GeCoS: A Framework for Prototyping Custom Hardware Design Flows

Antoine Floc’h†, Tomofumi Yuki†, Ali El-Moussawi‡, Antoine Morvan†, Kevin Martin†, Maxime Naullet‡, Mythri Alle‡, Ludovic L’Hours, Nicolas Simon‡, Steven Derrien‡, François Charot‡, Christophe Wolinski‡ and Olivier Sentieys‡
† Tocea
antoine.floch@tocea.com
‡ Irisa (Université de Rennes 1, EISIR)
firstname.lastname@irisa.fr
§ Université de Bretagne Sud
kevin.martin@univ-ubs.fr

Abstract—GeCoS is an open source framework that aims to be a highly productive environment for prototyping hardware design flows. GeCoS primarily targets custom hardware design using High Level Synthesis, distinguishing itself from classical compiler infrastructures.

Compiling for custom hardware makes use of many domain specific semantics that are not considered by general purpose compilers. Finding the right balance between various performance criteria, such as area, speed, and accuracy, is the goal, contrary to the typical goal in high performance context to maximize speed.

The GeCoS infrastructure facilitates prototyping of hardware design flows, which go beyond compiler analyses and transformations. Hardware designers must interact with the compiler for design space exploration, and it is important to be able to give instant feedback to the users.

I. INTRODUCTION

In a typical flow of compiler research, new compiler passes are implemented as a “proof-of-concept” and to enable experimental validation. Compiler infrastructures that allow for rapid prototyping of compiler passes can greatly accelerate compiler research and are therefore an important aspect of compiler research.

Several compiler infrastructures have already been developed for this purpose, including PIPS [1], SUIF [2], ROSE [3], Polyglot [4], and CETUS [5]. These compiler frameworks provide the groundwork for implementing custom analyses and transformations with a primary focus on source-to-source transformations. They mainly target high performance computing platforms (multi-core, GPGPUs, distributed memory clusters, etc.) and focus on C/C++ based programs. In such tools, the input program is transformed into a semantically equivalent version, supposed with better performance.

However, compiler research is not limited to general purpose or high performance computing. Designing and programming ultra low power high performance embedded hardware platforms also requires state-of-the-art compiler technology. As such platforms can be found virtually everywhere in daily life, e.g., cars, Smartphones, TVs, gaming devices, and so on, there is a growing need for compilers tailored to embedded hardware requirements.

One example of such domain specific compilers are High Level Synthesis tools, also known as C to hardware compilers [6], [7]. Such tools enable the derivation of application specific custom hardware from high-level algorithmic specifications in C or Matlab. Many typical application domains benefit from HLS tools, as most embedded hardware platforms integrate custom hardware accelerators for compute intensive tasks.

The input to HLS tools has significant impact on the synthesized hardware, just like how source level transformations influence general purpose compiler outputs. However, many of these influences are specific to HLS as it depends on how well the input maps to hardware. There is hence a recent trend toward exploring architectural variants through the use of HLS specific source-level transformations [8], [9], [10], [11].

General purpose and custom hardware compilers are used in two very different contexts. In a typical compiler, users expect the output to be optimized for speed. Hardware designers map performance critical parts of embedded systems, described in C/C++, on custom hardware accelerators. The design process consists in exploring the trade-offs between Quality of Results (QoR) and cost (area), while fulfilling some performance constraint.

A typical example is the choice of fixed-point over floating-point arithmetic, which alters the numerical accuracy of the implemented accelerator, but helps reduce its cost and/or improve its performance. The designer must choose a balance between accuracy and cost, but the desired balance is highly context sensitive.

This design space exploration is an iterative and interactive process done by designers. Tools assisting this exploration have two specific needs:

- The intermediate representation must be flexible and extensible enough to take advantage of domain specific semantics, such as bit accurate or fixed point datatypes.
- The tool must support interactions with the programmer, guiding design space exploration. Thus, it is essential for the users to be able to drive the compilation flow, and to receive feedback from the tool.
Standard compiler infrastructures are not necessarily well suited to address such problems. Their Intermediate Representations (IR) often lack the flexibility or extensibility required. For instance, we are not aware of an existing compiler IR with native support for bit-accurate or fixed-point datatypes. Moreover, existing compiler infrastructures lack the interactivity desired in a hardware design flow.

