Deliverable D2.3 - Illustration of system reconfiguration due to varying conditions: same-island, and migration

Erven Rohou

CONTINUUM

A FRENCH ANR PROJECT

Design Continuum for Next Generation Energy-Efficient Compute Nodes

Project Ref. Number ANR-15-CE25-0007

D2.3 – Illustration of system reconfiguration due to varying conditions: same-island, and migration

Version 1.1
10 September 2021
Final

Public Distribution

Inria

Project Partners: Cortus S.A.S, Inria, LIRMM

Project Partner Contact Information

Cortus S.A.S Michael Chapman 491 Rue Charles Nungesser 34130 MAUGUIO France Tel: +33 4 30 96 70 00 E-mail: michael.chapman@cortus.com	**Inria** Erven Rohou Inria Rennes - Bretagne Atlantique Campus de Beaulieu 35042 Rennes Cedex France Tel: +33 299 847 493 E-mail: erven.rohou@inria.fr
LIRMM Abdoulaye Gamatié Rue Ada 161 34392 Montpellier France Tel: +33 4 674 19828 E-mail: abdoulaye.gamatie@lirmm.fr	

Table of Contents

1 Introduction

Deliverable D2.3 is of type *Software*. This document shall be considered as accompanying documentation. The deliverable itself consists of an archive containing C/C++ source code.

Deliverable D2.3 demonstrates system reconfiguration at two different levels.

Same-island: the code keeps running on the same core, but a different version some key functions is used. It may consist in a different implementation of the function, or the same implementation optimized differently.

Migration: on a heterogeneous multicore system, the same code migrates to a different core to continue execution.

The most sophisticated way to generate code during execution would be to rely on a just-in-time (JIT) compiler or dynamic binary rewriting (see for example fittChooser [1]). These engines, however, are very complex pieces of software with huge development times. In addition they typically require OS support not available on small dedicated embedded systems. We relied on a much simpler approach that proved sufficient to demonstrate our approach.

When the necessary support is not available, we rely on multi-versioning: all the versions are prepared statically and must be available on the device. At runtime, the orchestrator only selects the best version from the available ones.

1.1 Poor man's JIT compiler

Our approach relies on relatively simple mechanisms provided by the Linux operating system.

We require that the function to be recompiled and all its callees be located in the same file (aka a compilation unit)[1]. This is usually not a constraint since critical functions are often computational kernels with little interaction with other parts of the application.

When a new version of the function is needed, the system proceeds as follows.

1. Invoke a regular static compiler (GCC, Clang...) on compilation unit, specifying the flags to generate a shared library (i.e. `-fpic -shared`).
2. Use `dlopen` to load the shared library in the application address space.
3. Use `dlsym` to retrieve the address of the function in the library. This address can be used, after proper casting, as a regular C function pointer.

This approach is used on x86 and Arm Linux platforms.

In the delivered source code, this can be found in the file `jit.c`.

[1]This is not strictly necessary, but it facilitates implementation.

1.2 Multiversioning

On targets without modern operating system, such as the Cortus devices, we instead rely on multiversioning.

The compilation units (e.g. `foo.c`) are compiled several times to generate several object files (e.g. `foo1.o`, `foo2.o`, `foo3.o`). Care must be taken to make all symbols *static* and to rename the entry point to avoid multiply-defined symbols. All object files are then linked together with the main program.

2 Experiments

We experimented with a MPEG encoder[2] where the critical function is the encoding of a single frame. The metrics we chose to tune the software are time per frame and energy per frame. We varied frame sizes and image content to observe the behavior of the auto-tuning mechanism.

This section presents selected results obtained thanks to the software delivered as D2.3.

2.1 Setup

We experimented with the following configurations:

1. Intel Core i7-5600U CPU, clocked at 2.60 GHz, running Linux 4.17.11, GCC 7.2.1 and Clang 5.0.0. Energy is measured thanks to the Intel RAPL energy monitors.
2. Odroid XU4, featuring an Arm big.LITTLE octocore processor, clocked at 2.0 GHz/1.4 GHz, running Linux 3.10.105, GCC 4.8.5. Energy is measured thanks to an external *SmartMeter* probe[3].

At the compilation level, we experimented with a few compilation flags as a proof-of-concept: `-O0`, `-O2`, `-O3`, `-march=native`, `-ffast-math`. All valid flags are obviously possible. Flags are defined in the file `s_config.def`.

In addition, we experimented with:

- a multi-threaded implementation where threads in a thread-pool[4] wait for frames to encode;
- a substitution of floating point number by a fixed-point implementation. This is useful for configurations that lack a hardware floating point unit, such as some ultra low-power Cortus cores.

