Is dynamic compilation possible for embedded systems ?

Henri-Pierre Charles, Victor Lomüller

Is dynamic compilation possible for embedded systems ?

[Extended Abstract]

Henri-Pierre Charles
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LIST, MINATEC Campus, F-38054 Grenoble, France
Henri-Pierre.Charles@cea.fr

Victor Lomüller
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LIST, MINATEC Campus, F-38054 Grenoble, France
v.lomuller@gmail.com

ABSTRACT

JIT compilation and dynamic compilation are powerful techniques allowing to delay the final code generation to the runtime. There is many benefits : improved portability, virtual machine security, etc.

Unforturnately the tools used for JIT compilation and dynamic compilation does not met the classical requirement for embedded platforms: memory size is huge and code generation has big overheads.

In this paper we show how dynamic code specialization (JIT) can be used and be beneficial in terms of execution speed and energy consumption with memory footprint kept under control. We based our approaches on our tool `deGoal` and on LLVM, that we extended to be able to produce lightweight runtime specializers from annotated LLVM programs.

Benchmarks are manipulated and transformed into templates and a specialization routine is build to instantiate the routines. Such approach allows to produce efficient specializations routines, with a minimal energy consumption and memory footprint compare to a generic JIT application.

Through some benchmarks, we present its efficiency in terms of speed, energy and memory footprint. We show that over static compilation we can achieve a speed-up of 21 % in terms of execution speed but also a 10 % energy reduction with a moderate memory footprint.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

JIT, Dynamic code generation

Keywords

Dynamic compilation, binary code generation

SCOPES-2015 SCOPES '15, June 01 - 03, 2015, Sankt Goar, Germany

http://dx.doi.org/10.1145/2764967.2782785 ...$15.00.

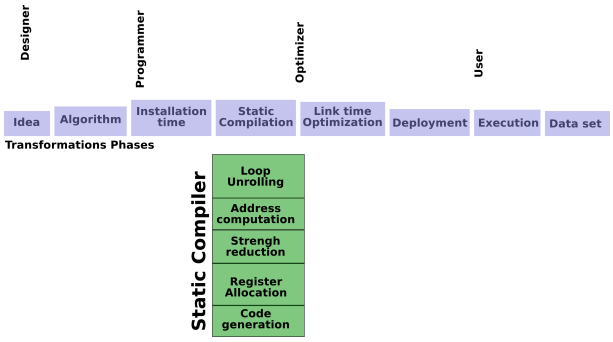

Figure 1: Multiple compilation time for a static compiler

1. INTRODUCTION

The general compiler usage is shown in the figure 1. This figure show the many steps from an intial idea to the running code using a data set on an architecture. The main drawback of this classical scheme is that all important decisions in the compilation process should be taken at static compile time. Unfortunately on modern processors the performance is mainly driven by data characteristics (data size, data alignment, data values). JIT compilation is a trial to delay code generation at run-time.

Just In Time (JIT) compilers' efficiency dramatically improved over the years. In the embedded domain, the increase of processor capabilities has enabled the usage of such technologies. Usually a JIT compiler is used to ensure portability across platforms, but it can also be used to perform program specialization in order to improve performance levels (in general, improve execution speed).

An issue raised by the use of such infrastructure is their resources consumption. By resources, we refer to memory usage, code generation time and energy consumption.

An alternative to JIT is the use of dedicated program specialization routines. These routines are dedicated to the specialization of particular code chunks (basic block, functions). They offer the benefit of performing their task at a low cost by precomputing some specialization works during the static compilation stage.

