Toward optimized code generation through model-based optimization

A. Charfi, C. Mraidha, S. Gerard, F. Terrier, Pierre Boulet

Toward optimized code generation through model-based optimization

Asma Charfi, Chokri Mraidha, Sébastien Gérard, François Terrier
CEA LIST, Laboratory of model driven engineering for embedded systems Point Courrier 94, F-91191, Gif sur Yvette, France
{asma.charfi, chokri.mraidha, sebastien.gerard, francois.terrier}@cea.fr

Pierre Boulet
LIFL, CNRS/INRIA,
Université de Lille 1, Parc de la Haute Borne, Bât A 40 avenue Halley, 59650
Villeneuve d'Ascq Cedex, France
Pierre.boulet@lifl.fr

***Abstract*—Model-Based Development (MBD) provides an additional level of abstraction, the model, which lets engineers focus on the business aspect of the developed system. MBD permits automatic treatments of these models with dedicated tools like synthesis of system's application by automatic code generation. Real-Time and Embedded Systems (RTES) are often constrained by their environment and/or the resources they own in terms of memory, energy consumption with respect to performance requirements. Hence, an important problem to deal with in RTES development is linked to the optimization of their software part. Although automatic code generation and the use of optimizing compilers bring some answers to application optimization issue, we will show in this paper that optimization results may be enhanced by adding a new level of optimizations in the modeling process. Our arguments are illustrated with examples of the Unified Modeling Language (UML) state machines diagrams which are widely used for control aspect modeling of RTES. The well-known Gnu Compiler Collection (GCC) is used for this study. The paper concludes on a proposal of two step optimization approach that allows reusing as they are, existing compiler optimizations.**

I. INTRODUCTION

Real-Time and Embedded Systems (RTES) have become more complex with an increased number of product features. At the same time, their achievement has to satisfy the demand for shortened development times and higher expectations of product quality. Model-Based Development (MBD) [1] is an approach that aspires to tackle the challenge by taking RTES development into a higher level of abstraction, by using models at the center of the development process. MBD permits automatic treatments of these models with dedicated tools like for instance synthesis of system's application by automatic code generation. Fully automatic code generation offers many advantages to RTES developers, including increased productivity, enhanced source code consistency and intends to improve the performance of the generated system. Among those advantages, performance improvement is the most difficult to achieve. In fact, RTES are often constrained by their environment and/or the resources they own in terms of memory, energy consumption with respect to performance requirements. Hence, an important problem to deal with in RTES development is linked to the optimization of their software part according to the resources provided by their platform. Usually, MBD approach rely mainly on compilers optimization frameworks to perform optimizations automatically. Although the enhancement of code generators and the use of optimizing compilers bring some answers to application optimization issue, most optimized compilers are still unable to perform optimizations related to modeling language semantics. In fact, compilers get all the information about the modeled system from the code level. Thus, the part of the modeling language semantics, which is lost during code generation, is still invisible to the compiler. We will show in this paper that optimization results may be enhanced by adding a new level of optimization at the model level. In this article, we will focus on memory optimization: an optimized code for us is a code that has the smallest size. Our arguments are illustrated with examples of the Unified Modeling Language (UML) state machines diagrams [2] which are widely used for control aspect modeling in RTES. The well-known Gnu Compiler Collection (GCC) [3] is used for this study. The paper is organized as follows. Section 2 gives a general background on the MBD approach. Section 3 presents an evaluation of our experience with GCC optimizations, leading to the need of building another level of optimization. Section 4 compares several alternatives to implement the new optimization level. Section 5 discusses some related works and Section 6 concludes and presents some perspectives.

II. MODEL-BASED DEVELOPMENT FOR RTES DESIGN

Using a MBD approach, designers have to perform three steps: building models, generating code from them and compiling the generated code. An overview about those three steps is given in the following subsections.

