Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Software for the Adaptivity for Patterned Applications on Initial Pattern Set for Heterogeneous Hardware Architectures D4.2

Due date of deliverable: 30.11.2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP4

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)	

Executive Summary

This document is the second deliverable for WP4 “Dynamic Adaptation of Parallel Software”. The purpose of this work package, according to the DoW, is to develop i) techniques for static mapping of components and data of patterned applications to the available hardware; ii) mechanisms for online machine-learning based scheduling for patterned applications; iii) an adaptive, Just-in-Time (JIT) compilation mechanism for patterned applications; and, iv) infrastructure for monitoring performance of patterned applications. The deliverable is the result of the first and second phases of tasks T4.1 (“Static Mapping of Software Components and Data to Hardware Resources”), T4.2 (“Adaptive Compilation of Patterned Applications”), T4.3 (“Dynamic Scheduling of Patterned Applications”) and T4.4 (“Performance Monitoring of Patterned Applications”). In this deliverable, we present the extensions to the basic versions of the tools for static mapping, dynamic scheduling and performance monitoring that were described in D4.1. These extensions deal with heterogeneity in the computing environments (for static mapping and performance monitoring) and allow the dynamic scheduling infrastructure to target multiple types of resources and adaptation strategies.

Contents

Chapter 2

Introduction

In the deliverable D4.1 we described a preliminary version of a set of tools for dynamic adaptation of parallel software. 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 to the underlying resources; and, iv) monitor application behaviour and the underlying hardware environment and detect parallelism bottlenecks. The tools described in D4.1 provided a basic dynamic adaptation infrastructure for homogeneous computing systems and generic parallel patterns, thus addressing a crucial aspect of software engineering for parallel systems.

In this deliverable, we present several extensions to the software described in D4.1. We present a novel adaptive offline mechanism for selecting the type of a component of a parallel application at runtime (Chapter 3). The mechanisms described in D4.1 assumed that the decision about what component implementation to use and on what device to map it is independent on the input data. The selection mechanisms described here use linear interpolation to make selection based on input parameters to the component, thus making it adaptive to the changes in the input data of the application and suitable for data-intensive applications. We also describe the extension to the dynamic scheduler (Chapter 4) which, compared to the version described in Deliverable D4.1, is able to i) target multiple (hierarchically organised) resources at the same time (e.g. processors, cores within processors and memory nodes of a NUMA machine); ii) separate estimation/prediction, allowing us to incorporate alternative prediction mechanisms without imposing any constraints on how these predictions are utilised in the formulation of an optimal strategy; and, iii) incorporates advanced *learning dynamics*, making the frequency of estimations and prediction adaptive to the current performance of an application. Finally, in Chapter 5, we describe how the performance monitoring infrastructure

described in D4.1 can be used for monitoring of applications that combine CPU and GPU code, and we also discuss adaptation of this infrastructure to the generic pattern interface (GrPPI) that was described in D2.4.

Chapter 3

Adaptive Offline Mapping of Parallel Components to the Heterogeneous Hardware

3.1 Introduction

Demand for distinct frameworks and application programming interfaces for different types of processors in heterogeneous computing systems has led to programmers having to write different implementations of the same computation/component for different devices, but without the unified API. Selecting the most suitable device for a given parallel component of an application, and the most appropriate implementation of the component, is a highly non-trivial problem, which usually requires a programmer to analyse both the target platform and the application itself, along with the implementation alternatives and the available libraries. Even then, the correct decision might depend on runtime parameters of the application, e.g. for some input values a CPU implementation might be optimal, while for others a GPU implementation is preferable [9].

An approach to solve this problem is to manually select the algorithm implementation and map the execution onto the underlying parallel device based on past knowledge. Nevertheless, this procedure becomes complex when dealing with multiple devices and libraries. A common technique is to define a set of constraints in order to guide a runtime scheduler to select the most suitable implementation. This technique, however, has non-negligible performance overheads, since it is necessary to check such constraints each time a routine is called. An alternative to the aforementioned technique is to make the selection decision at compile time. Several proposals leveraging this approach and based on analytic models, machine learning and adaptive optimization methods can be found in the literature [4]. However, there are two major drawbacks with this approach: *i)* it is strongly tied to the target platform; and, *ii)* it is limited to a concrete set of devices.