To address these issues, we designed a compiler infrastructure named GeCoS (Generic Compiler Suite) targeting custom hardware synthesis and embedded hardware in general. GeCoS aims to facilitate fast prototyping of complex compiler optimizations and analyses, where these implemented passes can be used interactively by hardware designers to explore trade-offs between hardware performance criteria, such as speed, area, and accuracy.

The GeCoS compiler focuses on extensibility and modularity, along with powerful IR Query facilities (including an advanced pattern matching engine). These features are brought together by using Model Driven Engineering techniques and provided the Eclipse Modeling Framework [12]. Interactions with users are supported through a script-driven interface to customize compilation flows, and through immediate and detailed user feedback in the source editor.

The rest of this paper is organized as follows. Section II describes the architecture of GeCoS and highlights its key features. Several tools for manipulating the intermediate representations of GeCoS are presented in Section III. Section IV illustrates how the IR of GeCoS can be extend along with its front-end and back-end. Section V describes the interactive aspect of GeCoS that allows for user-compiler interactions. Related work is discussed in Section VI and we conclude in Section VII.

II. AN OVERVIEW OF THE GECoS INFRASTRUCTURE

In this section, we present an overview of the GeCoS infrastructure. GeCoS is not simply a compiler, but rather an environment for developing custom hardware design flows. We achieve tight integration of the compiler into the development flow by taking advantage of the Eclipse IDE.

Figure 1 presents a high level overview of GeCoS and its related toolings. The overall flow is similar to a typical compiler: the input is parsed through the front-end, processed via several analyses and transformations, and then output is generated from the modified IR. In addition to various benefits from integrating GeCoS into the Eclipse IDE, its unique features are a script-driven compilation flow and compiler IRs modeled using the Eclipse Modeling Framework. Although it is not shown in Figure 1, Static Code Analysis Framework (Codan) plugins in Eclipse enable us to provide instant feedback to the editor.

A. Eclipse Integration

We made the choice to integrate our infrastructure into Eclipse to take advantage of its application framework. Hardware design flow benefits from an interactive and graphical user interface, and it was not an option to develop a full development environment from scratch.

Eclipse has many extension points, or features (e.g., editors), that can be customized through plug-ins extensions. We use these extension points to implement custom editors, to take advantage of content assist, provide instant feedback, and so on. The interactive features of GeCoS based on these tools are described in Section V.

The GeCoS front-end supports a superset of C99 including bit-accurate datatypes as defined by the SystemC standard (which are indeed C++ templates). We use the C/C++ Development Tooling (CDT) of Eclipse to parse C/C++ inputs. The CDT parsed tree is then converted to the GeCoS IR.

B. Compiler IR as Metamodels

One of our goals during the design of GeCoS was to reduce development and maintenance effort as much as possible. Since GeCoS was expected to be developed by generations of students, being able to reuse prior work was strongly desired. Furthermore, not all developers of GeCoS were expected to have a good background in software engineering, as our research spans in between computer science and electronic engineering.

Motivated by these goals, we adopted Model Driven Engineering techniques using the Eclipse Modeling Framework. MDE is a software engineering methodology where the development effort revolves around models; abstractions of domain knowledge. MDE is commonly accompanied by generative approaches, e.g., automatic generation of skeleton codes from models. Generative approaches contribute to standardized code structure, making it easier for the developers to use each others...
code. Developers with little software engineering background are also forced to follow certain basic principles, such as factory/visitor design patterns.