A. Raising the level of abstraction in RTES design

Nowadays, a lot of softwares are developed using UML. UML is an OMG standard general purpose language not specific to a domain. Thus, to model RTES, UML need to be extended. For example, the UML profile for MARTE [4] extends the capacities of UML for the modeling and analysis of RTES. MARTE provides facilities to annotate models with information required to perform specific analysis. Thus, models designed using MARTE contain all information about the system and its behavior, making possible an automatic generation of 3rd generation language code.

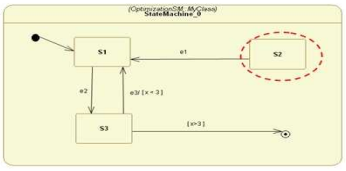
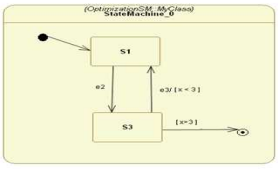
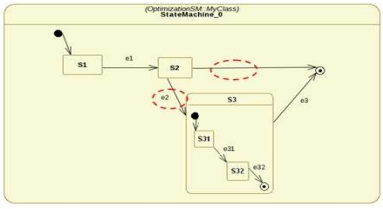
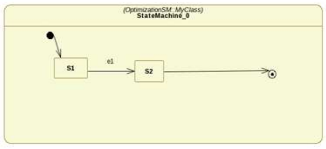
Required Optimization	Non optimized diagram	Size (bytes) of assembly code	Optimized model	Size (bytes) of assembly code	Optimization rate
Remove unreachable state		12669		11393	10.07 %
Remove transition originated from state that have a completion transition		48764		26379	45.90 %

Figure 1. Examples of model optimizations and impact of the optimizations on the assembly code size

B. Generating code from UML models

One of the MBD goals is to automate the design process. For several years, this step was not fully automatic. Some UML tools could generate application source code, but designers have to complete it by hand to get code ready to be compiled. UML tools have evolved and permitted to generate code not only from structural diagrams but also from some behavior diagrams such as state machines diagrams. However, some hand written code was always needed to get the final application. Nowadays, UML specification provides Action and Activities packages that allow full modeling of behavior making code generation from UML models a fully automatable step. In this paper, we are interested in generating C/C++ code since this language is the most used in the RTES development.

C. Compilation and optimization

The most widespread, well known and open source compiler is the GCC. It implements a large number of optimizations which all have a different impact on code quality, compilation time, code size, etc. For this reason, GCC provide a limited number of optimization levels: -O1 (default level), -O2 (decrease execution time), -O3 (-O2 optimizations plus those that may increase code size) and -Os (reduce code size). Since we deal with RTES design and especially with code size concerns, we are interested in -Os flag (s listed for size). GCC optimizations are also classified according to their level of abstraction. There are low level optimizations (closer to the target) and high level optimizations (closer to the source code). Before GCC 4.0, all optimizations operate in a low level of abstraction called RTL (Register Transfer Level) [5] . Being a low-level representation, RTL works well for optimizations that are close to the target (e.g., register allocation, peepholes optimizations, etc). However, many optimizations need higher level information about the program that is difficult to obtain from RTL (e.g., array references, data types). Moreover, too many target features, such as function calling convention are explicit in RTL, making difficult to implement optimizations that are not interested in target details. To overcome RTL drawbacks, GCC committee introduces since GCC 4.0, a new intermediate form based on tree presentation to allow the implementation of high level. This new representation [5] is called SSA because it is based on the Static Single Assignment from [6] which requires that program variables are assigned in exactly one location in the program. So, the compiler looks for only the last definition of a variable when it will be used. SSA was benefic to improve optimizations because most of the discovered optimization algorithms are mathematical ones that need to be executed on a higher abstract level than the RTL. Although GCC offers a lot of optimization passes (more than 100), RTES designers still not satisfied. In the next section, we will present some optimizations not provided by GCC.

