deGoal a Tool to Embed Dynamic Code Generators into Applications

Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo, and Rémy Gauguey

CEA, LIST, Département Architecture Conception Logiciels Embarqués,
F-38054 Grenoble, France
firstname.lastname@cea.fr

Abstract. The processing applications that are now being used in mobile and embedded platforms require at the same time a fair amount of processing power and a high level of flexibility, due to the nature of the data to process. In this context we propose a lightweight code generation technique that is able to perform data dependent optimizations at run-time for processing kernels.

In this paper we present the motivations and how to use deGoal: a tool designed to build fast and portable binary code generators called *compilettes*.

1 Introduction

Today, software development is facing two competing objectives:

- Improve programmers efficiency by using generic and expressive programming languages
- Generate efficient machine code to achieve the best execution speed and energy efficiency

These two objectives are competing because the more expressive and abstract a programming language is, the more difficult it is for a code generation tool-chain to produce efficient machine code.

We believe that code optimization, driven by run-time data characteristics, is a promising solution to tackle this issue. To achieve this, we propose deGoal: a lightweight runtime solution for code generation.

deGoal was designed with the limitations of embedded systems computing power and memory in mind: our bottom-up approach allows code generation of specialized kernels, at runtime, depending on the execution context, the features of the targeted processor, and furthermore on the *data to process*: their characteristics and their values. Runtime code generation is achieved thanks to tiny *ad hoc* code generators, called *compilettes*, which are embedded into the application and produce machine code at runtime. *Compilettes* have only the strict necessary processing intelligence to perform the required code optimizations. As a consequence, code generation is:

A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 107–112, 2014.

1. *very fast*: 10 to 100 times faster than typical JITs or dynamic compilers which allow to use code generation inside the application and during the code execution, not in a virtual machine.
2. *lightweight*: the typical size of *compilettes* is only a few kilobytes which allows its use on constrained memory micro controllers such as the Texas Instrument MSP430 which has only 512 bytes of available memory [1]. Standard code generators, such as LLC of the LLVM infrastructure, have Mbytes of memory footprint, making their use impossible in this context.
3. *produce compact code*: as we are able to generate only the needed specialized code and not all variants at the same time.
4. *portable* across processor family: i.e. a *compilette*is portable on RISC platforms or on GPU platforms.
5. able to perform *cross-jit* applications, i.e. a *compilette* can run on one processor model and generate code for an other processor and download the generated code.

2 Introduction to the `deGoal` Infrastructure

The `deGoal` infrastructure integrates a language for kernel description and a small run-time environment for code generation. The tools used by the infrastructure are architecture agnostic, they only require a python interpreter and an ANSI C compiler. We briefly introduce the language, how applications are written with `deGoal` and how they're are compiled and executed.

2.1 Kernel Description

We use a dedicated language to describe the kernel generation at runtime. This language is mixed with C code, this latter allowing to control the code generation performed in *compilette*s. This combination of C and `deGoal` code allows to efficiently design a code generator able to:

1. inject immediate values into the code being generated,
2. specialize code according to runtime data, e.g. selecting instructions,
3. perform classical compilation optimizations such as loop unrolling or dead code elimination.

`deGoal` uses a pseudo-assembly language whose instructions are similar to a neutral RISC-like instruction set. The goal is to achieve:

- A rich instruction set focused on vector and multimedia instructions.
- The capability to use the run-time information in the specialized code.
- Cross-platform code generation: the architecture targeted by the *compilette* may also be different from the architecture on which the code generator runs.
- Fast code generation, thanks to the "multiple time" compilation scheme. The intermediate representation (IR) is processed at static compile time. At run time the application has only to generate binary code mixed with data.

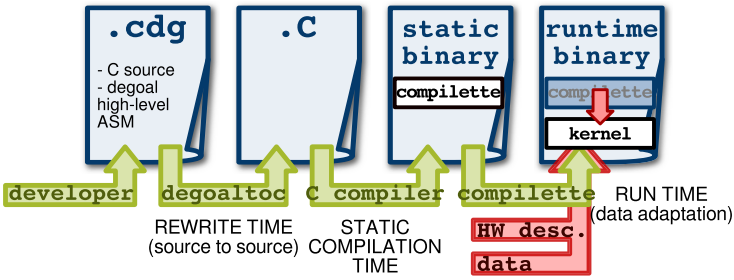

Fig. 1. deGoal work flow: from the writing of applications source code to the execution of a kernel generated at run-time

Table 1. deGoal support status

Architecture	Port status	SIMD support	Instruction bundling
ARM Thumb-2 (+NEON/VFP)	✔	✔	N/A
STxP70 (STHORM)	✔	N/A	✔
PTX (NVIDIA)	✔	✔	N/A
ARM32	✔	✖	N/A
MSP430	✔	N/A	N/A
K1	✔	✔	✔
MIPS	✔	✖	N/A

2.2 Compilation Chain

To achieve fast code generation, the compilation chain is split into several steps that run at different "times" (Figure 1).

Static Compilation. The *compilette* is rewritten into a standard C file by the `degoaltoc` tool. The C version of the *compilette* is then statically compiled and linked to the targeted architecture `deGoal` back-end using the C compiler of the target platform.

Runtime. The application first invocates the *compilette* to generate the machine code of the kernel, once the optimizing data from the execution context are available. The kernel can then be executed as a standard procedure.

Given that the back-end is composed of portable C functions, our compilation chain is able to generate cross-platform code.

2.3 Run-time

At runtime, the *compilette* generates code according to run-time data and environment (rightmost block on Figure 1). At this time, registers are allocated, instructions scheduled and bundled (for VLIW architectures).

