at the beginning block and move forwards, or the end and move backwards?
4. Given an approximation for two parent blocks, how are the approximations merged
   - Usually intersection or union
5. Write the flow equations for each statement in the language
   - How does each statement affect the approximation for that statement?
6. State the starting approximations
   - What is the approximation at the start of the analysis?
   - If there are loops in the control-flow graph, what assumption is made about the looping approximation?

4.2 Analysis for Causal-Block Models

This six-step procedure will now be used as a basis for the constant folding analysis performed in that optimization. Consider the model in Figure 5. Note that the constant blocks on the left hand side will produce a constant value for each time-step of the model. As well, the sum block will also produce a constant value. Before the summation block can be replaced by a constant block, this optimization must identify the constant value to be produced.

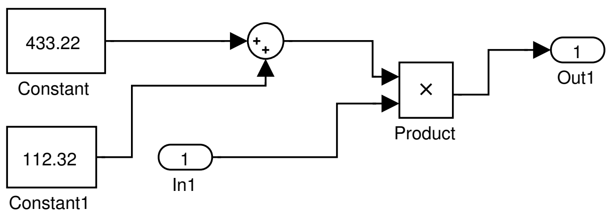


Figure 5: A model that can have constants folded

1. State the problem precisely
   - Does a block in the model produce a constant value for every time-step?
2. Define the approximations to build, as well as a partial ordering for these approximations
   - For each block, we will assign a constant value if it can be determined, a value BOTTOM if the value is unknown, and a value TOP if the value is not a constant. For the ordering, BOTTOM is below all constant values, which are below TOP

3. Determine if the analysis is a forwards or backwards analysis
    - This analysis is a forward analysis. The constant values are propagated forwards from the constant blocks
4. Given an approximation for two parent blocks, how are the approximations merged
    - Approximations are not merged in the forward direction for causal-block models. This is discussed below in detail
5. Write the flow equations for each block in the causal-block library
    - For a constant block, the approximation is the blocks' value
    - For an arithmetic block (including mux and demux blocks):
        - If any input is BOTTOM or TOP, the approximation is TOP
        - If all inputs are a constant value, the approximation is the operator performed on the inputs
    - For a switch block, determine if the switch value is known at analysis-time and produce that output
    - For a delay block, produce a constant value approximation if the initial condition and the input value are the same
    - For all other blocks, a value of TOP is produced
    - Note that future work must provide precise equations for all blocks in the Simulink library
6. State the starting approximations
    - All blocks are assumed to produce a BOTTOM value at the beginning of the analysis

We now address the issue of removing the merge operation step for forward direction analyses. This step is used in dataflow analysis to merge together two approximations when code execution could have taken two different paths. We argue that this situation cannot happen for forward analyses for causal-block models. The point at which a line is input to a block is termed a 'port', where each port can only be the destination of one line. This means that there cannot be two different approximations meeting at one port, and thus no need for a merge operation in a forward analysis.

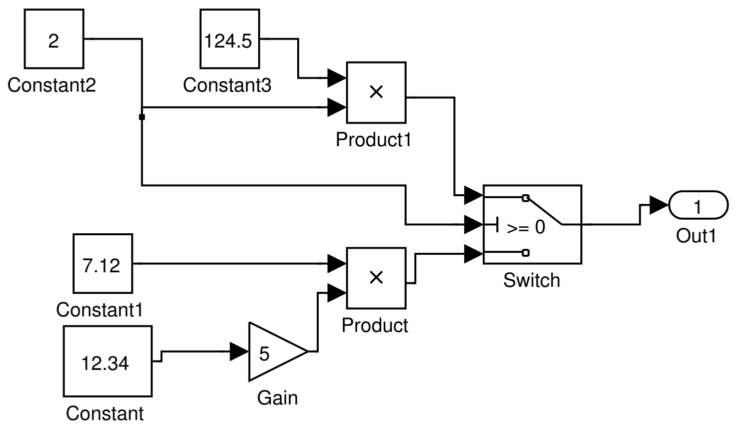


Figure 6: No two values for the same line

Figure 6 contains a switch block, which is roughly equivalent to an 'if' statement from dataflow analysis. Note that the middle line is the signal line. Currently, the switch block is set to pass through the top value if the signal line is greater than or equal to zero. Otherwise, the bottom value will be passed through the block. Again, we note that there is no concept of 'merging' data lines when moving in a forward-direction analysis. The switch block itself will determine which data or data approximation should be output.

In a backward-direction analysis, we note that lines do merge and thus a merge operation is needed in this case. This will be seen in the definition of the dead-block optimization analysis.

5 Optimizations

This section will provide precise details for the three optimizations currently implemented in our framework. For each, the applicability of each optimization will be discussed, the platform-dependence will be indicated, and example models will be provided. A later section will discuss the experimental framework used, and the timing information for executing these optimizations.

5.1 Constant Folding

The first optimization we shall examine is constant folding as described above in Section 4.2.

5.1.1 Description and Benefit

Constant folding is the process of determining which blocks in a model only produce a constant value, and then replacing those blocks with a constant block. The benefit of this analysis is that removing blocks from a model may reduce the model's execution time in simulation or code.

5.1.2 Context

Constant folding is applicable when there are constant values in the model. In our example models, these values are provided by constant blocks. The actual performance benefit of performing this optimization is based on the number of constant values present in the model, and the structure of how they are used.

5.1.3 Level

This is a model-level optimization. The model should be transformed into an equal or smaller sized version, without a change in semantics. Note that as mentioned before, some versions of constant-folding that differentiate between integer and floating-point numbers may be more adequately classified as a platform-independent optimization.

5.1.4 Analysis

The analysis procedure has been described previously in section 4.2 and will be skipped for brevity.

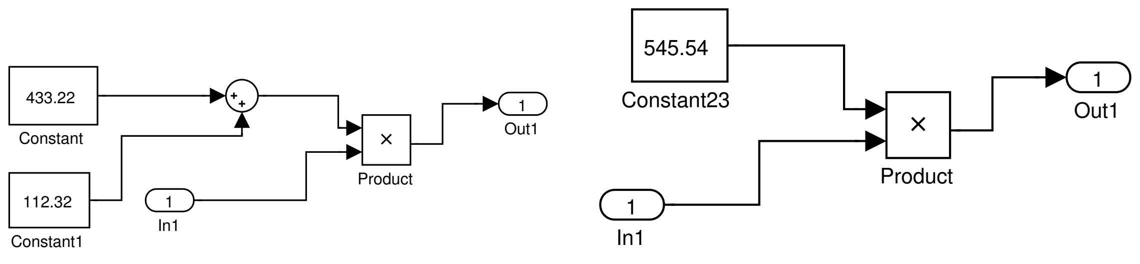
5.1.5 Transformation

The analysis of this optimization will indicate which blocks in the model can be replaced with constant blocks. The transformation therefore becomes relatively simple.

