Block Diagram Optimization Transformations

Bentley James Oakes, Bart Pussig, Joachim Denil,
Pieter Mossterman, Hans Vangheluwe

February 12, 2014

Abstract

Abstract

Contents

1 Introduction

2 Background

2.1 Causal Block Diagrams

Causal block diagrams are typed graphs, where each vertex has input and output ports. Data, such as scalars and vectors passs along data lines between ports. Each block then performs an operation n the inut dta and outputs it.

2.2 Simulink

Simulink is fun. I love Simulink.

Simulink is a tool created by Mathworks to aid in the development of software. It provides a visual environment to create causal block diagram-like models, which are then generated into C code. This code is then the implementation of the model.

2.3 Compiler Optimization Techniques

Code is compiled to a lower level, such as C-code to machine code, or Java code to Java bytecode.

While performing this translation step, code compilers will also do a number of optimizations to remove redundant code.

We aim to identify optimizations that are relevant to the causal block diagram domain and measure their impact.

2.3.1 Flow Analysis Steps

An important tool for these optimizations is flow analysis, As code can be represented in control flow blocks, information can be thought of to flow between these blocks.

This allows compilers to identify variables and lines of code that can be removed, for example.

We take Laurie's six-step process for flow analysis for code compliation here.

1. Define the approximations to build, as well as a partial ordering for these approximations
   - Such as sets of variable for live variable analysis. The ordering is based on set inclusion.
2. State the problem precisely
   - Usually stated for program point p, and concerning paths to and from the start and end blocks
   - Such as which variables have been used above this line of code.
3. Is the analysis a forwards or backwards analysis

   - Does the analysis start at the beginning block and move forwards, or the end and move backwards through the topological sort

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

2.3.2 Applicability of Flow Analysis to Causal-Block Diagrams

It is important to note the distinctions between the standard code compliation control-flow graph and a causal-block diagram. For example, in a control-flow graph, there is a distinct start block. In a causal-block diagram, any block without a parent is executed in some order. This means that scheduling of the causal-blocks must be taken into consideration.

In Simulink, there is also the concept of subsystems, which are not quite analogous to functions. As well, there is the concept of sampling, which means that there are timing dependencies and considerations when defining optimizations.

2.4 Model Transformations

In order to perform the optimizations on the model itself, we use model transformations. These transformations are composed of a number of rules, which each have a LHS and a RHS. THe LHS matches the elements within with elements in the model. The elements are then replaced with the lements in the RHS of the rule. There are also pivots.

3 Related Work

4 Motivation

Our motivation for this work is to explicitly define optimizations and apply them to models. The advantage of creating optimized models instead of only producing optimized code is that the modeller can then visually see the differences and changes in the model. Even with the traceability links in the Simulink generated code, it can still be difficult to precisely determine which elements in the model a given line of code represents. This model-o-model optimizaion adds extra traceability information where the optimizations are seen in the ssame modeling foramalism.

As well making optimizations to the model removes burden on the target compiler. Optimizations may now be done with more information and structure, and will be applicable to all targets.

5 Categorization

While sourcing optimizations, we began with a preliminary categorization of them.

5.1 Intent

See intents catalog by Levi et al.

Some optimizations may attempt to increase the program's performance in some fashion. However, as the model transformation can change the model for the modeller to then further refine, we realized that there are a number of other transformations to annotate the model or change the user's view of the model.

For instance, we discovered a 'style optimization' to change a model structure from a data-flow orientation design to a control-flow orientated design. This may allow the modeller to model in their own style, potentially reducing errors from unfamiliar structures.

Another 'feedback optimization' is to analyze the model and annotate blocks that perform an errorneous operation such as a divide by 0. This gives the user visual feedback directly on the model as to where error-handling techniques should be performed.

5.2 Layer

Some optimizations may be done in a totally platform-independent way. However, some may depend on the target architecture or language.

5.2.1 Platform-independent

The model optimizations attempt to optimize the model to increase the performance of the model, regardless of the eventual code target. These optimizations include such things as removing useless blocks or reducing expensive operations, and include mostly mathematical operations such as constant propagation and algebraic simplification.

5.2.2 Platform-dependent

The platform-dependent optimizations depend on the target platform.

Some platform characteristics affecting optimization:

Target language Declarative/functional/FPGA can affect how variables are used

Number of target cores Parallel optimizations may be done

Single-/Multi-rate sampling Collection/splitting of blocks running on same sample

Cache strategies Data locality optimization depends on cache size and strategies

Processor heterogenity

Floating-/fixed- point calculations

6 Sourcing Optimizations

In order to source these optimizations, we looked towards two sources to determine available optimizations. However, some optiiizimiations may not be applicable to a causal-block diagram, and of these, some may not be suitable to be done using transformations. For instance, . Also, the list of optimizations available is not complete.

6.1 Code Compilers

Code compilers have a number of analyses and procedures to increase a program's performance. Therefore, we examined a number of sources to gather a list of compiler optimizations. These include the constant folding and algebraic simplification optimizations.

6.2 Simulink Optimizations

Another source of optimizations was found by examining the Simulink documentation. One part was the optimizations done in code generation. For these, an example model and text described the procedure and benefit of the optimization. Another useful list of optimizations came from the Simulink model-checker. This model-checker highlights possible issues with the model and suggests fixes. We identified a number of these checks to be encoded as a transformation and fit within our framework.

7 Optimizations

We will now discuss each optimization in turn. For each, we will discuss where the optimization is applicable, describe the algorithm, and demonstrate example performance benefits.

7.1 Constant Propagation

Constant propagation is the determination of which blocks in the model can be replaced with a constant block. For example, it is possible to statically compute which blocks can be replaced. This optimization then reduces computation, as those blocks do not need to be computed.

7.1.1 Context

When is transformation applicable? This optimization is applicable when there are constant blocks in the model. The performance benefit obtained relies on the calculations performed, but this optimization will improve performance more when there are more constants in the model, and if they are used in close proximity to each other.

7.1.2 Level

This is a platform-independent optimization, as it will make the model smaller without changing the semantics for any target.

7.1.3 Algorithm

Based on flow analysis:

Approximation For each output line, the approximation set will contain a constant value if the output is constant. Otherwise it is empty.

Precise problem Which output lines can be replaced with a constant block

Forwards/Backwards analysis Forwards

Merge operation As output lines cannot be merged, this is not applicable

Flow equations If the

7.1.4 Example Cases (synthetic)

7.1.5 Complexity of Transformation

7.1.6 Example Results

Flattening Constant Propagation Dead-code Elimination Algebraic Simplification Common-subexpression Elimination Code Hoisting Strength Reduction

Scalar Replacement (of array references) For-loop fusion

Procedure Specialization Clone Detection

Instrumentation Removal Instrumentation Addition Vector Indexing Modification (1-to-0-based indexing) Duplicate Name Detection Switch-to-if-else Modification Protection Addition (put warning on div by 0)

Data-locality Annotation

8 Benchmarking