III. EXPERIMENTAL STUDIES

In order to illustrate the fact that we can not rely only on compiler optimizations, we introduce an example of state machine diagram with unreachable state. Obviously, the code related to this state will not be executed and can be considered as a dead code. One of the GCC optimizations is called dead code elimination. This optimization should be able to remove all dead code. The following section shows if the GCC is able to detect an unreachable state as a dead model part.

A. Building state machine diagram

UML state machine diagrams are used to specify the dynamic behavior of active objects. Behavior is modeled as a traverse of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of event occurrences [2]. Our diagram (top left of fig. 1), designed with Papyrus [7], contains 3 states, 2 pseudo states (initial and final states) and 5 transitions. We can notice that S2 is an unreachable state because it has no incoming transitions.

B. Generating the C++ code from UML state machine

There are number of patterns that may be used to implement a UML state machine. Most popular ones are: the State Pattern [8] where each state is implemented as a whole class, the State Table Transition (STT) [9] which consists in building a 2 dimensions table describing the relation between states and events, and the Nested Switch Case statements [10] which is the most commonly used pattern. The latter pattern consists in having an outer case statement that selects the current state and an inner case statement that selects the appropriate behavior given the type of the received event. Moreover, UML contains semantic variation points that define an intentional degree of freedom for the interpretation of the

model semantics [11]. For state machine diagrams, semantic variation points mainly concern the events and the transitions selection policy. Before generating code from our diagram, we have fixed the execution semantic as well as the implementation pattern which is the Nested Switch Case. All measures presented in the rest of this section are related to code generated using the Nested Switch Cases pattern.

C. Compiling and optimizing the generated code using GCC

To compile the generated code, we used the -Os flag of the GCC 4.3.2. GCC provides options to view intermediate forms generated from code level until binary code. For each optimization pass, GCC generates the corresponding file. In the dead code elimination file, we have found that code related to the unreachable state still exists, which means that GCC did not remove the dead code. The size of the generated assembly code is *12669* bytes. Now, let's optimize the model by removing the unreachable state. Our optimization tool (implemented in java) gives the user the ability to choose the optimization that he would perform. It generates then the optimized model after running the selected optimization. From the new optimized model, we generate C++ code and recompile it. We obtain an assembly code measuring only *11393* bytes (). It is true that the gain in term of assembly code size is not significant (only *10.07* **%**), but this gain is proportional to the number of removed states/transitions. It depends also on the kind of state machine. The example presented in the second line of Fig. 1 deals with hierarchic state machine that contains a composite state S3. There are two outgoing transitions from State S2. To move from S2 to S3, event e2 is needed, however we do not need a particular event to move from S2 to final state. This particular transition is called a *completion* transition. According to the UML semantic, the completion transition is first fired whatever the received event is. It means that our composite state S3 is never active. Thus, after optimizing this model, we won more than *45 %* of the assembly code size (Fig. 1). In fact, in our state machine implementation, each composite state has a reference to a C++ class that implements the submachine. When we optimize the model, the whole class is removed. It should be noted that the gain can depend also on the implementation pattern. We have generated code from the hierarchic state machine presented in the Fig. 1, using State Pattern and STT pattern to prove that we always have gain in terms of code size independently of the used pattern. Table1 shows that the implementation of the same state machine using the STT is less compact than the other patterns. It shows also that whatever the pattern is, we obtain a significant gain when dealing with hierarchical state machine.

D. Results interpretation

We said before that the RTL representation contains some parasite information that prevents from building the control flow graph which is essential to run a lot of SSA optimizations. We said also that some information that can be helpful to optimize code were lost when we move to the RTL form such as type notion, data structure, etc. Unfortunately, SSA form, although it introduces new optimizations to the GCC compiler, suffers from the same RTL drawbacks listed above. Even SSA form does not contain all information about the system. In fact, GCC gets all information about the system from the generated code. However, some analyzes and transformations need higher level information about the application that is not possible to obtain from SSA level. These pieces of information exist in models but are lost by code generation. For example, the information that said "a state with no incoming transition is an unreachable state, so its code is a dead code" is lost when we move from model to code. Thus, GCC can not remove this kind of dead code. Therefore, we have to exploit UML models semantics information before their lost.