The research in our team focus on tools which help to improve performance and functionality on embedded platforms. Our final dream is shown in figure 2. In this figure, we can choose, at static algorithmic time, what optimization to implement and when the optimizations will take place (when

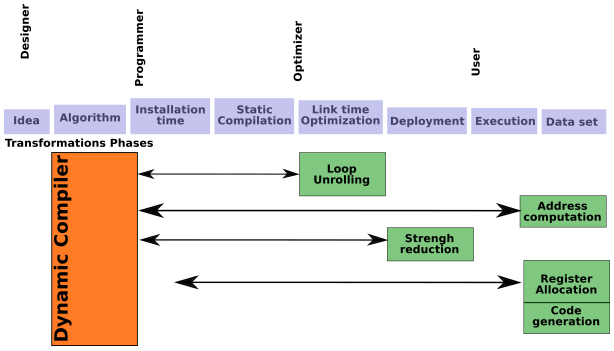

Figure 2: Multiple compilation time for a dynamic compiler

the imformation is available).

In this article, we present two tools that are steps in this final direction : `deGoal` which help to generate `compilettes` and Kahuna to automatically generate such specialization routines.

2. TOOLS DESCRIPTION

This section will give an bird eye view of the two tools `deGoal` and Kahuna.

2.1 `deGoal` code generator

The deGoal infrastructure [2] 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 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 compilettes. 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.

As a consequence, code generation is:

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. 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 familly: i.e. a compiletteis 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.

More information can be found in the article [2].

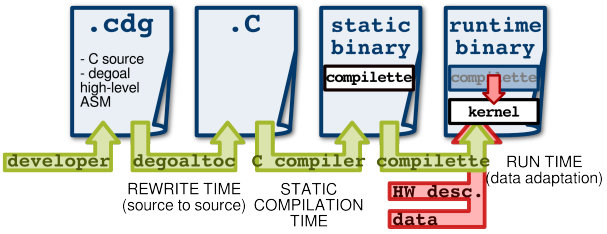

Figure 3: Degoal Compilation flow

The very high flexibility of `deGoal` has a cost : the programmer has to developp manually the `compilette` : which is a small optimizer embedded into the application. The next tool Kahuna has tried to remove this constraint.

2.2 Kahuna

Kahuna is a retargetable runtime program specializer generator independent from any high-level language and that can target embedded platforms. The purpose is to produce specialized function following the context at the lowest cost possible during runtime. It was invented and devolopped by Victor Lomüller during his PhD [6].

An overview of the approach to produce specializers is shown in Figure 4 . A creator (point "C") will annotate a LLVM program to identify (point 1):

- Values unknown at compile time but constant at runtime,
- Location in the program that will trigger the specialization.

This annotated program goes through the Kahuna compilation chain to produce a low overhead specializer (point 2). To do so, the compilation chain produce templates and the specializer that will instantiate the function with those templates. At runtime, when the values are known, the specializer can be trigger to produce the specialized function.

Kahuna can work in 2 modes: out-of-place and in-place mode. Like for most runtime program specializer, the out-of-place mode create a dedicated memory space in which the specialized code is written. This mode has the advantage to allow several specialized functions at the same time and strong runtime optimizations (like dead code elimination, loop unrolling, strength reduction) but at the expense of a longer specialization time. The in-place mode modifies directly the template to create a valid function. This mode allows less runtime optimizations, but considerably lowers down the specialization cost.

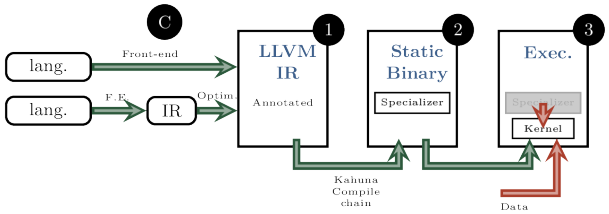

Figure 4: Kahuna Compilation flow

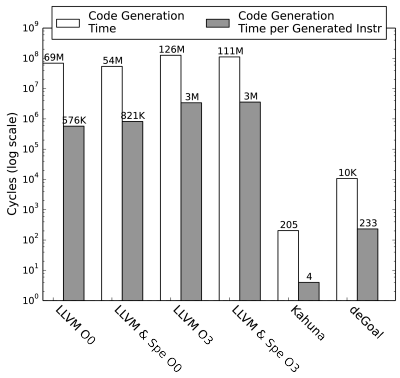

Figure 5: Code generation speed

3. ACHIEVED RESULTS

In this section we compare the results obtained by our tools with differents version of LLVM (Static, JITTed, JITTed with specialization).