In the deliverable D4.1, we presented a number of mechanisms for deciding on

instantiation of components of a parallel patterned application on heterogeneous (CPU+GPU) systems. The mechanisms there were able to decide the amount and type (CPU or GPU) of each worker in the parallel pattern. The decisions made there were, however, not driven by the input data of the application, i.e. they assume that the selection of the best type of component for a given computation does not depend on any runtime information. In this deliverable, we present an extension to these mechanisms that are able to express constraints on the input data and decide, at runtime and based on profiling data, which version of a component to use. Specifically, we make the following contributions:

- we present an adaptive offline profiling-based mechanism for mapping application parallel components to heterogeneous hardware;
- we exploit two novel features of the C++ standard, concepts and attributes;
- we evaluate the performance of the selector by analyzing its convergence time and the impact on the application execution time, using a general matrix-matrix multiplication kernel and an image-processing application.
- we demonstrate hardware-portability of the framework, showing how it supports different types and number of devices, and how it can be extended to other types of restrictions.

The remainder of the chapter is structured as follows. Section 3.2 describes the concepts and attributes C++ language features along with the hardware parallel platform description language. Section 3.3 introduces our adaptive offline mapping mechanism and its decision algorithm. Section 3.4 evaluates the convergence time and performance benefits of the mapping mechanism. Finally, Section 3.5 closes the chapter with some concluding remarks and future works.

3.2 Background

This section gives a brief overview of the two C++ language features used for developing the implementation of the mapping interface: C++ attributes and concepts. Furthermore, we describe the hardware parallel platform description language leveraged by the mapping framework to keep track of available resources.

3.2.1 The C++11 attributes

Attributes are a new feature of the C++11 language for defining properties of programming entity units. The power of the attributes is that they provide extra information to the compiler in order to perform a given action on the entity that has been annotated. One of the main advantages of the attributes, with respect to pragmas, is their flexibility, as they can provide properties to any entity, e.g. variables, functions or types, and do not need to appear on a separate line [5]. The basic syntax for attributes in C++11 is defined with: `[[namespace::attribute]]`.

However, unlike attributes from other languages, C++11 attributes are compiler-defined, i.e., the user cannot define their own attributes at runtime. Indeed, the C++ Run-Time Type Identification (RTTI) mechanism does not keep any attribute information, so the knowledge given in the attributes is not accessible from the user application. Since the attribute extensions require modifications in the C++ compiler, the purpose of this feature is to allow future C++ extensions without extending the set of keywords or grammar.

3.2.2 The C++ concepts

Concepts are a novel extension of the C++ programming language that allows defining and evaluating, at compile time, constraints set on template arguments [10]. Particularly, concepts deliver a better support for error checking in generic programming contexts and thus they can be seen as a mechanism to diagnose and prevent improper uses of templates. Lexically, concepts can be expressed with the keywords `concept` and `requires`. Their current implementation can be found in the GCC C++ experimental branch, namely *concepts lite* [11].

Listing 3.1 shows an example of using concepts to overload the `mult` function implementing the matrix-vector and matrix-matrix multiplications. In this case, two concepts have been declared to check whether the type `T` is a vector or a matrix. Thus, when `mult` is called passing matrices, the compiler automatically selects the first version, since the requirements are met for the `Mat` concept. On the contrary, if it is called passing a vector and a matrix, the second implementation is taken. Finally, if none of the requirements are met, the compiler warns the user with a message of the concept violated.

Other kind of restrictions in order to limit the use of a template can be set by declaring constant expressions. For example, Listing 3.2 declares `r1` and `r2`, two constant expressions that return a boolean depending on the value received as argument. Using the `require` clauses along with these expressions, it is possible to overload the function `foo` depending on the argument passed. As can be seen, if the argument is less than 100, the first implementation is compiled; otherwise the second is taken instead.