1. Create a new constant block with the constant value as determined in the analysis
2. Connect output of new constant block to all children of block to be replaced
3. Remove replaced block
4. Determine dead ancestor blocks of replaced block and remove them

Note that this last step is itself another optimization – dead-block removal.

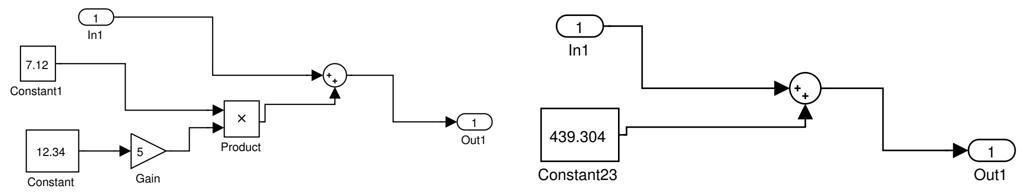
5.1.6 Example Models



(a) Before (b) After

Figure 7: Constant Folding Model 1 - Before and after optimization

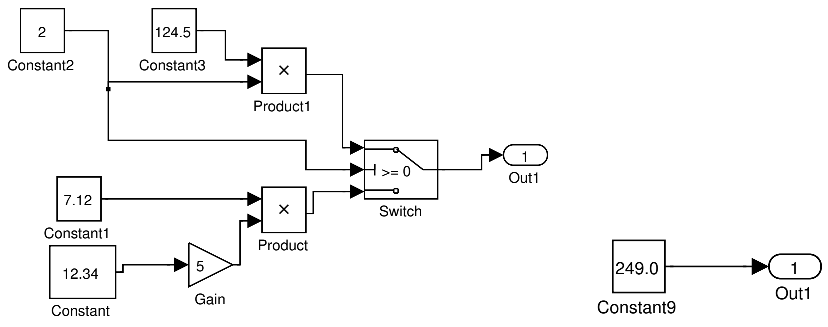
Figure 7 is a simple test of the constant folding optimization. As noted above, the constant blocks and summation block in Figure 7a can be replaced with a constant block, as seen in Figure 7b. Note that a further optimization could be performed. In the work of Denil *et al.*, they perform an optimization where a constant block and a product block are transformed into a gain block, which performs the same mathematical operation. This optimization has not yet been implemented in our framework, but would be applicable in this example.



(a) Before (b) After

Figure 8: Constant Folding Model 2 - Before and after optimization

Figure 8 demonstrates how a larger number of blocks can be removed by this optimization.

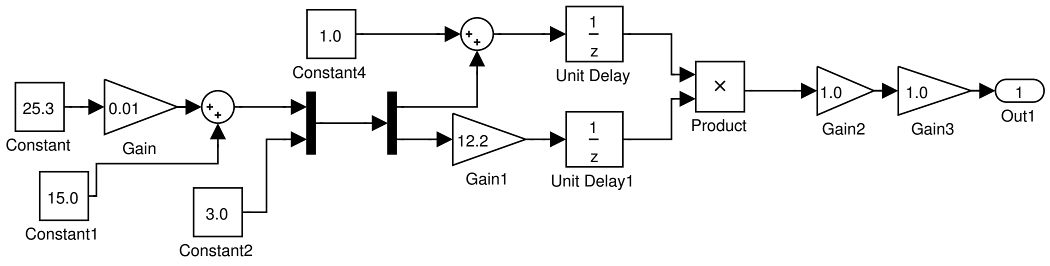


(a) Before (b) After

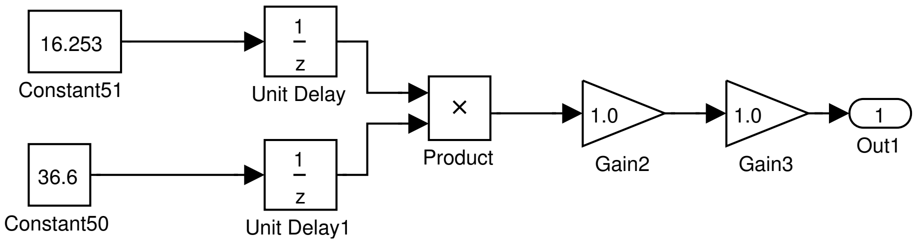
Figure 9: Constant Folding Model 3 - Before and after optimization

Figure 9a contains a switch block, where the branch to take is known at analysis-time. The top input will be output from the switch block, and since the top input is a constant, the switch block can be replaced by a constant block in Figure 9b.

Figure 10 contains a number of Simulink elements which can be optimized away. Of note are the thick black lines in the middle of Figure 10a. These are mux and demux blocks. The mux block takes the input



(a) Before



(b) After

Figure 10: Constant Folding Model 4 - Before and after optimization

data and places it into an array ordered by port number. The demux reverses the process. Therefore, the mux - demux pairing is an identity operation which does not affect the data in the lines.

In Figure 10b, a number of elements remain. The delay blocks have different initial conditions then their input values. As such they cannot be replaced by constant values. We also note three other possible operations. First, it may be possible to move the product block before the delay blocks and combine the delay blocks into one. Second, gain blocks of value one are identity operations and can be removed. Third, two gain blocks in a row may be combined into one gain block. Our future work will attempt to implement these optimizations as well.

5.1.7 Other Comments

In the Simulink tool, a constant folding optimization can be applied if there are model parameters that are constant throughout the execution of the model. These parameters are essentially named constants, and can be folded into the model. Note that technical issues prevented this version of constant folding in our current implementation.

5.2 Dead-block Removal

5.2.1 Description and Benefit

Dead-blocks are blocks whose result does not affect the output of the system. Therefore, these blocks should be removed in order to make the model visually simpler, and to avoid the potential for wasted calculations.

5.2.2 Context

Dead-block removal can be used on any model to remove all useless blocks. As well, dead-block removal can be utilized in other optimizations. For example, in constant folding a block is replaced by a constant block. The old block and some of its ancestor blocks will then become dead and can be removed.

5.2.3 Level

This is a model-level optimization, as it does not depend on the target platform.