We evaluate different approaches against an audio benchmark. SOX[1]: we took a kernel from the SOX application: "lx biquad filter". This function computes a pass-band filter using 5 float coefficients over a block of an audio signal (9 000 samples). We also use this kernel to perform a complete study on execution speed, memory footprint and energy consumption for all approaches described in the previous section.

For this application, the data depend optimization plug the coefficient filter inside the binary code. The advantages are : avoid to reload the coeffcient filter (reduce memory pressure), multiplication become multiplication with a constant.

On this benchmark we experiments 5 different code generation schemes, 3 using LLVM and one using `deGoal` and the last one using Kahuna. On these code generation scheme we have used many metrics.

Due to space constraint, in this article, we only focus on code generation time (how many time to generate the code) and overhead recovery (how many time the code should be executed to amortize the code generation).

3.1 Code Generator Execution Time

There is no big differences in the generated code speed. The following table give speedup compared to the LLVM static version. The results are in % compared to the static version. The LLVM+JIT give no speedup compared to the static version. Kahuna give a 21% speedup acceleration thanks to the applied code specialization. LLVM+JIT+Spe is an implementation where the code specialization is hardcoded into the application. Not surprisingly, it give the same speedup than Kahuna, but with a higher cost which will discussed later. The `deGoal` version give a higher speedup (27%) because it contains it's own code generator.

Scheme	Speedup
LLVM+JIT	0
Kahuna	21
LLVM + JIT + Spe	21
deGoal	27

Figure 5 presents the code generation time in cycles took by the different approaches. It presents both the time took by code generators and time took per generated instruction. Note that the LLVM backend has a broader range of runtime capabilities than Kahuna. The high cost of the LLVM application is a price to pay for those capabilities.

The specialization of the Kahuna version provide an important improvement in terms of speed over the regular LLVM. The Kahuna approach improves its performances by specializing its back-end. All the optimization is made at static compile time, leading to restricted but fast code generator and still provides speed-up for the generated kernel. In the in-place mode, the code generation is limited to emit the instruction and store it at the right place, leading to this very low code generation time. In the out-of-place mode, the code generator also copies the different basic blocs in the created memory space. This increasing the code generation time but its kept negligible compare to LLVM.

The genericity of the LLVM back-end allows efficient code transformations but at a higher cost. The specialization of Kahuna allows fast code generation with a resulting code quality equivalent to LLVM.

More specific results for SOX, the deGoal approach improve performance by specializing the back-end. At runtime, there is no automatic scheduling and only limited optimizations, such as instruction selection, simplified register allocation and peephole optimizations. With this tool, most of low-level optimizations are made manually leading to both a fast code generator and generated code. The difference between Kahuna and deGoal comes from the fact that Kahuna uses the LLVM scheduler (at static compile time), which leads to the same performances as LLVM & Spe. However, the followed path in the LLVM back-end is not exactly the same between the Kahuna and the LLVM & Spe versions. Kahuna benefits from generic optimizations at static compile time. Unlike LLVM & Spe, Kahuna handles constants through moves instead of loads.

3.2 Overhead recovery

Another important metric, and one that makes the link between the two previous ones, is the "overhead recovery". In other words, the number of required samples to process (workload) in order to recover the runtime overhead produced by the specialization process and surpass the static version.

Table V presents the overhead recovery (in samples) of the different specialization techniques. The column "Workload" represents the workload (number of signal samples) that can be processed by the static version during the code generation time. It gives an idea of the specialization weight over the performance of the static implementation. The "Overhead Recovery" column shows the required number of samples required to be faster, in average, than the static version when we take into account both code generation time and processing time.