Listing 3.1: Example of overloaded functions using concepts.

```
template <typename T>
concept bool Mat() { return requires(T t) {...} }

template <typename T>
concept bool Vec() { return requires(T t) {...} }

template <Mat M>
T mult(M a, M b) { /* a, b: matrices */ }

template <Vec V, Mat M>
V mult(V v, M m) { /* v: vector; m: matrix */ }

int main(){
  double vec [] = ...;
  double mat [][] = ...;
  auto resMat = mult(mat, mat); // gemm
  auto resVec = mult(vec, mat); // spmv
}
```

Listing 3.2: Example of overloaded template using non-type requirements.

```
constexpr bool r1(int n)
{ return n < 100;  }

constexpr bool r2(int n)
{ return n >= 100; }

template <int val> requires r1(val)
void foo()
{ ... some compute ... }

template <int val> requires r2(val)
void foo()
{ ... some compute ... }

int main() {
  foo<90>();  // Uses the first implementation
  foo<200>(); // Uses the second implementation
}
```

To sum it up, concepts promote *generic programming* by means of overloading templates and disabling the ones which types do not satisfy the predefined constraints. While this feature is similar to the `enable_if` C++11 meta-function, the syntax stated for the concepts becomes much simpler and general.

3.2.3 The hardware parallel platform description language

The Hardware Parallel Platform Description Language (HPP-DL) is an open specification for describing features of Heterogeneous Parallel Platforms (HPP) and leveraging hierarchical models [7]. The HPP language is intended to be used for making platform-specific information available to developers and tools, such as auto-tuners, compilers, schedulers or runtime systems. This specification comprises information about the following classes: *i) components* describe the hardware resources of the platform, involving processors (CPU sockets), cores, main memory, GPUs or OpenCL based devices; *ii) links* represent relationships between two *components* in one way and incorporate information about data transmission: throughput and latency; and *iii) resources* portray interfaces for accessing to computing devices attached, e.g. FPGAs or DSPs. These resources comprise I/O ports, IRQs or address ranges.

3.3 Adaptive Offline Mapping Mechanism

In this section we describe an adaptive offline implementation of the mapping framework. Figure 3.1 shows the general workflow of this framework. In the first step, the system administrator provides users with all the possible implementations of a function by producing, using the header generation tool, necessary function headers for the two proposed interfaces, i.e., attributes and concepts. This command-line tool takes as inputs the function name and available implementations along with the devices supported for each of them. Afterwards, the function headers in the generated files should be completed by the administrator with the corresponding code for calling to specific implementations for a given algorithm. Note that all generated headers are automatically instrumented for measur-

ing execution time. This is done in order to support the profile-guided approach implemented by the selector. In parallel, the administrator should also obtain required hardware information, using the automatic tool provided for storing it into a `HPP.json` file, according to the HPP-DL specification.

On the other hand, users should call the interfaces from the generated header files, in order to provide static information to the selector in the form of attributes or template arguments. Next, the attribute and concepts-based implementation of the mapping mechanism calls the selector module to determine the most suitable implementation using the static information provided by the user and performance data from previous executions. Finally, once the application has been compiled and executed, the performance file (`PERF.h`) is updated accordingly with the new performance data in order to improve the selector knowledge in future decisions.

In the following sections, we describe how the concepts and the attribute mechanisms should be used within the framework. After that, we describe the algorithm to actually select the most suitable implementation used by the selector module.

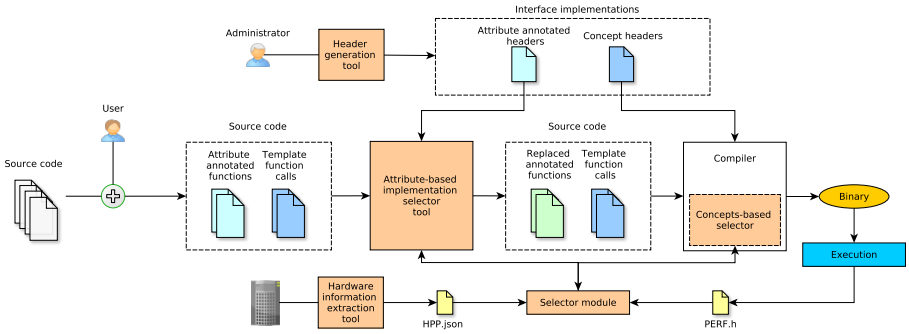


Figure 3.1: Diagram of the adaptive offline implementation selector.