5.2.4 Analysis

The analysis for the dead-block removal analysis is also a dataflow analysis. Therefore it can be described using the six-step analysis defined previously.

1. State the problem precisely
   - Can the output of a given block affect the output of the system?
2. Define the approximations to build, as well as a partial ordering for these approximations
   - For each block, we will assign a value REMOVE if it should be removed from the system, or KEEP if it should be kept
3. Determine if the analysis is a forwards or backwards analysis
   - Backward. We wish to determine which blocks are connected to the output
4. Given an approximation for two parent blocks, how are the approximations merged
   - Given approximations from different child blocks, if any is KEEP, then the merged approximation should be KEEP. Otherwise the approximation should be REMOVE
5. Write the flow equations for each block in the causal-block library
   - Scope blocks provide output to the user, while outport blocks connect to other systems. Their approximation should be set to KEEP.
6. State the starting approximations
   - All blocks are marked as REMOVE at the beginning of the analysis

5.2.5 Transformation

For this transformation, all blocks marked as REMOVE should be removed from the system.

5.2.6 Example Models

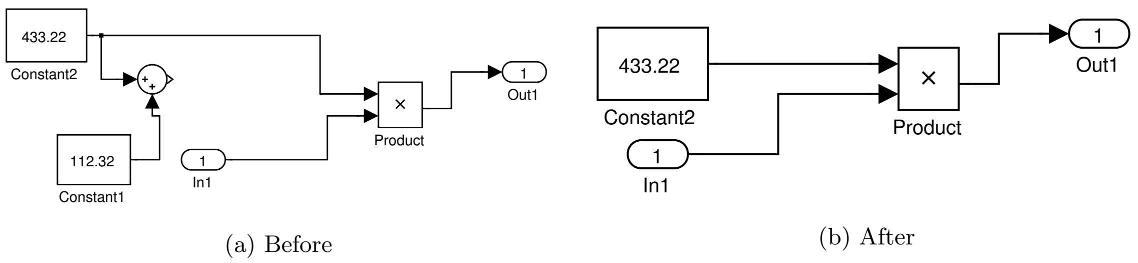


Figure 11: Dead-Block Removal Model 1 - Before and after optimization

Figure 11 demonstrates that blocks that do not connect in any way to the output can be removed. In this case, the lower constant and summation block on the left-hand side of Figure 11a are removed.

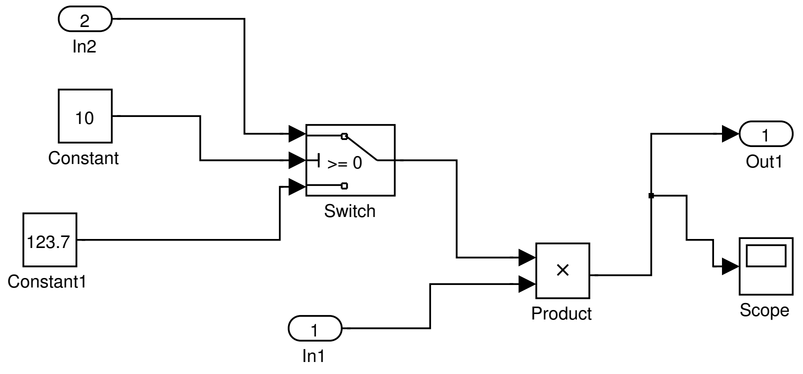


Figure 12: Dead-Block Removal Model 2 - Optimization does not change model

Figure 12 illustrates a complex version of dead-block removal that has not yet been implemented. In this model, the switch block will always output the top value received. Therefore, the bottom and signal lines can be removed, and the top line connected directly to the signal block's children. This optimization would be composed of two analyses. The first analysis would propagate constants down each line, similar to constant folding. The second analysis would determine if any switch block had a constant signal. An even more complex optimization may also determine the possible range of values each line could take, in order to determine the behaviour of the switch.

5.3 Flattening

5.3.1 Description and Benefit

A causal-block model may be structured in a hierarchical manner. For example, in Simulink subsystem blocks have input and output ports, which connect to blocks within the subsystem block. Figure 13a shows a model with a subsystem block, while Figure 13b shows the inner subsystem.

There are a number of benefits to performing a flattening procedure on the model. First, a modeller may wish to see all blocks involved in the model. Second, a flattening transformation must be performed before code generation in any case. Performing this step ahead of time may increase the speed or reduce the complexity of the code generation step. Third, the flattened model may allow further optimizations to be performed such as constant folding. This can be seen in Figure 13c, where the inner product block can be replaced with a gain block.

5.3.2 Context

Dead-block removal can be used on any model with subsystems. However, it may not be desirable for all subsystems in the model to be flattened. The modeller should indicate which subsystem should be flattened. This control is not currently implemented in our framework.

5.3.3 Level

This is a model-level optimization, as it does not depend on the target platform.

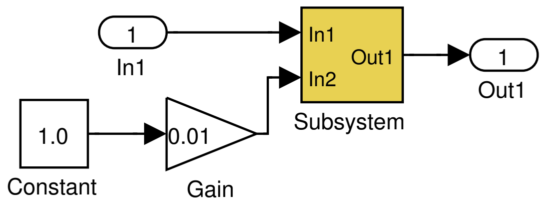
5.3.4 Analysis

The only analysis that must occur in the flattening optimization is to find all subsystems which are not the root system. As such, it is not a dataflow analysis. However in the current implementation, this analysis uses the framework defined for the other optimizations to locate all subsystems.

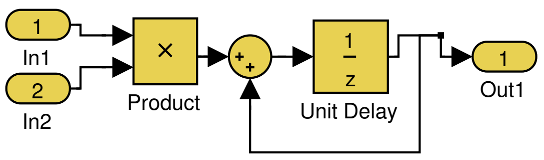
5.3.5 Transformation

The transformation must correctly 'rewire' all input and output lines of the subsystem to be flattened. This is a non-trivial transformation as the line connected to a particular input port inside the subsystem must be rewired to connect to the input on the outside of the subsystem. We omit details here for brevity. Note that a model transformation method for performing flattening is provided by Feher *et al.* [5]. We intend to implement a model transformation version of flattening as this may allow us to formally verify the transformation [9].

