Project no. 644235

RePhrase

Research & Innovation Action (RIA)
Refactoring Parallel Heterogeneous Resource-Aware Applications – a Software Engineering Approach

Report on the Interoperability of Adaptivity Tools D5.3

Due date of deliverable: 31st January 2017

Start date of project: April 1^{st}, 2015

Type: Deliverable
WP number: WP5

Responsible institution: University of St Andrews
Editor and editor's address: Vladimir Janjic, University of St Andrews

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	13/02/17	Vladimir Janjic	USTAN	Created initial version
2	14/02/17	Chris Brown	USTAN	Fixed minor typos and editorials
3	14/02/17	Vladimir Janjic	USTAN	Added the dependency diagram
4	14/02/17	Vladimir Janjic	USTAN	Added citations

Contributions Per Partner

Institution	Contribution
USTAN	Introduction (Chapter 1), Interoperability of the adaptivity toolchain (Chapter 2) and Conclusions (Chapter 3)

Executive Summary

This document is the third deliverable of the WP 5 “Integration and Overall Software Engineering Methodology”. The purpose of this work package, according to the DoW, is to investigate the issues of interoperability with respect to the set of the tools developed in other work packages, transforming them into a coherent parallel software development methodology. The deliverable is the result of the first phase of the task T5.2, and it describes interoperability of the adaptivity tools set described in deliverables D4.1 and D4.2.

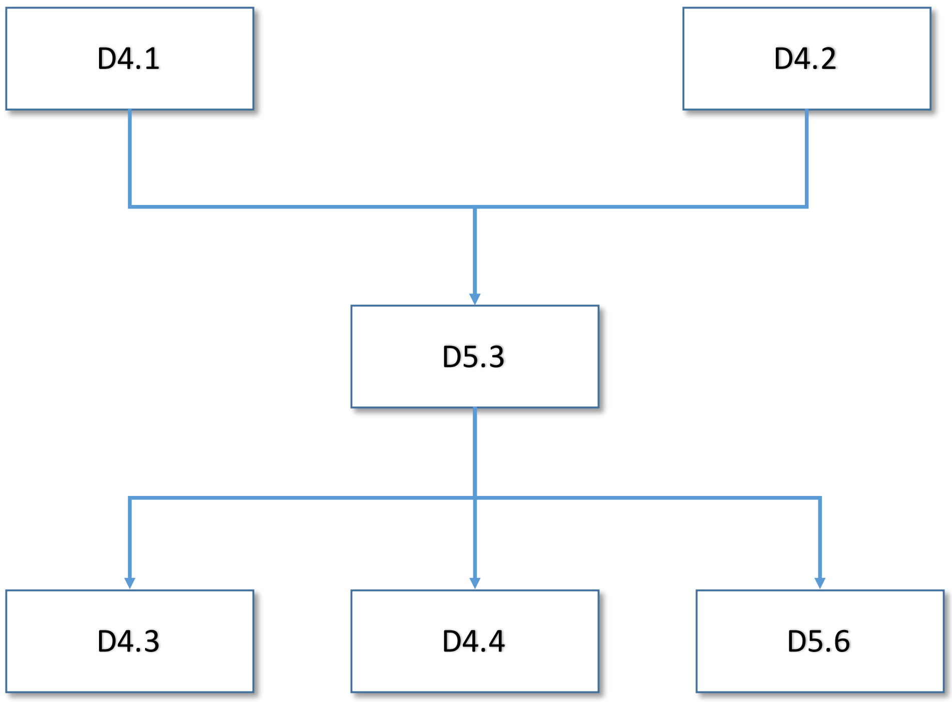

Figure 1: Dependencies on other deliverables

Contents

Chapter 2

Introduction

In this deliverable we describe the mechanisms to ensure interoperability of the adaptivity tool chain described in deliverables D4.1 and D4.2. As a reminder, these tools take as an input a patterned parallel application with predetermined shape, as prepared by the tools and libraries developed in WP2 and WP3 and i) decide on an initial instantiation of the application (in terms of number and type of components) and initial distribution of data; ii) dynamically switch between different prepared versions of the same component, as a response to the changes in the application behaviour or system load; iii) allow dynamic rescheduling of application threads of the underlying resources; and, iv) monitor application behaviour and the underlying hardware environment and detect parallelism bottlenecks. This ensures dynamic adaptation of the software to the changes both in the environment and in the application itself, e.g. when the application goes through different phases of parallelism or when type of the input changes. In this way, we address a crucial aspect of software engineering for heterogeneous parallel computing systems, thus increasing reliability, robustness and adaptivity of parallel software.