3.3.1 The attributes-based interface

The interface of the framework based on C++ attributes is intended to define constraints in order to guide the compiler to select a concrete implementation between the available ones. These attributes are part of our previous work presented in [8]. First, the administrator needs to run the header generation tool, with the *attributes* flag option set, so as to generate the headers containing the annotated function prototypes for a given algorithm. Next, the administrator should complete these attribute-annotated prototypes with the corresponding function calls for the interfaces generated. The attributes inserted, under the `ais` (adaptive implementation selector) *namespace*, are the following:

- `ais::implements`: This attribute specifies that the code under the attribute is an alternate implementation of a given interface. Basically, it receives, as the sole parameter, the function name to let the selector know which implementations are available for that interface.

- `ais::device`: This attribute bounds a given implementation to a specific target device. Examples of valid parameters for this attribute are: `CPU`, `GPU`, `PHI` (for the Intel Xeon Phi), etc.

Afterwards, the users are responsible for annotating candidate function calls in the application code in order to be analyzed by the attribute-based implementation selector tool of the framework. These C++ attributes are the following:

- `ais::interface`: This attribute indicates that the function call annotated is an interface, should be treated by the tool and replaced during the selection process.

- `ais::target`: It defines the preferred target device to execute the annotated interface, e.g., `ais::target(CPU)`. Valid arguments for this attribute are those accepted by the `ais::device` attribute.

- `ais::size`: This attribute is used if the user knows a priori the problem size during the function call. This attribute receives an integer as a single parameter.

- `ais::min` and `ais::max`: Alternatively, if the user is not able to know the problem size, these attributes can be used to let the selector know the lower and upper bounds of the problem size used to execute a certain function.

For instance, Listing 3.3 contains an example of user code with different C++ attribute-annotated function calls matching the interface `dgemm` defined in the header file. Note that both first and second calls to `dgemm` have been annotated using the attributes for a fixed size and a range of sizes, respectively. After preprocessing the annotated source code, the annotated interfaces are automatically replaced by the most suitable function implementations. As can be seen in Listing 3.4, both `dgemm` calls have been replaced by function calls to the clBLAS and Intel MKL highly-tuned kernel implementations, respectively.

Listing 3.3: Attribute-annotated function interfaces.

```
[[ais::interface, ais::size(1024)]]
dgemm( ... );
[[ais::interface, ais::min(256), ais::max(512)]]
dgemm( ... );
```

Listing 3.4: Replaced attributes by implementations.

```
// Replaced annotation by clBLAS-CPU dgemm
dgemm_clBLAS_CPU( ... );
// Replaced annotation by Intel MKL dgemm
dgemm_MKL( ... );
```

3.3.2 The concepts-based interface

The interface of the framework based on concepts is an alternative to the aforementioned mechanism based on attributes that pursues the same goal. In this case, the system administrator should use the header generation tool to introduce an algorithm implementation and the devices supported with the *concepts* flag option set. This will generate a header file containing the function templates with the concepts in charge of determining the most suitable implementation for a given input size or range of sizes. Next, the function templates in the header file should be completed with the corresponding function calls to specific implementations for such an algorithm.

Afterwards, the users should call the supported function interfaces providing the necessary information, in the form of template arguments, so as to guide the compiler to link against concrete implementations. As an example, the first call to function `dgemm` in Listing 3.5 provides a fixed problem size of the matrices, while the second provides a range when the sizes used in the multiplication are not known in advance. This static information allows the compiler to link against the `dgemm` implementation complying with the concepts introduced by the header generation tool and the requirements defined by the user.