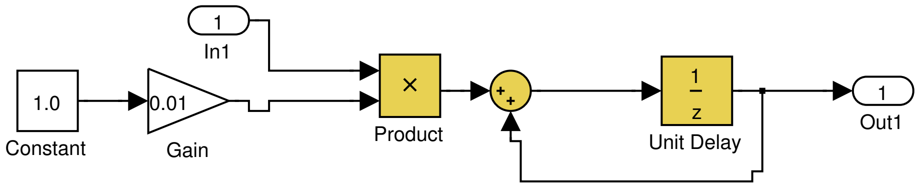
5.3.6 Example Models



(a) Outer model



(b) Inner subsystem

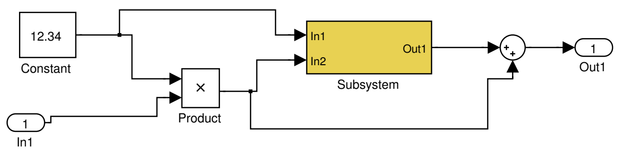


(c) Transformed model

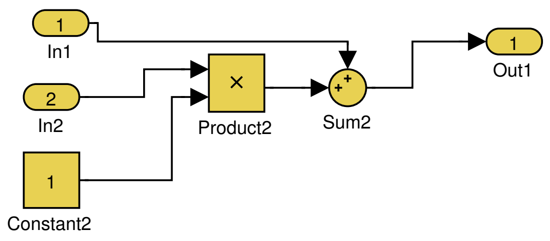
Figure 13: Flattening Model 1 - Original model and subsystem, and transformed model

Figure 13a is a model containing one subsystem, highlighted in yellow. Figure 13b shows the blocks within that subsystem. The result of flattening these subsystems is Figure 13c. As before, the yellow highlight shows the blocks that originated from within the subsystem.

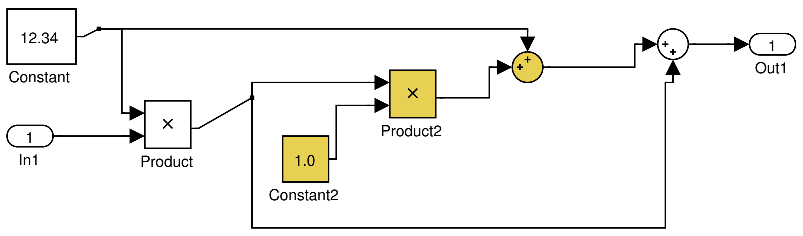
Figure 14 is another example of the flattening optimization. The original model and subsystem are Figures 14a and 14b, while the flattened model is Figure 14c. Careful readers will note that the names for the blocks within the subsystem are different than the blocks in the outer model. This is due to a technical issue, where two blocks with the same name could not be added to the same model when exporting back to Simulink. This will be addressed in future implementations.



(a) Outer model

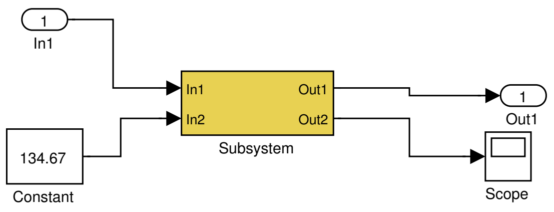


(b) Inner subsystem

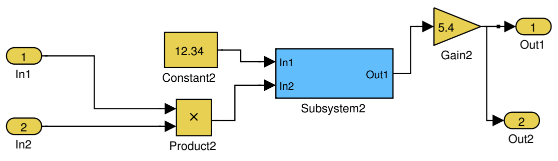


(c) Transformed model

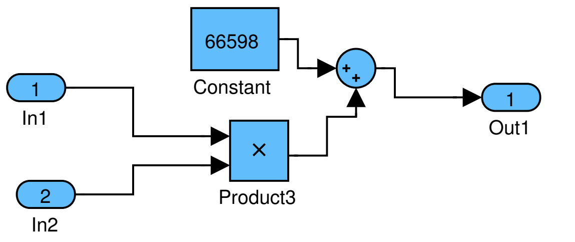
Figure 14: Flattening Model 2 - Original model and subsystem, and transformed model



(a) Outer model



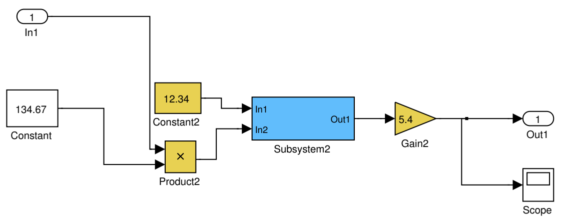
(b) Child subsystem



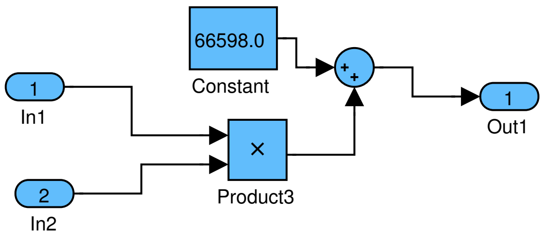
(c) Grandchild subsystem

Figure 15: Flattening Model 3 - Original model, nested subsystem, and twice-nested subsystem

Figure 15 shows the test model for a double-flattening experiment. The outer model contains a subsystem (marked in yellow) which itself contains a subsystem (marked in blue). This model could then be used to test when it was possible to flatten two or more levels of hierarchy at once. However, technical issues in the Simulink export process prevented more than one subsystem from being flattened at once. In our experiments, the outer-most subsystem was flattened resulting in the intermediate model seen in Figure 16. The subsystem in this model was then flattened, resulting in the final model in Figure 17. Future implementations of our framework will address this technical issue.



(a) Flattened top-level and child subsystem



(b) New child subsystem

Figure 16: Flattening Model 3 - Model with child subsystem flattened into top-level

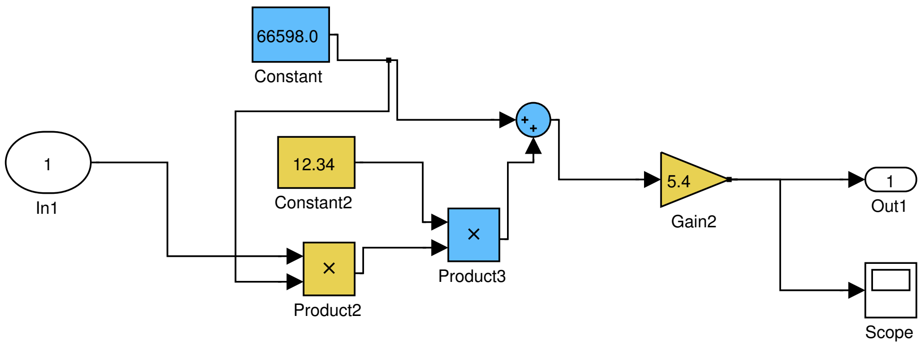


Figure 17: Flattening Model 3 - Completely flattened model

6 Experimental Framework and Results

6.1 Framework

Our experimental framework is based upon direct communication with the Simulink tool thanks to the code from Denil et al. [3]. A Simulink instance is queried for all details of the model, which are then used to build a typed graph in the Himesis format [12]. This typed graph is then transformed into connected Python objects, and connected to resemble the original model.