3 Current Status

Table 1 details the current support status of `deGoal` (MIPS is a work in progress). The column "SIMD support" shows the ability to take advantage of hardware vectors efficiently. The last column indicates if `deGoal` is able to generate code for VLIW processors.

The core infrastructure is licensed under a BSD style license but all hardware-specific developments are restricted to their respective owners.

4 Related Works

There is an extensive amount of literature about approaches related to our work with `deGoal`. Other works are related :

Java JIT mix interpretation and dynamic compilation for hotpots. Such techniques usually require large memory to embed JIT framework, and performance overhead. Some research works have tried to tackle these limitations: memory footprint can be reduced to a few hundreds of KB [6], but the binary code produced often presents a lower performance because of the smaller amount of optimizing intelligence embedded in the JIT compiler [12].
Java JITs are unable to directly take data value as parameters. They use indirect hotspot detection by tracing the application activity at runtime.
In `deGoal`, the objective is to reduce the cost incurred by runtime code generation. Our approach allows to generate code at least 10 times faster than traditional JITs: JITs hardly go below 1000 cycles per instruction generated while we obtain 25 to 80 cycles per instruction generated on the STxP70 processor.

LLVM [9] (Low Level Virtual Machine) is a compilation framework that can target many architectures, including x86, ARM or PTX. One of its advantages is the unified internal representation (LLVM IR) that encode a virtual low-level instruction with some high-level information embedded on it. Various tools were built on top of it.
In `deGoal`, we don't use IR at run-time, we keep only calls (with parameters) to binary code generators.

Partial evaluation Our approach is similar to partial evaluation techniques [4,8], which consists in pre-computing during the static compilation passes the maximum of the generated code to reduce the run-time overhead. At run-time, the finalization of the machine code consists in: selecting code templates, filling pre-compiled binary code with data values and jump addresses.

Using deGoal we compile statically an *ad hoc* code generator (the compilette) for each kernel to specialize. The originality of our approach relies in the possibility to perform run-time instruction selection depending on the *data* to process [2].

DyC [7] is a tool that creates code generators from an annotated C code. Like `C, it adds some tokens such as @ to evaluate C expressions and inject the results as an immediate value into the machine code.
deGoal is different from DyC because the parameters given to the binary run-time generators can drive specialized optimization such as loop-unrolling or vectorizers.

5 Application Domain Examples

As examples, here are some references of work in different application domains where *compilettes* have been used:

Specialized memory allocator: memory allocators are specialized depending on the size of the memory to manage. *Lhuilier et al* [10] built an example with a very low memory footprint, able to adapt itself to the size of data set.

Hardware support thanks to the fast generation code scheme we are able to generate specialized code which run faster with a low overhead. We have used this support in

- **Mono-core specialization** in an MPSoC context where each node is able to generate an optimized version of a matrix multiplication function [5].
- **GPU code specialization** on an NVIDIA GPU we have developped a "cross-JIT" approach where a CPU generate a specialized GPU code depending on data sets [3].
- **Microcontrollers** with hardware support for floating point arithmetics, where we are able to generate on the fly 10× faster specialized floating point routines [1].

Video compression need specialized code depending on data sets as shown in [11]. deGoal can be used in this domain.

Thanks to the low memory footprint of both code generator and generated code, deGoal is a perfect match for embedded systems.

This article is only an introductory tutorial; results with discussions about acceleration and produced code size can be found in the following bibliography.

References

1. Aracil, C., Couroussé, D.: Software acceleration of floating-point multiplication using runtime code generation. In: Proceedings of the 4th International Conference on Energy Aware Computing, Istanbul, Turkey (December 2013)
2. Charles, H.P.: Basic infrastructure for dynamic code generation. In: Proceedings of the Workshop "Dynamic Compilation Everywhere", in Conjunction with the 7th HiPEAC Conference, Paris, France (January 2012)

3. Charles, H.P., Lomüller, V.: Data Size and Data Type Dynamic GPU Code Generation. In: GPU Design Pattern. Saxe-Coburg publications (2012)
4. Consel, C., Noël, F.: A general approach for run-time specialization and its application to C. In: Proceedings of the 23th Annual Symposium on Principles of Programming Languages, pp. 145–156 (1996)
5. Couroussé, D., Lomüller, V., Charles, H.P.: Introduction to Dynamic Code Generation – An Experiment with Matrix Multiplication for the STHORM Platform. In: Smart Multicore Embedded Systems, ch. 6, pp. 103–124. Springer (2013)
6. Gal, A., Probst, C.W., Franz, M.: HotpathVM: An effective JIT compiler for resource-constrained devices. In: VEE 2006, pp. 144–153. ACM, New York (2006)
7. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: An expressive annotation-directed dynamic compiler for C. Theor. Comput. Sci. 248(1-2), 147–199 (2000)
8. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. 28, 480–503 (1996), http://doi.acm.org/10.1145/243439.243447
9. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002)
10. Lhuillier, Y., Couroussé, D.: Embedded system memory allocator optimization using dynamic code generation. In: Proceedings of the Workshop "Dynamic Compilation Everywhere", in Conjunction with the 7th HiPEAC Conference, Paris, France (January 2012)
11. Sajjad, K., Tran, S.M., Barthou, D., Charles, H.P., Preda, M.: A global approach for mpeg-4 avc encoder optimization. In: 14th Workshop on Compilers for Parallel Computing (2009)
12. Shaylor, N.: A just-in-time compiler for memory-constrained low-power devices. In: Java VM 2002, pp. 119–126. USENIX Association, Berkeley (2002)