Deliverables D4.1 and D4.2 described each of these tools, but did not described how all of them work together, i.e. how outputs that one tool produces can be used as inputs to the tool that is next in the tool chain. Section 3.1 provides a short overview of the adaptivity tools, together with the "big picture" of how all of them fit together. Section 3.2 then discusses inputs and outputs of each of the tool, and how these need to be modified to ensure interoperability and the overall flow of the work of the tool-chain. Chapter 4 concludes and describes the future work that needs to be carried on in the workpackage WP5.

Chapter 3

Interoperability Between the Tools of the RePhrase Dynamic Adaptivity Tool-Chain

The goal of the **RePhrase** tools for dynamic adaptivity is to take an existing "partial" application with a pre-determined parallel structure and, firstly, complete the application and, subsequently, ensure that the application is able to adapt to the changes in both its behaviour and in the execution environment. A partial application has a fixed parallel structure, in terms of the parallel patterns used and their composition, but possibly has values for some extra-functional parameters (such as number of CPU and GPU workers for each farm or chunking parameters) still unassigned. A near-optimal values of these parameters are first determined, and the application then goes to the dynamic compilation stage, where a binary code, together with the intermediate representations of the alternative versions of the components of the code that might be dynamically switched, is produced. The application is then executed, and the dynamic scheduler and performance monitor control the execution and adapt the application execution to respond to changes in the environment, such as change in the load as a result of new applications entering the system, and in the application itself, such as divergence of the inputs fed to the application to the expected ones. Figure 3.1 shows the overview of the workflow for the adaptivity tool-chain.

In the subsequent sections, we describe briefly each tool in the tool-chain and outline, before discussing the interoperability issues.

3.1 Dynamic Adaptivity Tools

3.1.1 Static Mapping

The static mapping infrastructure takes a partial parallel application and, firstly, completes it by deriving near-optimal values values for extra-functional parameters

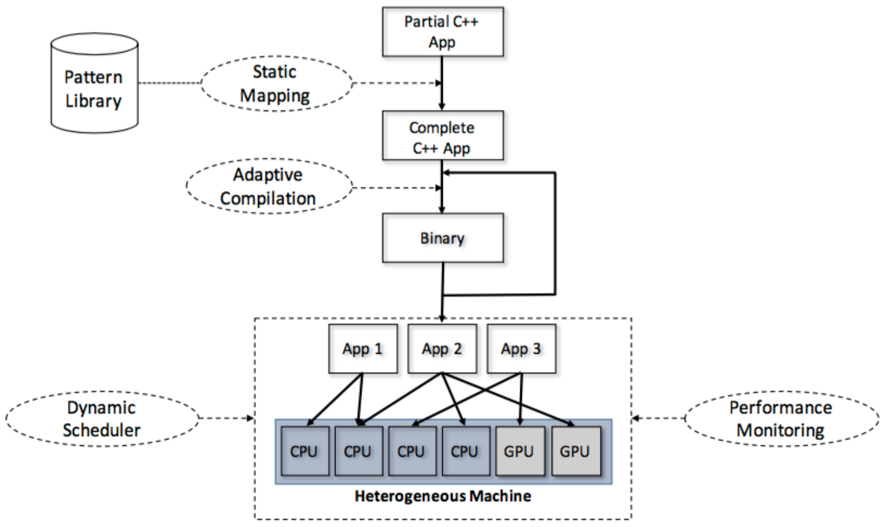

Figure 3.1: **RePhrase**Adaptivity Tool-Chain

and, secondly, chooses an appropriate implementation of the given parallel patterns from the library of patterns. For example, the partial application can have the following pattern structure

$$\text{Pipe}(\text{Farm}(C_1, G_1), \text{Farm}(C_2, G_2), \text{Farm}(C_3, G_3)),$$

where Pipe and Farm represent, respectively, parallel pipeline and tasks farm parallel patterns, and the parameters for each farm pattern are the number of threads running the farm computation on CPU cores (C_i) and the number of threads running the computation on GPU devices (G_i). Other extra-functional parameters, such as size of a chunk in cases chunking is needed to improve granularity, can also be left unassigned. The static mapping infrastructure, then, uses heuristics described in D4.1 to derive near-optimal values for the parameters C_i and G_i, and then select the appropriate implementation of each of the task farms (e.g. the ones that explicitly replicate the shared data on memory nodes of a NUMA machine). In this way, we decide on how many threads and of which type to create for each parallel pattern in the application, as well as the initial versions of the components that will be used, and that can possibly be dynamically replaced by their alternative versions during the execution time by the dynamic compiler. By choosing the appropriate pattern implementation, we also decide on initial placement of data with respect to the memory layout of the target machine, which is especially important for data-intensive applications.