Once re-built in Python, the model can then be optimized. Depending on the optimization, an analyzer procedure traverses the model in a forward or backward direction and creates the needed approximations. Once completed, this analysis is passed to a transformation procedure, which transforms the model as needed. Currently, all transformations are performed in code upon the Python model. In the future, it is desirable to perform these transformations using model transformations. This would allow the transformations to be formally validated such as in [9].

Once all analyses and transformations have been performed, the Python model is written out to the Himesis format. The final step is to communicate with the Simulink instance and build the new model using Simulink's API.

Table 1 shows the timing for each component in the optimization framework when constant folding was performed on the model in Figure 10a. Results are presented for only one optimization and model as this result is representative of all results gathered. It can be seen that the majority of the time taken is to communicate with the Simulink instance and extract the model information. This is due to communication with Simulink requiring the loading of shared libraries, and the startup of a Simulink instance. In contrast, the time taken for analysis and transformation is negligible across all optimizations implemented, remaining well below one second. We note that this result means that it is feasible for an optimization to contain many analyses and transformations, extending the power of one optimization.

Framework Component	**Avg. Time (sec.)**	**Std. Dev.**
Connect to Simulink	11.71	.24
Import from Simulink	4.94	.04
Create model in Python	.08	.01
Analysis	.02	.00
Transformation	.01	.00
Export to Simulink	.01	.01

Table 1: Representative timings for performing constant folding

6.2 Experiments

Our experiments were based on performing optimizations on a number of sample models. Time and technological restraints meant that no industrial-sized benchmarks could be optimized. Future work will focus on proving our optimizations on further models. The sample models optimized can be seen in Section 5, before and after the optimization was performed. These figures show how the optimizations make the model smaller and simpler for the modeller, even if all optimizations do not lead to an increase in performance.

We measured the performance gains of our optimizations by generating simulation code for each model before and after optimization. The relevant settings can be seen in Section A.2, where the settings of special note are under ‘Optimization’. These settings denote the optimizations that Simulink performs when generating code. In particular, we believe ‘Inline invariant signals’ to be equivalent to our constant folding optimization. Therefore our experiments are run with these code generation optimizations on and off to determine their effect on the simulation performance.

The time taken to generate and compile code was recorded. However, the times varied widely even for the same model, and can offer no useful analysis. Once generated, this simulation code ran for a number of seconds to produce output data. The output data was then examined to make sure that optimization did not change the results of the model.

6.3 Optimization Results

Figure Num.	**Original Model (sec.)**		**Transformed Model (sec.)**	
	Avg.	**Std. Dev.**	**Avg.**	**Std. Dev.**
Constant Folding				
Figure 7	**19.78**	**.05**	**16.95**	**.21**
Figure 8	**18.35**	**.07**	**15.89**	**.13**
Figure 9	**23.53**	**.09**	**20.77**	**.09**
Figure 10	**18.01**	**.09**	**17.22**	**.23**
Dead-Block Removal				
Figure 11	16.79	.27	16.91	.25
Flattening				
Figure 13	18.87	.22	18.75	.19
Figure 14	21.77	.16	21.75	.38
Figure 15	19.54	.12	19.94	.06
Figure 17	-	-	20.48	.42

Table 2: Simulation timings without code gen opts

Table 2 shows the simulation timing results for when code was generated using no Simulink code generation optimizations. For each model, the simulation was run six times. The average and standard deviation are reported for both the original and transformed model. Models for which there is a significant gain in performance have their results bolded.

The constant folding optimization gave a significant performance benefit to all models. As blocks were successfully removed from these models, the simulation no longer has to generate these outputs. From the framework timings above, we note that the actual analysis and transformation for constant folding takes a negligible amount of time. Therefore, this optimization has a net benefit on performance.

The dead-block optimization did not provide a performance increase. This is most likely due to the fact that the sum operation calculation removed was proportionally small compared to the other calculations in the model. A larger number of blocks removed may show a performance increase. Note also that the second dead-block model was not simulated, as the optimization had no effect.

The flattening optimization also shows no performance increase when performed. This was expected, as the flattening step is already performed by Simulink when generating code.

Figure Num.	**Original Model (sec.)**		**Transformed Model (sec.)**	
	Avg.	**Std. Dev.**	**Avg.**	**Std. Dev.**
Constant Folding				
Figure 7	16.62	.09	16.73	.23
Figure 8	16.37	.12	16.75	.05
Figure 9	17.06	.05	16.76	.15
Figure 10	16.72	.28	16.64	.10
Dead-Block Removal				
Figure 11	16.36	.15	16.50	.07
Flattening				
Figure 13	17.24	.08	17.42	.16
Figure 14	18.36	.16	17.85	.32
Figure 15	17.97	.09	17.91	.17
Figure 17	-	-	18.00	.22

Table 3: Simulation timings with code gen opts

Table 3 shows the simulation timings when the code generation optimizations are enabled, as seen in Section A.2. In contrast to the previous experiments, these results show no significant difference between the original and constant-folded models. This is not surprising as it was expected that Simulink would perform a similar constant folding optimization during code generation. The dead-block and flattening optimizations had little effect as well, as explained previously.

7 Conclusions and Future Work

This project addresses a number of topics on defining optimizations for causal-block models. The first topic is how to classify optimizations for this domain. We propose to create a classification based upon platform-dependence, where more specific optimizations are farther down a dependence hierarchy. This could allow a specialization work-flow, where models are progressively refined until they are optimized for a particular platform. We also created connections with the intent work for model transformations, which may aid in the discovery and classification of future optimizations.

As our second contribution, we outlined a method of defining analyses for optimizations, based on that used in the compiler domain. This result was quite successful, as the procedure is very similar and allows for precise definition of analyses.

Our third result is the definition of a number of optimizations for the causal-block domain, including example models. These optimizations fit within our definition framework well, and will provide inspiration for future optimizations.