2.2 Results on x86

On our x86 platform, cores are homogeneous, making migration a non-issue (although functional). In addition, instantaneous power consumption is mostly constant during the run of our workload, hence energy and time are equivalent metrics.

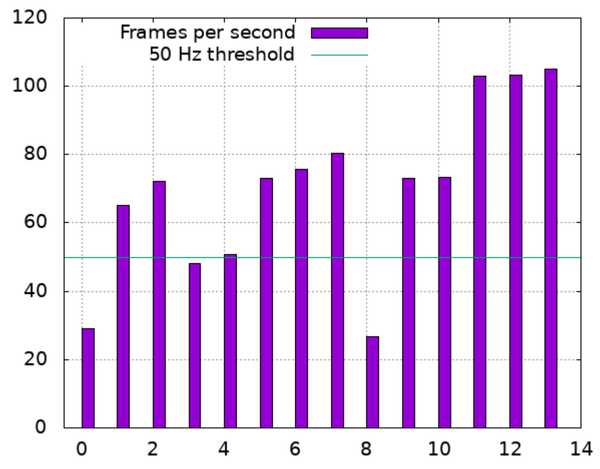
Figure 1 illustrates the various solutions explored by the code generator and their respective performance (in frames per second).

The thread pool is functional, but does not provide any advantage on this example. This is due to the granularity of the critical function: encoding a single frame is fast, in the order of the overhead of assigning a thread to a task.

[2]Public domain software from Jon Olick, available at `http://jonolick.com/uploads/7/9/2/1/7921194/jo_mpeg.cpp`.

[3]`http://odroid.com/dokuwiki/doku.php?id=en:odroidsmartpower`

[4]Thread pool from Johan Hanssen Seferidis, available at `https://github.com/Pithikos/C-Thread-Pool`.



0: gcc -O0
1: gcc -O2
2: gcc -O3
3: gcc -O2 +fixed point
4: gcc -O3 +fixed point
5: gcc -O2 -march=native
6: gcc -O3 -march=native
7: gcc -O3 -march=native -ffast-math
8: clang -O0
9: clang -O2
10: clang -O3
11: clang -O2 -march=native
12: clang -O3 -march=native
13: clang -O3 -march=native -ffast-math

Figure 1: Number of frames per second achieved by encoder on x86 platform, depending on compilation flags

2.3 Results on Arm big.LITTLE

The big.LITTLE platform supports migration of tasks across cores. We combined dynamic code generation and migration and measured both execution time and energy consumed for each configuration. Figure 2 illustrates the results. We can clearly see the tradeoff between faster and more energy-hungry computations of the big core, and slower and cheaper LITTLE cores.

To assess the impact of computations with fixed-point representation of decimals, we compared two versions:

1. regular floating-point, but forcing the compiler to rely on the software libraries (flag `-mfloat-abi=soft`);

2. fixed-point library.

Figure 3 shows the performance of both implementations, on big and LITTLE cores.

2.4 Cortus

Due to the lack of OS support for a JIT-like auto-tuner, we rely on multiversioning for the auto-tuner on the Cortus hepta-core. No output can be shown, due to the limited bandwidth between the FPGA board and the PC, so we report only performance metrics.

In this experiment, we used a Cortus hepta-core mapped [2] on an FPGA board, illustrated in Figure 4. It consists of a high-performance (HP) core of type APSx2, a dual-issue superscalar, and two low-power (LP) clusters made of a combination of APS25 and FPS26 core. The latter feature a floating-point unit.

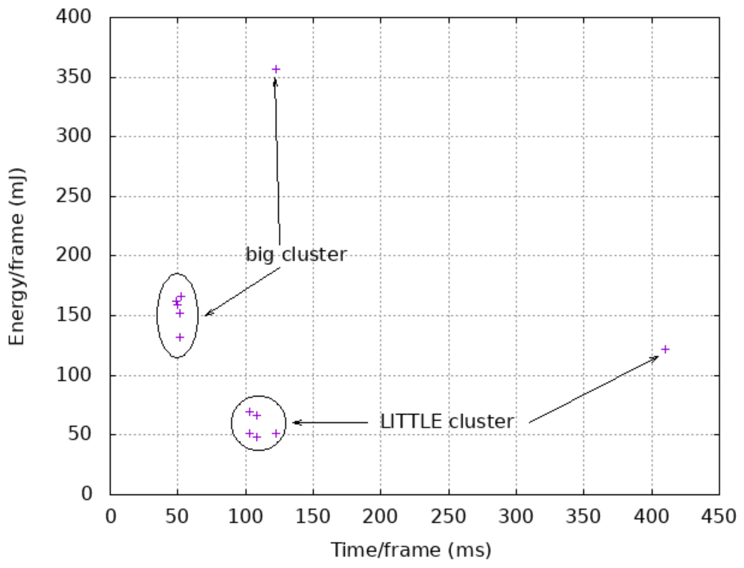


Figure 2: Performance/energy trade-off on Arm big.LITTLE. Outliers represent `-O0`.

