How to generate data dependent applications and handle platform heterogeneity ?

Henri-Pierre CHARLES, Maha KOOLI, Thaddée BRICOUT, Benjamin LACOUR*

CEA, LIST, Université Grenoble Alpes, F-38000 Grenoble, France

FirstName.LastName@cea.fr

Abstract

Modern applications are dynamic. There are many dynamicity aspects : dynamic allocation, user chosen data sets, user chosen parameters, dynamic / interpreted languages, indirect memory access, transprecision ... `RISCV-V` modern processors contains also dynamic behaviour : data and instruction caches, pipelines ...

Aggressive compilers are unable to catch so many dynamic application aspects, but there is a need to adapt binary code to those two dynamic aspects because run-time behaviour contains huge optimization opportunities.

`RISCV-V` has also a specific difficulty : since the ISA is open, there is many implementations with subtle variants : specifics accelerators, special instructions, special register banks. Those variants are difficult to handle with a monolitic compiler.

`HybroGen` is an experimental compiler designed to build application with dynamic binary code adaptations.

Introduction

Modern applications behaviour are data dependent because of the dynamicity of modern applications. This dynamicity may come from the dynamic memory allocation, since the size of the data set is generally chosen by the user, at run-time. Other dynamic behaviour may come from indirect memory access used in complex application : graph handling, sparse computations, data sorting, etc.

In another direction, hardware designers have chosen to help application with statistically helpful accelerators : multilevel data caches, deep instructions pipelines, branch predictors, etc.

Mixing those two dynamicity axes make the compiler tasks very complex. This presentation advocates the need for modern application to have a possible application binary dynamic reconfiguration scheme that can adapt running code to dynamic parameters.

Compilation scenarios and compiler global view

The figure 1 illustrates our objective regarding the code generation scenarios for a repetitive call to a computing kernel : instead of having a static compilation scenario and repetitively call the same binary kernel (scenario a), we want to proceed by a dynamic compilation (scenarios b) and thus have the capacity of either :

1. generate the binary kernel at the beginning (Program initialization scenario),

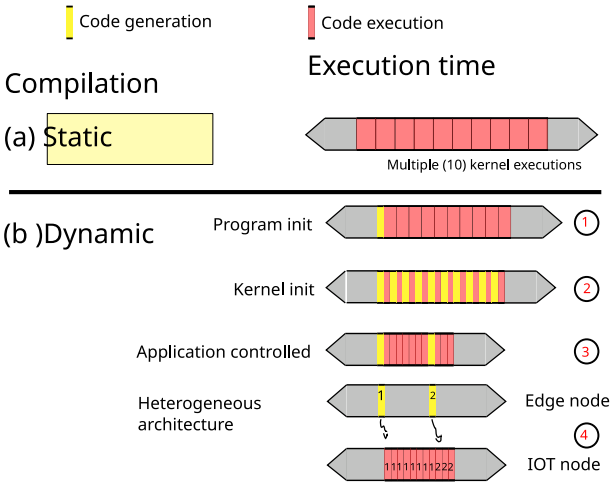

Figure 1: *Compilation scenarios that can be done with `HybroGen`*

2. generate the binary at each kernel call (Kernel initialization scenario)
3. or generate the binary during the code execution at a frequency controlled by the programmer (Application controlled scenario).

Of course all those scenarios are useful only if we are able to generate code generators fast enough to benefit from the faster code generated. This is not the case for standard dynamic code generators like in Java JIT or in LLVM. Our code generator is faster than classical Just in time compiler.

In order to setup those compilation scenarios we have developed an experimental compilation platform which is shown on figure 2. It contains many software blocks that are run at different “compilation times”.