Fourth, we integrated the optimizations into a framework that can import models from Simulink, run analyses and transformations, and then export the transformed models to Simulink. This model-to-model optimization increases the traceability of the optimizations by staying within the Simulink tool, and allows for further analysis of the transformed model.

Our final result is timing information in each component of the framework, as well as performance results for each optimization. These results were obtained by generating simulation code for each model, before and after transformation. The timings indicated that constant folding does provide a performance boost of around 10 percent, when the Simulink code optimizations are not used.

7.1 Future Work

We have indicated a number of areas where there are potential improvements. It will be an ongoing process to refine and improve these optimizations, as well as create new optimizations. One large component of the difficulty is due to the sheer number of Simulink blocks that can be placed in a model, as each analysis must have a dataflow equation for each block possible. Our suggestion would be for Simulink developers to embed these equations directly within each Simulink block. This would allow us to query each block to determine the dataflow semantics, without essentially re-implementing each block's semantics in our framework.

Another ongoing goal for this project is to collect larger test models, and ensure that our optimizations are working correctly on these. We also plan on implementing some of the optimization transformations as model transformations as in [5]. As mentioned before, this could allow us to formally verify our transformations [9].

Acknowledgements

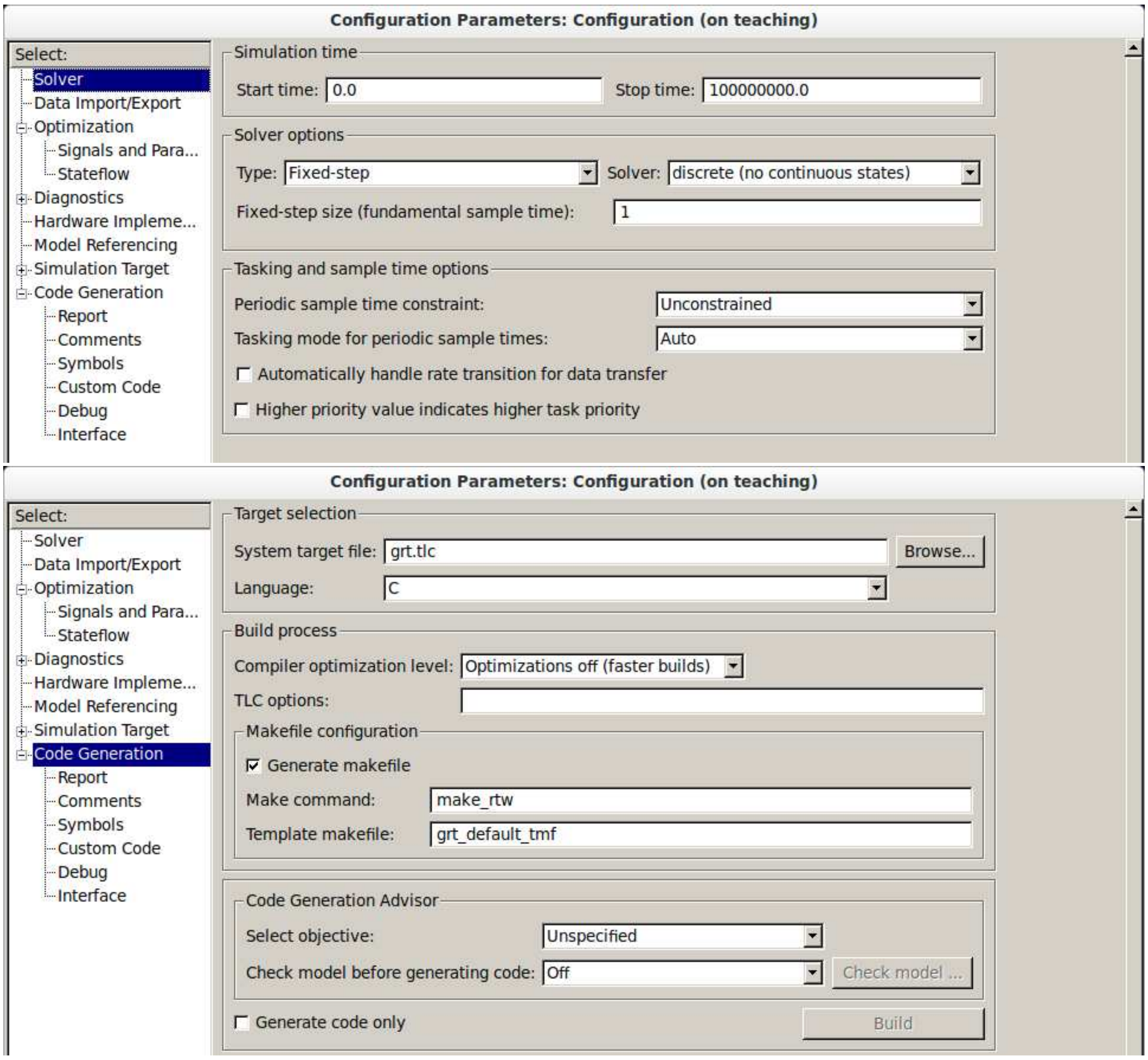
We would like to thank Joachim Denil and Bart Pussig for their code contributions to the project. As well, generous support was provided by Clark Verbrugge, Hans Vangheluwe, Levi Lúcio, Pieter Mosterman, and The Mathworks Inc.

A Appendix

A.1 Project Source Code

The code and documentation for this project can be found at `https://github.com/BentleyJOakes/BDOT`.

A.2 Code Generation Settings



Configuration Parameters: Configuration (on teaching)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Para...
  - Stateflow
- Diagnostics
- Hardware Impleme...
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface

Simulation time

Start time: 0.0 Stop time: 100000000.0

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Fixed-step size (fundamental sample time): 1

Tasking and sample time options

Periodic sample time constraint: Unconstrained

Tasking mode for periodic sample times: Auto

[ ] Automatically handle rate transition for data transfer

[ ] Higher priority value indicates higher task priority

Configuration Parameters: Configuration (on teaching)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Para...
  - Stateflow
- Diagnostics
- Hardware Impleme...
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface

Target selection

System target file: grt.tlc Browse...