TABLE I. OPTIMIZATION GAIN FOR THREE DIFFERENT PATTERNS

Implementation Pattern	Size (bytes) assembly code of non optimized model	Size (bytes) assembly code of optimized model	Optimization rate
STT	13885	9607	30.81%
Nested Switch	48764	26379	45.90%
State Pattern	49863	23663	52.54%

IV. EXPLOITING UML SEMANTICS FOR OPTIMIZATION

There are three alternatives to implement optimizations related to the UML semantics: implement them before the code generation, during code generation and after code generation.

A. Before code generation

By analogy to changing levels of optimization in GCC (switch from a low level RTL to a higher level SSA) to implement new higher level optimizations, we consider another level of optimization which is higher than SSA. Our level is the UML model. Implementing those optimizations at the model level allows us to exploit the useful model abstraction level for optimizations issues. This alternative is more benefic than the alternative that consists in implementing those optimizations in the compiler process. In fact, given the C++ representation of the code to be compiled, GCC has to build the control flow graph of this sequential form to perform SSA optimizations. However, in the model level we have already the control flow graph expressed by the state machine diagram.

B. During code generation

One of the interesting code generator features, other than the fully automatic code generation, is model debugging. Some code generators such as [12] provide a model debugger that permits to debug models using breakpoints. However, if we decide to implement optimizations related to UML semantics in the code generator, model debugging will not be an easy task (we may put breakpoints on elements that will be removed by optimization). Thus, optimizing during the code generation is likely to widen the gap between the model and the code.

C. After code generation

This alternative consists in extending GCC optimizations to make UML semantics visible to the compiler. We can add new optimizations passes using the GCC plug-in architecture. However, GCC has no stable API that would allow such addition. Moreover, UML state machines have a number of semantics variation points that may potentially lead to infinity of possible interpretations that would be hardly implementable in lower levels of GCC. A classification of the alternatives (Table 2) shows that implementing optimizations before code generation is the only alternative that is independent from the implementation. This is an expected result since the implementation occurs after modeling. The model debugging is only affected by implementing optimizations during the code

generation. Otherwise, it can not be affected before code generation since debugging takes place after code generation nor in the compiling process since models are not visible to compilers. UML semantics variation points are fixed from the beginning, in the model, so all alternatives depend on this chosen semantic. If we change the semantics, all optimizations implementations have to be changed. Since the model is more compact and did not contain parasite sequential code found in the code, implementing the optimizations before code generation or during code generation is easier than implementing them in the compiling process. We can conclude from the Table 2 that implementing optimizations related to UML semantics information at the model level is the best way to exploit UML semantics information before their lost.

V. Related Works

There are other approaches that are not satisfied by compiler optimizations level such as [13] that decided to not only extend the C++ language (by Concepts) but also aims to extend the compiler to understand this extension. This approach differs from ours in term that it extends both the language and the compiler and it does not trust the compiler. However, in our approach we trust the compiler and we aim to reuse its powerful optimizations. We did modify neither the language nor the compiler; we simply add another level of optimization. In [14], the author presents cases where source language knowledge is important for large gains in FORTRAN optimizations. Related works presented above describe, like our approach, the necessity of exploiting source language semantics in the compilation process. However they are not interested in generating optimized code from UML models. The xtUML [15], a subset of UML with defined execution semantics, offers the ability to translate UML model directly into 100% complete and optimized code using a model compiler. Ref. [16] and [12] are examples of model compilers. Model compilers are likely to become complex and hard to maintain systems since all optimizations are implemented in those systems. xtUML allows the use of only a subset of UML. This can be considered as an advantage because it facilitates model compiling (only a well founded subset of UML is used). However, this can prevent designers from modeling their systems using all UML concepts. In contrast to this approach, our approach will give designers all freedom to use all UML concepts. A model optimization is a kind of model refactoring [17]. It is a model transformation that guarantees the transition from non optimized model to an optimized one by keeping unchanged its behavior. Our optimization tool is a whole framework based on model transformation that allows, just like [18] and [19], model refactoring.