Listing 3.5: Concepts mechanism used in the `dgemm` function interface.

```
dgemm</*size*/ 1024>( ... );            // This call will be linked
   (cont.)against clBLAS-CPU dgemm
dgemm</*min*/ 256, /*max*/ 512>( ... ); // This call will be linked
   (cont.)against Intel MKL dgemm
```

3.3.3 The selector module

This section describes the internal algorithm of the adaptive implementation selector module. This algorithm has been implemented for accepting the two aforementioned interfaces: attributes and concepts. Note that the attributes-based implementation selector has been developed as a preprocessing tool using the Clang 3.8.0 compiler API, while the concepts-based selector is implicitly embedded into the semantic concepts code and interpreted by the compiler.

The selection algorithm implemented by our framework is entirely based on the problem size and boundaries specified by the user. Depending on the information provided by the user, the algorithm proceeds as follows:

- If the template argument or attribute for a fixed size is set, the selector takes the implementation offering the minimum execution time. This is done using the information stored in the `PERF.h`. In order to obtain the implementation delivering the best performance for a fixed problem size, the selector performs a linear interpolation for the requested problem size for all implementations available in such a function interface, in case it is not present in the performance file. Otherwise, if multiple implementations deliver the same

minimum performance, the selector randomly picks one of them. Consider the scenario in Figure 3.2a showing the behavior of the `dgemm` interface offering six different implementations. For instance, if the user sets the size parameter to 1,024, the selector will consider the clBLAS version running on CPU, while if the problem size is fixed to 504, the selector will randomly select GSL, MKL or clBLAS (Xeon). However this random policy can be eventually replaced by another that takes into account the lower maximum performance in order to avoid extreme behaviors.

- On the contrary, if the developer has indicated a range of possible problem sizes, the selector module computes the area under the performance curve (or integral) between the ranges for each implementations available in a function interface. Then, the selector takes the implementation that has the smallest area between the ranges. As shown in Figure 3.2b, if the user selects a range between 256 and 512 as for the minimum and maximum problem sizes, the selector module will compute the integrals for the six implementations available. Afterwards, it will compare the areas below the curves and take that having the smallest area, i.e., MKL. Note that if there are no performance values in the boundaries of the range, the values that intersect the boundaries are computed via linear interpolation. As in the previous option using a fixed size, if there are two or more implementations whose integral value is equal, the selector will pick one randomly.

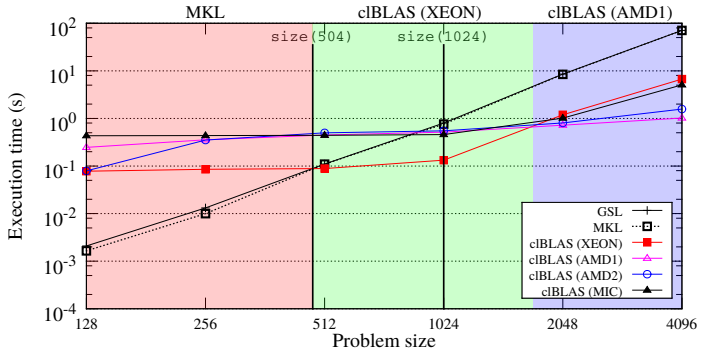
Note that, at present, the selection algorithm only considers the problem size to select the fastest implementation. In the future, we plan to extend the set of user template arguments and attributes to allow users to specify other kinds of restrictions, such as memory usage or energy consumption. This will allow the selector to make multi-objective optimizations.

3.4 Experimental evaluation

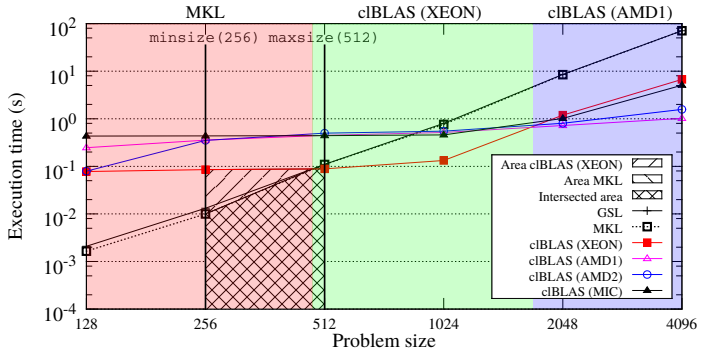
In this section, we evaluate the behavior of our adaptive offline mapping framework along with its selection algorithm using the dense matrix-matrix multiplication (GEMM) and an image-processing application targeted to embedded systems (StereoBM). First, we perform an evaluation of the accuracy and convergence of the selector algorithm of the framework. Finally, we study the adaptability to changes of the selection framework in heterogeneous platforms.