Language: C

Build process

Compiler optimization level: Optimizations off (faster builds)

TLC options:

Makefile configuration

[✓] Generate makefile

Make command: make_rtw

Template makefile: grt_default_tmf

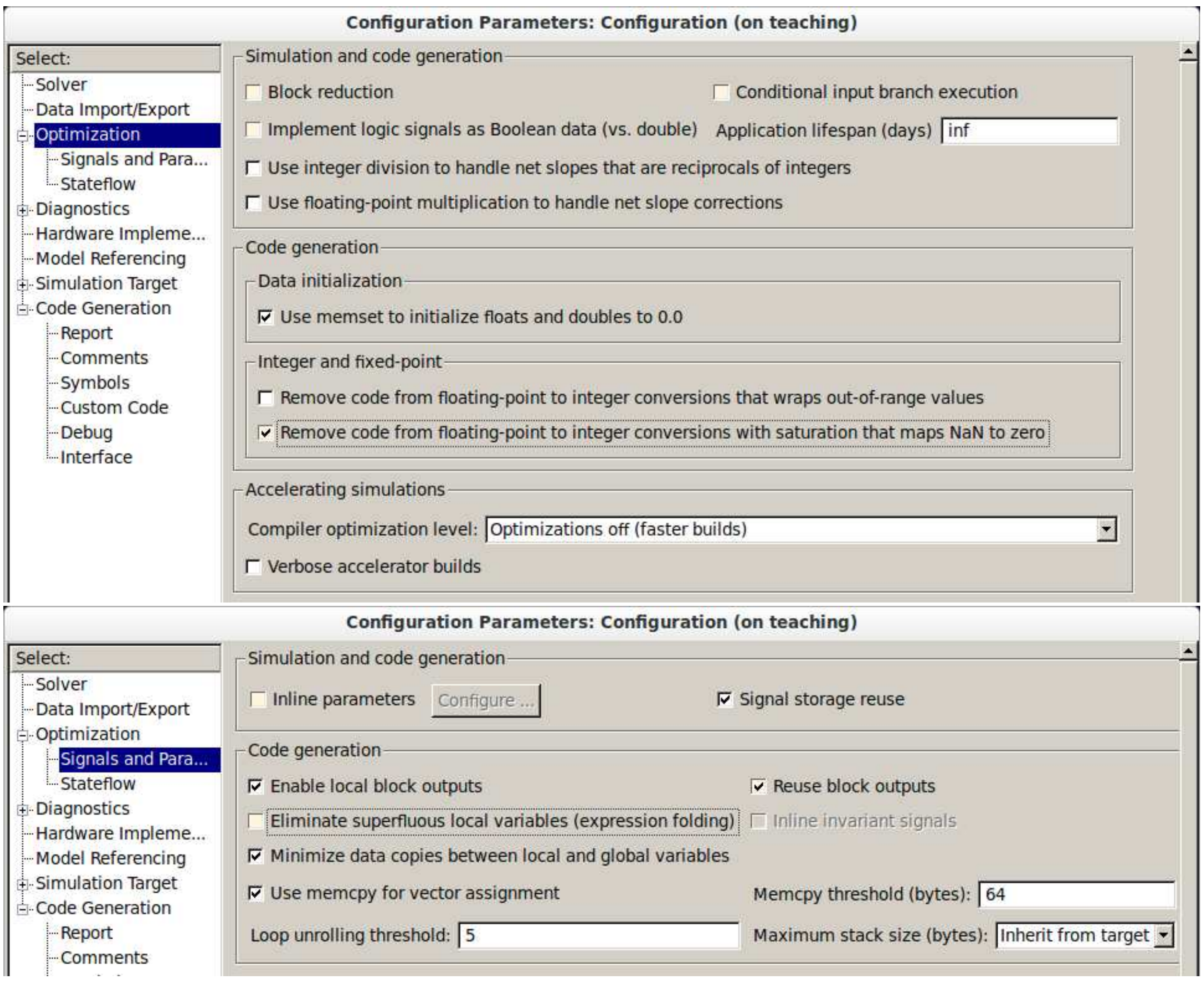
Code Generation Advisor

Select objective: Unspecified

Check model before generating code: Off Check model ...

[ ] Generate code only Build

Figure 18: Settings for code generation



Configuration Parameters: Configuration (on teaching)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Para...
  - Stateflow
- Diagnostics
- Hardware Impleme...
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface

Simulation and code generation
- [ ] Block reduction
- [ ] Conditional input branch execution
- [ ] Implement logic signals as Boolean data (vs. double)
- Application lifespan (days): inf
- [ ] Use integer division to handle net slopes that are reciprocals of integers
- [ ] Use floating-point multiplication to handle net slope corrections

Code generation

Data initialization
- [✓] Use memset to initialize floats and doubles to 0.0

Integer and fixed-point
- [ ] Remove code from floating-point to integer conversions that wraps out-of-range values
- [✓] Remove code from floating-point to integer conversions with saturation that maps NaN to zero

Accelerating simulations
- Compiler optimization level: Optimizations off (faster builds)
- [ ] Verbose accelerator builds