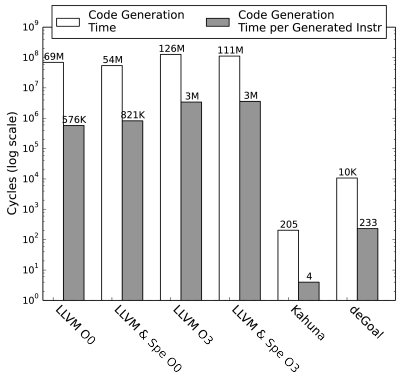

Figure 6: Overhead recovery

Thanks to the specialization of the back-end, the Kahuna have a small overhead recover or even null one has it requires only 1 samples. Converted into audio signal length, the over-head recover is 0.2 ms for the SOX filter in its in-place mode. Such results would allow efficient specialization in applications that needs to change values during the process.

For deGoal, the results are similar to Kahuna in its out-of-place mode with the SOX benchmarks. For the LLVM & Spe option, as the code generation time is important, the overhead is as important. Converted to an audio signal, its represent 90 s of signal.

It is interesting to note that all three specialization schemes provide similar results in terms of produced code efficiency, but at a very different cost. The overhead recovery provides a tangible way to evaluate the cost of specialization. In terms of speed, this evaluation is classical, we have used the same metrics for energy and shown that it is harder to pay off specialization from this point of view.

Regarding the applicative domain, the low overhead provided by Kahuna mean that a dynamic code generation can be used in an audio application without noticeable interrup (The well known "Klic" in audio applications).

4. RELATED WORKS

Java JIT mix interpretation and dynamic compilation for hostpots. 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 , but the binary code produced often presents a lower performance because of the smaller amount of optimizing intelligence embedded in the JIT compiler [7]. 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

Fabius [3] specializes ML routines. At compile time , the system identifies variable to specialize and create templates to specialize the routine at runtime. DyC [**?**] offers also low-overhead runtime specialization mechanism. It starts with an annotated C program, and similarly to Fabius, it creates templates that will be instantiated at runtime. Tempo [**?**], also starts with C programs, but uses "scenarios" to describe the specializing process. To our knowledge, Tempo is the only specialization system described above that has been used on an embedded platform [2]. Yet, all these specialization techniques only focused on timing and size performances, the energy performance was not studied. deGoal uses its own language and emits instruction on a per instruction basis. Kahuna uses a per block approach like DyC, but is not tied to any language, leaving the control of the code generator to the front-end or optimization passes. Finally, Kahuna and deGoal were developed for embedded system and are easily retargetable.

We have other results using dynamic compilation that can be found in the following publications : [5] using NVIDIA GPU processors, [4] which describe the thechnology more precisely.

5. CONCLUSION

We have shown in this article that using sophisticated code generation at run-time is possible on small processors dedicated to embedded systems.

There is no ideal tool so far, but we have shown that it is possible to build tools with good compromise between the code quality produced at high speed and low overhad recovery.

6. REFERENCES

[1] C. Bagwell, R. Sykes, and P. Giard. SoX - Sound eXchange - v14.4.1.

[2] H.-P. Charles, D. Couroussé, V. Lomüller, F. A. Endo, , and R. Gauguey. degoal a tool to embed dynamic code generators into applications. Jan 2014.

[3] M. Leone and P. Lee. A Declarative Approach to Run-Time Code Generation. In *In Workshop on Compiler Support for System Software (WCSSS*, pages 8–17, 1996.

[4] V. Lomüller and H.-P. Charles. A LLVM Extension for the Generation of Low Overhead Runtime Program Specializer. In *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems - ADAPT '14*, pages 14–16, Jan 2014.

[5] V. Lomüller, S. Ledru, and H.-P. Charles. Scilab on a Hybrid Platform. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, Advances in Parallel Computing, pages 743–752. IOS Press, 2014.

[6] V. Lomüller. *Générateur de code multi-temps et optimisation de code multi- objectifs*. PhD thesis, École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique" Université de Grenoble, 11 2014.

[7] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Java Virtual Machine Research and Technology Symposium*, pages 119–126, 2002.