By using models to specify the grammar of compiler IRs, which is called metamodeling in modeling terms, we gain access to many benefits from MDE, directly applicable to compiler development [13]:

- Providing structural consistency: The structure of the IR is defined in the metamodel, or the grammar of the IR. Thus, tools to check if the data structure of a model instance (IR of a input source) matches its metamodel (grammar of IR) can be directly used for validating IR transformations. Such structural validation can catch bugs in transformations that deeply impact the IR.
- Access to modeling tools: There are many tools for analyzing and manipulating models. These tools include parser generators and code generators to/from models that are useful for compiler construction. Other tools include basic tree-based editors of IRs, serialization of IRs to XMI, and so on.

III. QUERYING AND MANIPULATING GeCoS IR

Analyzing and manipulating Intermediate Representations is an important part of any compiler analysis or transformation. In this section we describe a set of tools that we have implemented for manipulating GeCoS IRs.

The base IR of GeCoS is an enriched Abstract Syntax Tree (AST) that represents C programs constructs. There are many forms and auxiliary informations associated with the IR, such as control/data flow graphs, static single assignment (SSA) form, and DAG representation of basic blocks.

Many analysis and transformation passes require IR querying and traversal operations to identify patterns ranging in complexity. With the help of MDE, GeCoS provides several tools for reducing the effort required to query its IRs. In this section, we describe two of the tools we have in GeCoS, (i) Tom/Gom term re-writing system, and (ii) graph adapter as a common interface to graph operations.

A. Tree pattern matching and term rewriting with Tom/Gom

While many languages natively support pattern matching (e.g., Scala, Caml, Haskell), the Tom/Gom tool [13] provides much more powerful constructs along with a tight integration into the Java language.

These rules are much easier to express and to understand than visitor based implementations. However, Tom/Gom requires bindings from expressions in its language to Java objects. We have developed a tool where these mappings can be specified in a Domain Specific Language (DSL). From a DSL specification of how elements in a metamodel (IR grammar) map to Tom/Gom terms, the necessary bindings are automatically generated. Our approach is very similar to that of Bach et al. [15] except that it offers more flexibility and customization opportunities. Figure 2 shows a small Tom/Gom use case.

Fig. 2: Examples of expression simplification rules defined using Tom/Gom. Tom/Gom terms, such as affine and intTerm, correspond to Java class instances, where the mapping is provided in a separate file. Using these rules, users can define complex re-writing rules. The first rule defines a factorization of affine expressions of the form $ac + bc$ to $(a + b)c$. The second rule defines a simplification rule that removes integer terms that evaluate to 0.

B. Graph Adapter

Graphs are key data structures for compiler analysis, used to capture many different kinds of information. Examples of graphs used in compilers include call graphs, control flow graphs, dataflow dependence graphs, program dependence graphs, and so on.

While the abstract notion of graphs is common across the board, its concrete implementations can take different data structures. For example, a graph edge may be explicitly represented by an object instance (this is the case for dependence graphs, where edges can be of different types) or simply by a list of vertex-to-vertex cross references. In addition, many graph algorithms can benefit from external optimized library implementations. While algorithmic reuse is highly desirable in such a context, it raises many difficulties.

We use the adapter design pattern, illustrated in Figure 3 to alleviate reuse of graph algorithms. However, when using adapter design patterns for many different graphs, defining the adapter also requires considerable effort.

Fig. 3: Illustration of the Graph Adapter. Any graph structure can be first adapted to a common abstract representation of graphs. The adapted graph can now make use of available graph operations, including an interface to the JGraph library. All the efforts in analyzing and manipulating graphs can be concentrated on the adapted graph, and can be reused by many different graphs found in a compiler.
To address this, we have implemented a toolset to automatically generate adapter patterns enabling the reuse of a large set of graph analysis and traversal algorithms (from breadth-first search to subgraph isomorphism) over many different graph representations. Our approach leverages the structural information provided by the compiler IR metamodel to derive the adapter code. We do so by using an abstract specification (specified in a textual DSL) of a set of IR Classes defining a graph. Our approach builds on the idea of Model Typing [16] developed by the Model Driven Software Engineering community.