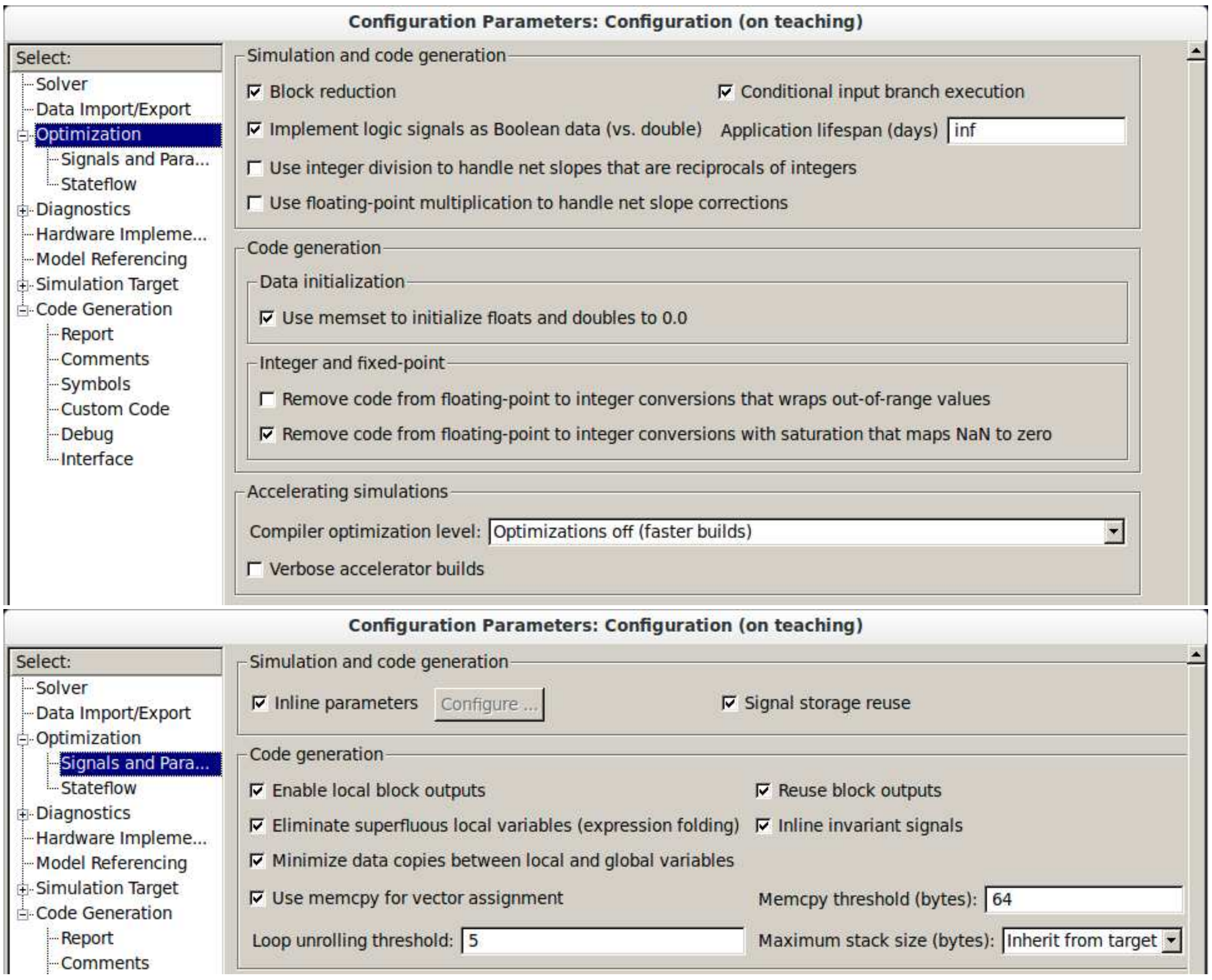
Configuration Parameters: Configuration (on teaching)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Para...
  - Stateflow
- Diagnostics
- Hardware Impleme...
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments

Simulation and code generation
- [ ] Inline parameters Configure ...
- [✓] Signal storage reuse

Code generation
- [✓] Enable local block outputs
- [✓] Reuse block outputs
- [ ] Eliminate superfluous local variables (expression folding)
- [ ] Inline invariant signals
- [✓] Minimize data copies between local and global variables
- [✓] Use memcpy for vector assignment
- Memcpy threshold (bytes): 64
- Loop unrolling threshold: 5
- Maximum stack size (bytes): Inherit from target

Figure 19: Settings for code generation with no optimizations


Configuration Parameters: Configuration (on teaching)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Para...
  - Stateflow
- Diagnostics
- Hardware Impleme...
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface

Simulation and code generation

✓ Block reduction

✓ Conditional input branch execution

✓ Implement logic signals as Boolean data (vs. double)

Application lifespan (days): inf

☐ Use integer division to handle net slopes that are reciprocals of integers

☐ Use floating-point multiplication to handle net slope corrections

Code generation

Data initialization

✓ Use memset to initialize floats and doubles to 0.0

Integer and fixed-point

☐ Remove code from floating-point to integer conversions that wraps out-of-range values

✓ Remove code from floating-point to integer conversions with saturation that maps NaN to zero

Accelerating simulations

Compiler optimization level: Optimizations off (faster builds)

☐ Verbose accelerator builds

Configuration Parameters: Configuration (on teaching)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Para...
  - Stateflow
- Diagnostics
- Hardware Impleme...
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments

Simulation and code generation

✓ Inline parameters Configure ...

✓ Signal storage reuse

Code generation

✓ Enable local block outputs

✓ Reuse block outputs

✓ Eliminate superfluous local variables (expression folding)

✓ Inline invariant signals

✓ Minimize data copies between local and global variables

✓ Use memcpy for vector assignment

Memcpy threshold (bytes): 64

Loop unrolling threshold: 5

Maximum stack size (bytes): Inherit from target

Figure 20: Settings for code generation with optimizations

References

[1] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a model transformation intent catalog. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 3–8. ACM, 2012.

[2] Bruno Barroca, Levi Lúcio, and Clark Verbrugge. Code optimizations using model transformations. Presentation at CAMPaM 2013, 2013.

[3] Joachim Denil, Pieter J. Mosterman, and Hans Vangheluwe. Rule-based model transformation for, and in simulink. In *Theory of Modeling and Simulation 2014 (to appear)*, 2014.

[4] Peter Feher, Tamas Meszaros, Laszlo Lengyel, and Pieter J Mosterman. Data type propagation in simulink models with graph transformation. In *Engineering of Computer Based Systems (ECBS-EERC), 2013 3rd Eastern European Regional Conference on the*, pages 127–137. IEEE, 2013.

[5] Péter Fehér, Tamás Mészáros, Pieter J Mosterman, and László Lengyel. Flattening virtual simulink subsystems with graph transformation. *CoSMoS 2013*, page 39, 2013.

[6] Laurie J. Hendren. Comp 621 - program analysis and transformations. Course notes, January 2014.

[7] The Mathworks Inc. Inverted pendulum with animation. `http://www.mathworks.com/help/simulink/examples/inverted-pendulum-with-animation.html`.

[8] The MathWorks Inc. Simulink. `http://www.mathworks.com/products/simulink/`.

[9] Levi Lúcio, Bentley James Oakes, and Hans Vangheluwe. Symbolic verification of translation model transformations. *Software and Systems Modeling (submitted)*, 2014.

[10] Steven Muchnick. *Advanced Compiler Design and Implementation*, chapter 11, pages 319–328. Morgan Kaufmann, 1997.

[11] Ernesto Posse, Juan De Lara, and Hans Vangheluwe. Processing causal block diagrams with graph-grammars in atom, 2002.

[12] Marc Provost. Himesis. `http://msdl.cs.mcgill.ca/people/mprovost/projects/himesis/index.html`.

[13] Bart Pussig, Joachim Denil, Paul De Meulenaere, and Hans Vangheluwe. Generation of co-simulation compliant fufunction mock-up units from simulink, using explicit computational semantics. In *Theory of Modeling and Simulation 2014 (to appear)*, 2014.

[14] David Thibodeau. A domain specific language for describing dataflow analyses. COMP 621 project, 2014.