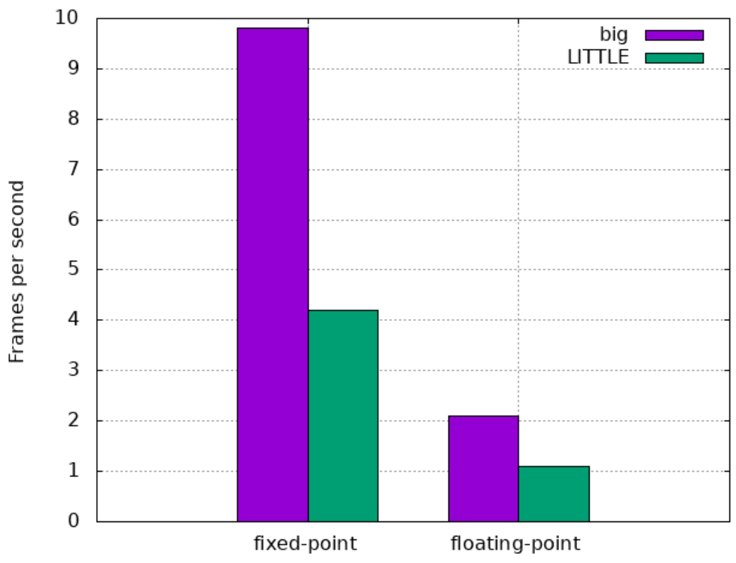


Figure 3: Impact of fixed-point representation on FPU-less processors

Our autotuner encodes one frame on each available core, and for various optimization levels. Figure 5 illustrates the resulting numbers.

Clearly, the APS25 cores are penalized by the lack of floating point unit. All computations are emulated in software, resulting in a 10x slowdown. When switching to a fixed-point implementation [5] (32-bit containers, 16 bits of fractional part), we observe that the performance is close to what the FPS26 delivers. Note that the quality of produced frames is lower, the FPS26 is preferred is available. For reference, we also report the performance of the fixed-point representation on APS26: it is slightly worse that native floating point, but very close.

Unexpectedly, the optimization level `-Ofast` is not the best on this architecture: `-O2` performs better.

Finally, despite a more advanced micro-architecture, the APSx2 is not significantly faster than the FPS26 on this application. The APSx2 may be better used for overall orchestration, leaving actual computations to the more power-efficient FPS26.

Combining this raw numbers and energy data (unavailable on our FPGA implementation), the auto-tuner is able to define and deploy its strategy. It may range from using all core for highest throughput at the cost of maximum energy consumption, to carefully selecting the cores that achieve the required number of frames per second in the most energy-efficient way.

[5] Since this type is not standard, the fixed-point representation was developed from scratch at Inria as a C++ class.