1. At “installation time”, instructions sets are cate-

*Corresponding author: `Henri-Pierre.Charles@cea.fr`

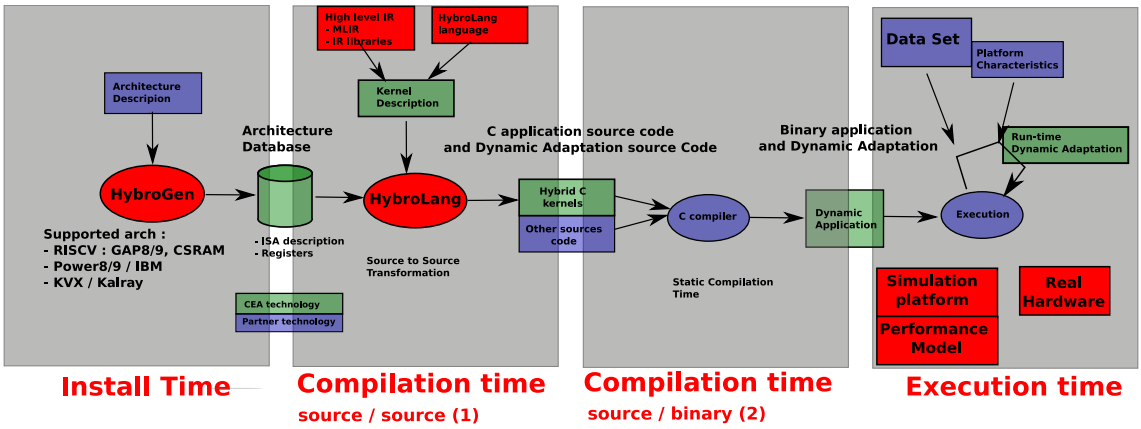

Figure 2: *Global view of the* `HybroGen` *compiler*

gorized and put into a relational database.

2. At "source to source" compilation time a part of the code analysed and transformed in a C runtime code generator.
3. Then a standard compiler (`gcc` or `clang`) transform the C code in a binary form
4. At "run time" the code is executed and the kernel binary code is regenerated depending on the code generation scenario. The binary code can be run on a silicon platform or on an emulator such as QEMU.

Those scenarios can be used in many application examples described in the following paragraphs.

Application examples

Code heterogeneity

We know that the `RISCV-V` platform can have multiple ISA variants and the programmer may want to have the same binary code running on multiple variants of the RISC-V platform.

This example will use the scenario "program initialization" (1)

Code specialization

In this publication [1], we have shown the possibility to specialize the running code in order to specialize for performance depending on the data parameters.

This example use the scenario "Kernel initialization" (2)

Transprecision

For numeric applications it can be useful to dynamically change the data type at run time. In this publication we have shown [2] the capacity of `HybroGen` to change the datatype of computing variables during the code execution.

This example uses the scenario "application controlled" (3)

Computing in memory and Heterogneity

Our compiler can also be used for heterogeneous platforms which use accelerators with multiple ISA. This is the case for the platform described here [3] which contains a `RISCV-V` platform and a "compute in memory" accelerator.

In this scenario we use a more elaborated scenario where the `RISCV-V` platform generate instruction for the accelerator on the fly.

This example uses the scenario "heterogeneity" (4)

Conclusion

`HybroGen` is opensource (`https://github.com/CEA-LIST/HybroGen`) as well has the QEMU plugin allowing to functionnaly emulate the "compute in memory" functionnality

The current public release support multiple architectures : multiple RISCV variants, ARM AArch64, Kalray, IBM Power and a CEA "compute in memory" platform which use a RISCV.

References

[1] Damien Couroussé and Henri-Pierre Charles. "Dynamic code generation: An experiment on matrix multiplication". In: *Proceedings of the Work-in-Progress Session, LCTES* (2012).

[2] Julie Dumas et al. "Dynamic compilation for transprecision applications on heterogeneous platform". In: *Journal of Low Power Electronics and Applications* 11.3 (2021), p. 28.

[3] J-P Noel et al. "A 35.6 TOPS/W/mm^2 3-stage pipelined computational SRAM with adjustable form factor for highly data-centric applications". In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 286–289.