3.1.2 Dynamic Compilation

The dynamic compiler is based on the LLVM compiler infrastructure [4], and aims, given multiple versions of the same function, to enhance run-time execution of the program by choosing the appropriate version to run, given the current objective metric (e.g. performance or energy consumption). This is done by dynamically compiling the different versions of the function and measuring the appropriate metrics (e.g. execution time in the case of performance). The dynamic compiler takes as an input a complete C++ source, as produced by the static mapping infrastructure, and produces a binary version of the code, together with the intermediate representation of the alternative versions of the functions that might be dynamically interchanged. For example, we can have different implementations of a task farm, one which does explicit replication of the data and the other that lets the operating system deal with data placement. We can start with the latter version, as decided by static mapping, but also create intermediate representation of the former. At some point during the application execution, for example if the inputs fed to the application become large, we might decide to dynamically switch to the version that does explicit data replication, which would be achieved by compiling its intermediate version into the binary code and replacing all the addresses of the farm function within the application binary with the address of the newly compiled function.

Dynamic compilation requires an external *controller* code that will receive the information from the performance monitoring infrastructure and then, based on that, decide when the recompilation is needed, select the appropriate new implementation of the target function and invoke dynamic compiler to replace the existing implementation of the function with the new one. We are currently in the process of implementing this, and this feature is still not a part of the **RePhrase** adaptivity tool-chain.

3.1.3 Dynamic Scheduling

In D4.1, we described the `PaRLSched` library, a dynamic scheduler for remapping application threads and data to the available resources developed in the **RePhrase**project. The goal of the dynamic scheduler is to increase the robustness and resilience of parallel and data-intensive applications when running under non-homogeneous and possibly shared hardware resources. The scheduler is fully automatic and is able to automatically discover optimal allocation of resources, independently of the nature of the applications (advanced parallel patterns, data-intensive applications, etc.) and potential (dynamic) changes in the environment (e.g., availability of resources). The current version of the scheduler makes dynamic decisions about the pinning of the threads to the cores of a multi-core CPU, and mapping of the data to the memory banks of a NUMA machine. It serves as a replacement for the mechanisms that are available in operating systems, in the cases where the performance of individual cores of the CPU drops because of, for example, an increased load of the system. The scheduler is based on *reinforcement learning* algorithm that periodically checks the performance of cores and makes small adjustments to the mapping of the threads to the cores.

3.1.4 Performance Monitoring

Performance monitoring mechanisms collect various type of information during the application execution and use the information to discover reasons for performance bottlenecks in the patterned applications. In Deliverable D4.1, we described different libraries for performance monitoring that give information at different levels. We described the native monitoring mechanisms available in the FastFlow pattern library that give high-level information related to parallel patterns, e.g. the number of items in the queues that connect different stages of a pipeline pattern that can be used to detect bottlenecks in pipelines, or the service time of each worker in a farm. This kind of information can be used to pinpoint bottlenecks that come from the overall structure of the parallelism allowing us to, in combination with dynamic compilation, change the overall parallel structure of the application, or alternatively replace the current version of a component with its alternative representation. We also described the Mammut [1] and PMLIB [3] libraries that work on lower level and can give information about, for example, energy consumption of individual cores. This can be used by the dynamic scheduler to remap threads and data and thus adapt to the drop in performance.

Both the higher and the lower level mechanisms work by instrumenting the source code of the application, i.e. inserting the calls to the appropriate library functions. Currently, this has to be done manually by the programmer, but we envisage that the **RePhrase** refactoring tool will be able to assist with introducing the appropriate calls in the user code.

3.2 Interoperability Issues

3.2.1 Static Mapping and Dynamic Compilation

The output of the static mapping, as illustrated in Figure 3.1, is a complete C++ application that can readily be compiled into the binary code. The only interaction between static mapping and dynamic compilation parts of the tool-chain concerns the selection of the alternative versions of the same function or component that the dynamic compiler will use. Currently, we assume that the dynamic compiler uses all of the versions of the same function that are available. The initial implementation of the function that is to be used is decided by the static mapping part, and the library containing all of the others, with a separate C++ file for each implementation, is sent to the dynamic compiler to create intermediate versions that can later be compiled to their binary forms.