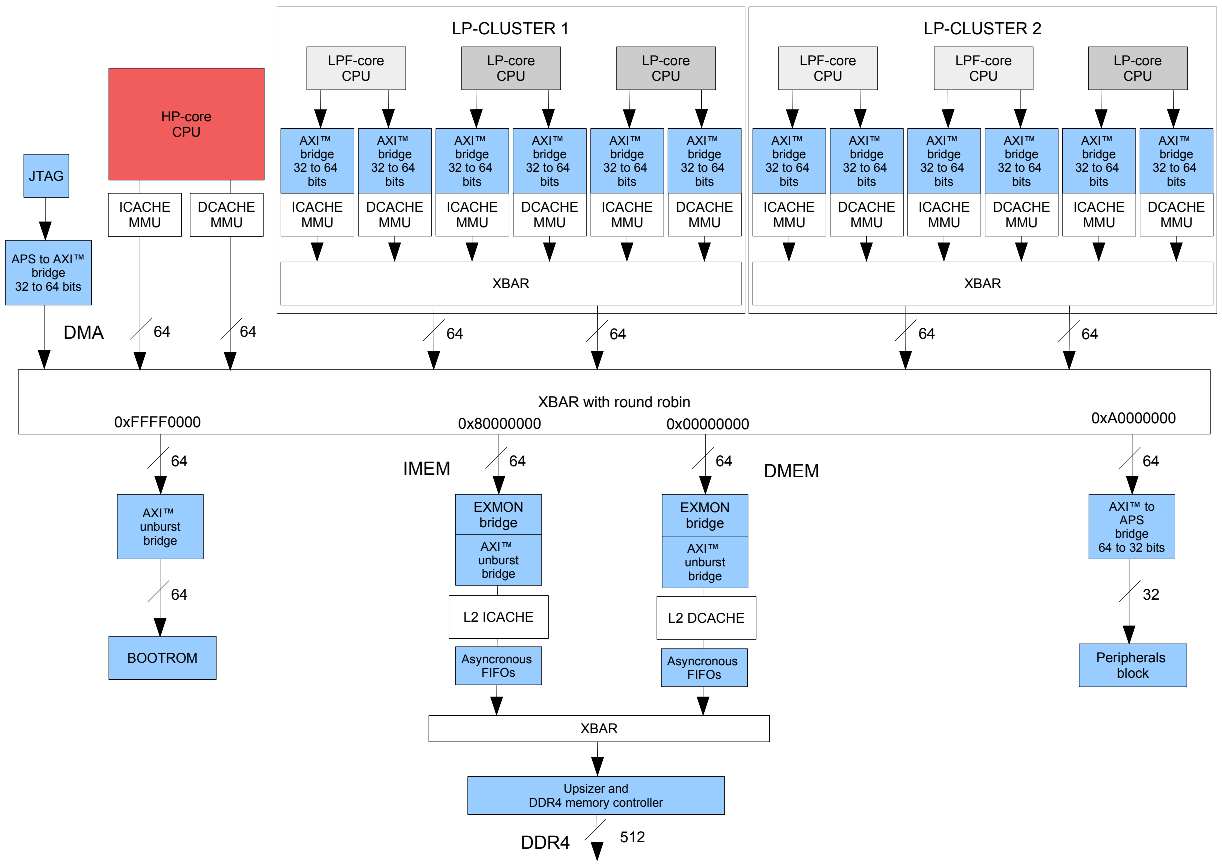


Figure 4: Schematics of the Cortus heptacore

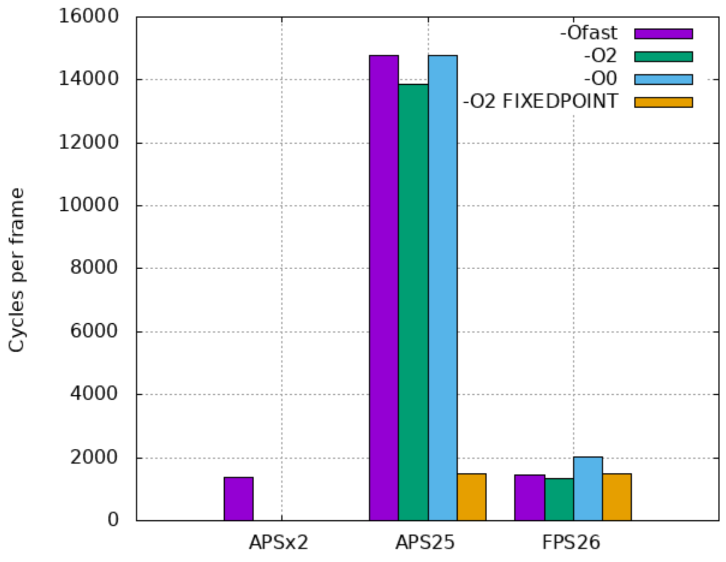


Figure 5: Impact of core type and optimization level on Cortus multicore

3 Conclusion

This document is the accompanying report for deliverable D2.3, which is of type *Software*. Through a few experiments we illustrated the capability of the autotuner to generate code on-the-fly and to migrate across core to adjust to varying conditions.

References

[1] Ap, A. A., Le Bon, K., Hawkins, B., and Rohou, E. fittChooser: A dynamic feedback based fittest optimization chooser. In *2018 International Conference on High Performance Computing Simulation (HPCS)*, pages 98–105, 2018. doi: 10.1109/HPCS.2018.00031.

[2] Gamatié, A., Devic, G., Sassatelli, G., Bernabovi, S., Naudin, P., and Chapman, M. Towards energy-efficient heterogeneous multicore architectures for edge computing. *IEEE Access*, 7: 49474–49491, 2019. doi: 10.1109/ACCESS.2019.2910932. URL `https://doi.org/10.1109/ACCESS.2019.2910932`.