IV. Extensible IR

In this section, we illustrate the extensibility of the GeCoS IR through two examples. The first example demonstrates domain specific code generation by using domain specific representations for regions of the program. We use polyhedral code generation as an example to extend the IR with domain specific information, and to take advantage of the IR for code generation.

The second example presents an extension to the front-end to handle a subset of Matlab programs. The base IR of GeCoS can represent most but not all features of Matlab. We hence extended the IR to support new types and operations (e.g., aggregate computations operating on non-scalar data structures).

A. Extensibility by Adapters

The common mechanism used resembles the adapter software design pattern. The GeCoS base IR has several abstract nodes, such as Block, Instruction, and Type. The behavior of analyses and transformations may be different for each instance of these abstract classes. When an analysis or a transformation traverses the AST, it may delegate the handling of specific instances to its handler. In GeCoS delegation usually occurs when the object instance class is not known to the transformation/analysis (i.e., when the class is not part of the base GeCoS IR). Similar design is used in Polyglot [4] for providing a framework to implement Java dialects.

We use an adapter-like mechanism combined with the plug-in framework of Eclipse to achieve a modular and extensible implementation of analyses and transformations. The key in the modularity is that an analysis or transformation does not need to be aware of all possible implementations of an abstract node. Corresponding handlers will be loaded as necessary by Eclipse at run-time, separating domain specific behaviors into its own modules (Eclipse plug-ins).

B. Polyhedral Code Generation

The polyhedral model [17] is a formalism enabling the analysis and transformation of complex loop structures. The model is restricted to a class of loop with affine bounds and array indexing functions, known as Static Control Parts (SCoPs). For this class of loops, the model provides a unified framework for capturing, checking and applying complex combinations of loop and array layout transformations. Polyhedral techniques have proven to be very effective for automatic parallelization [18] on multi-cores, GPGPUs and computing clusters, and are also very relevant for embedded hardware design.

To enable polyhedral transformations within GeCoS, we have extended the IR to represent polyhedral regions (SCoPs) in a program. This was done by introducing an inheritor of Block, namely SCoPBlock, to model SCoPs. This block models loop nests in polyhedral representation, and goes through different sets of analyses and transformations. Instances of SCoPBlock are extracted from the initial program using a combination of complex pattern matching rules and normalization transformations both heavily relying on the Tom/Gom framework (the rewriting rules shown in Figure 2 are one such example).

Although it is possible to convert SCoPs blocks back to the base IR so as to benefit from the existing code generator, doing so prevents us from leveraging the domain specific semantics of SCoP blocks during code generation. For example, it is possible to generate very efficient HDL code from most SCoP based representation, whereas this becomes much more challenging to do from a standard control-flow graph representation. To address this issue, we rely on the extensibility of GeCoS to provide custom code generator extensions which handle code generation stages for all instances of SCoPBlock.

C. Matlab Front-end

A more involved extension example is the extension of the IR to support a new language front-end, in our case a subset of Scilab/Matlab. Thanks to the use of an EMF-based IR, we were able to benefit from advanced textual DSL design frameworks such as Xtext [19]. This allowed us to build an IDE and parser front-end for the Matlab language from a single specification, where the parsing result is directly in the form of an extended GeCoS IR.

However, the real challenge stems from the fact that, in addition to being dynamically typed, these languages offer built-in vector and matrix types along with many corresponding operators. Deriving custom hardware from such programs first requires that the compiler is able to infer (at least semi-automatically) the type of each program statement statically. Then arrays and vector operations need to be lowered into simpler loop based constructs that can be supported by a high-level-synthesis back-end.

Extending the IR for such languages hence involves not only adding new extensions to the Block and Instruction classes, but also extending the type system by introducing new Type classes to model vector/matrix types and partial/incomplete typing information which is to be resolved during type inference.

Similarly, we also designed the type inference engine by reusing and extending the initial GeCoS type propagation pass, by delegating the handling of extended types to a Matlab specific type propagation pass.
V. Interactive Development in GeCoS

In this section, we illustrate the interactive aspect of GeCoS. The two key features for user interaction are (i) script-driven compilation flow, and (ii) static analysis feedback while programming.