3.2.2 Static Mapping and Dynamic Scheduling

The decisions made by the static mapping infrastructure serve as a starting point for the optimisations performed by the dynamic scheduler. The C++ code that is the result of static mapping provides an initial mapping of the threads to the CPU cores. This part is currently left to the operating system, although in future we envisage making more intelligent decisions with respect to, for example, grouping of the threads and deciding which group of threads should be executed on the "nearby" cores. Static mapping also provides initial mapping of the data to the memory regions. Again, this is either left to the operating system (in the case when simple versions of parallel patterns are used), or we explicitly determine what data is replicated and put on which NUMA memory node (using the `libnuma` library, as described in the deliverable D4.1). In order to allow dynamic scheduling, the C++ code provided by static mapping needs to be modified. The code to create and invoke the `PaRLSched` scheduler (as a separate operating system threads) is required to be put in the application code, and the threads created by the patterned application need to be registered with the scheduler. This part currently needs to be done manually, but in the future we plan to develop refactorings that will do this (semi-)automatically.

3.2.3 Dynamic Scheduling and Performance Monitoring

As we have described in Section 3.1.3, dynamic scheduling makes decisions to remap the threads and data of the application that is being executed, i.e. to pin the threads to the particular CPU cores and data to the particular memory regions/nodes, as a response to the changes in the performance of the application. This requires the scheduler to have information about, for example, the performance of individual cores or, more precisely, execution of threads on these cores. For this purpose, we use the low-level performance monitoring mechanisms that give information at the thread/core level. In particular, we use the PAPI [2] interface for real-time collection of performance counters during the execution of a thread. This requires the application source code to be modified and the appropriate calls to the PAPI routines to be inserted, which are then collected by the scheduler and the remapping decisions are made. This part currently has to be done manually by the programmer, but it is also possible to automate it by refactoring.

3.2.4 Dynamic Compilation and Performance Monitoring

The high-level information from the performance monitoring, such as the number of items in the pipeline queues and the service time of farm workers, can be used by the dynamic compiler to make decisions about recompilation and replacing an existing implementation of a function with the new one. For example, large number of items in the queue of one pipeline stage might indicate that this stage is a potential performance bottleneck that is slowing down the pipeline, which might suggest that we might need to increase parallelism in this stage if possible (by, for example, farming a computation in it if it is not already farmed, or assigning more resources, like CPU cores or GPU devices, to it if it is already farmed). Alternatively, imbalance in the queues of a pipeline might indicate that some stages of the pipeline might need to be merged to reduce overall parallelism. All this can be accomplished by dynamic recompilation *while the application is running*, i.e. replacing the pipeline function with its alternative version that has fewer pipeline stages or where some stages are farmed. While we have still not implemented this functionality in the dynamic compiler, this kind of interaction would, similar to the previous examples, involve modifying the application source code, either manually or by refactoring, to insert the calls to performance monitoring routines, and also an additional mechanism to notify the dynamic compiler that changes need to be made. In future, this part will be handled by the dedicated adaptivity *controller* process that will collect the information from the performance monitor and act as the main point for making decisions about recompilation and adaptation.

Chapter 4

Conclusion

In this deliverable, we have described briefly the adaptivity tool-chain developed in workpackage WP4 and we discussed how different tools of that tool-chain are able to communicate and work together. In most of the cases, the tools are self-contained libraries and are used within the parallel application iteself, by modifying its source code with the calls to the functions of these libraries, e.g. creating an instance of the `PaRLSched` scheduler class and running it. Therefore, interoperability is achieved with modifications of the source application, since the source application acts as a main point of communication between different parts of the tool chain. In other cases, the output of one part of the tool-chain trivially serves as an input to the other. For example, the static mapping infrastructure produces the C++ application that serves naturally as a starting point for inserting performance monitoring and dynamic scheduling capabilities. We have also identified the need for a dedicated, controller logic (possibly a separate operating system process) that will i) collect informations from performance monitoring; ii) make decisions about what kind of adaptation is needed and when; and, iii) send information to the dynamic compiler and/or dynamic scheduler to initiate their actions. In the future, we plan to implement this controller, as well as to automate (using refactoring) instrumentation of the application source code to add monitoring/dynamic scheduling capabilities.

Bibliography

[1] Mammut Library for Real-Time Power Monitoring, https://github.com/DanieleDeSensi/mammut.

[2] Performance Application Programming Interface, http://icl.utk.edu/papi/.

[3] Performance Monitor Library, https://github.com/avr-aics-riken/PMlib.

[4] The LLVM Compiler Infrastructure, llvm.org.