VI. Conclusion and Perspectives

In this article, we discussed the optimization issues in a MBD for RTES. Code optimizations in MBD are almost done in the compiling process. Most of compilers offer a lot of optimizations passes, but they are still unable to perform some optimizations. We have presented examples of optimizations that GCC can not perform. This is due to the loss of some UML semantics information during the code generation. We have studied different alternatives to implement optimizations related to lost UML semantics. We concluded that optimizing in the model level is the most advantageous alternative since it is independent from the model implementation, do not affect model debugging and is easier to maintain and evolve. We proposed then, a two step optimization approach where optimizations are performed both in the model and compiler levels. In the current version of our optimization tool, the users choose manually the optimizations to perform. We plan to improve our tool in a way that it automatically executes optimizations that correspond to the UML model. Given the benefits of the model compilation approach presented in section 5, we aim to compile directly our models to binary code. This approach brings the advantages to avoid the unnecessary use of two higher level languages: Executable UML and a 3rd generation language. Since there is no relationship between models and sequences of 0s and 1s, we have to transform models into intermediate forms that allow us to optimize more until reaching binary code. To do that, we are investigating to build a UML GCC front-end.

TABLE II. Classification of the three alternatives

	Easy to implement	Easy to detect	Affect model debug	Independent from model implementation	Independent from semantics
After code generation	NO	NO	NO	NO	NO
During generation	YES	YES	YES	NO	NO
Before code generation	YES	YES	NO	YES	NO

Acknowledgment

This work has been partially funded by the *Agence Nationale de la Recherche* in the context of the RT-Simex project.

References

[1] B. Schätz, A. Pretschner, F. Huber, J. Philipps, Model based development of embedded systems, Lecture Notes in Computer Science, vol 2426, 2002, Springer, 2002, pp.331-336.

[2] OMG, OMG Unified Modeling Language, Superstructure, 2008.

[3] GCC, the GNU Compiler Collection, http://gcc.gnu.org.

[4] http://www.omgmarte.org/

[5] D. Novillo, Tree SSA A New Optimization Infrastructure for GCC, GCC Developers Summit, Ottawa, Ontario Canada, 2003.

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM TOPLAS, vol 3, pp:451–490, October1991.

[7] http://www.papyrusuml.org/

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[9] R.C. Martin, UML Tutorial: Finite State Machines, C++ Report, 1998.

[10] P. Metz, J. O'Brien and W. Weber, Code Generation Concepts for Statecharts Diagram of UML v1.1, Object Technology Group, 1999.

[11] F. Chauvel and J-M. Jézéquel, Code generation from UML Models with semantic variation points, Proceedings of UML MoDELs, 2005.

[12] http://www.mentor.com/products/sm/model_development/bridgepoint/

[13] P. Gottschling, A. Lumsdaine, Integrating Semantics and Compilation Using C++ concepts to develop robust and efficient reusable libraries, GPCE, 2008.

[14] T. Moene, Front End Based Optimization: Premature or Inevitable?, GCC Developers' Summit, Ottawa, Ontario Canada, 2007.

[15] S.J. Mellor, M.J. Balcer, Executable UML: A Foundation for Model Driven Architecture, Addison Wesley, ISBN 0-201-74804-5, 2002.

[16] http://pathfindermda.com/

[17] Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.

[18] M. Lawley and J. Steel, Practical Declarative Model Transformation With Tefkat, Workshop Model Transformation in Practice, 2005.

[19] A. Boronat. MOMENT: A Formal Framework for Model Management. PhD thesis, Universtat Politècnica de València, 2007.