A. Script-Driven Compilation Flow

Users of GeCoS “compile” a program by providing a script specifying the sequence of compiler passes to invoke. Figure 4 illustrates an example of a script that runs a sequence of passes in GeCoS. Although the default options are used in the figure, many of the commands in the script have multiple optional parameters to fine tune its corresponding compiler pass.

Using the Eclipse framework brings additional features that are crucial to increase productivity of both users and developers of compiler passes. For example, we have taken advantage of the content assist module of Eclipse so that users can access the available list of commands, just like searching for Java methods in Eclipse. As another example, users may use the “Open Definition” feature of Eclipse to locate the implementation of a command in the script, which is a Java method.

B. Immediate Feedback to Users

Those who have used Eclipse or any other IDE for programming have experienced how quickly an IDE can flag simple syntactic errors to the user. These tools constantly check for compilation errors/warnings and reflect them in the editor. Even purely syntactic checking can help improve productivity by avoiding going back and forth between the code and compiler outputs.

Optimizing compilers perform a significant amount of analysis to select legal transformations. For instance, automatic parallelization requires sophisticated dependence analysis to find sets of parallel operations. In the interactive compilation flow, it is desirable to reflect such analysis results to the user, in a manner similar to syntactic checks.

The Eclipse Static Code Analysis Framework (Codan) provides an API for providing such feedback. Figure 5 illustrates examples of how we use the Codan framework to give instant feedback to the user.

VI. Related Work

In this section, we discuss compiler infrastructures in relation to GeCoS. There are many compiler infrastructures with different goals, but we are not aware of other frameworks that target prototyping of hardware design flows.

There are several compiler infrastructures already available, such as PIPS [1], SUIF [2], ROSE [3], and CETUS [5]. These infrastructures aim at offering data structure and APIs that can scale to millions of lines of code, often at the expense of ease of use. To the contrary, hardware synthesis focuses on computational hotspots that rarely consist of more than a few thousands of lines of code, yet it commonly takes minutes if not hours to synthesize.
Moreover, they are intended to be robust compilers, primarily focusing on general purpose processors. Thus, they have little or no support for embedded platforms, or user interaction.

CoSy [20] is a compiler framework for slightly different goals. They have developed a production quality compiler for embedded processors with a high degree of flexibility. They propose to utilize the flexibility to create customized versions tailored for specific architectures. However, they target embedded processors, and not custom hardware.

Polyglot [4] is a compiler framework that focuses on supporting domain specific dialects of Java. They also provide an adapter-like framework to allow users to develop extensions with little effort. HLS tools typically take C/C++ as inputs and not Java, and thus Polyglot is not suitable for compilers targeting hardware design.

CHiLL [21], POET [22], and AlphaZ [23] are program transformation frameworks that allow users to specify transformation sequences via script. However, all of these frameworks target high performance computing (multi-cores, and GPGPUs), and lack support for hardware-oriented compilation.

VII. CONCLUSION

We have presented GeCoS, a compiler infrastructure targeting custom hardware design. Compiler research for hardware design has very different goals and requirements, making existing infrastructures difficult to use. In particular, the need to model domain specific semantics, and to support interactions with the user are much stronger in our context.

The tool is open source and a ready to use Eclipse installation is also available on our website 1. Readers can also watch a 5 minute YouTube video 2 to see GeCoS in action.

Our paper may also be viewed as an advertisement of the Eclipse framework and Model Driven Engineering for compiler development. GeCoS benefits a lot from tools and frameworks coming from Eclipse. Taking advantage of Eclipse can provide an ideal environment for implementing domain specific languages, which even comes with an IDE, with little effort.

ACKNOWLEDGEMENTS

This work was funded in part by the INRIA STMicroelectronics Nano2012-S2S4HLS project, the French ANR Compa project (ANR-11-INSE-0012), and the European Commission Seventh Framework Programme (FP7/2007-2013) with Grant Agreement 287733.

REFERENCES