We assess the framework using a platform equipped with an Intel Xeon processor, two AMD Radeon GPUs (connected via PCI Express 3.0) and an Intel Xeon Phi co-processor. Table 3.1 describes the details of these components. As additional information, this platform runs a Linux Ubuntu 14.04 as for the OS with the GCC v5.0 compiler with *concepts-lite* extension enabled.

To carry out the experiments in this section, we leveraged both GEMM and StereoBM uses cases. For the GEMM case, we employ the CPU implementa-



(a) Scenario using fixed sizes.



(b) Scenario using a range of sizes.

Figure 3.2: Execution time of the `dgemm` operation for different problem sizes and implementations. Note the region boundaries set by the selector module.

tions from the GSL and Intel MKL libraries, while the clBLAS implementation is intended to run on all available devices. Figure 3.2, from the previous section, shows the execution times for these implementations using square matrix sizes ranging from 128 to 4,096. On the other hand, for the STEREOBM use case, we use the sequential and Intel TBB framework using OpenCV routines versions for CPU and accelerators.

3.4.1 Evaluation of the accuracy

In this section we evaluate the convergence and accuracy of the implementation selector algorithm. Figure 3.3 evaluates the convergence of the decisions made by the selector using fixed sizes and ranges of sizes through different number of training iterations for both use cases. For each iteration, we execute an instance of the `dgemm` kernel and the STEREOBM application with random matrix sizes and image resolutions, respectively, in order to train the system. To evaluate the accuracy of each training iteration we perform 100 runs using random problem sizes and compute the average hit rate. Obviously, calls to the `dgemm` kernel and OpenCV

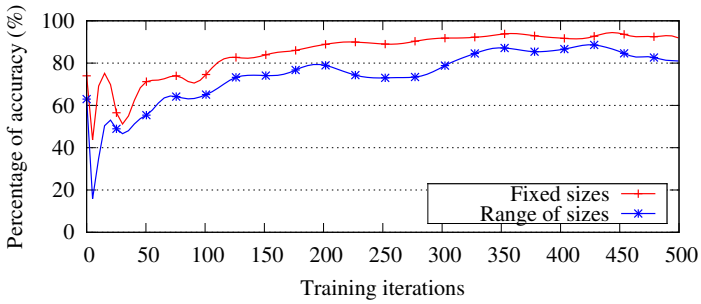
Table 3.1: Heterogeneous testbed platform.

	XEON	AMD1	AMD2	MIC
Model	Intel Xeon® CPU E5-2695	AMD Radeon™ R9 290X series	AMD Radeon™ R9 285 series	Intel Xeon Phi™ 3120 series
Core clock	1.2 GHz	1030 MHz	928 MHz	1.1 GHz
Computing units	24	44 (2816)	28 (1792)	224
Memory	128 GB	4 GB	2 GB	6 GB
OpenCL version	AMD-APP OpenCL 2.0 (1642.5)			Intel OpenCL™ Runtime 14.2

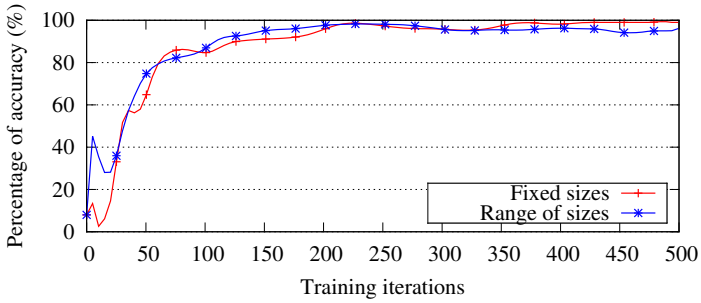
routines in the STEREOBM application have been provided with information about the problem size or range of sizes, and processed prior the compilation phase. Note as well that the annotation mechanisms (attributes and concepts) are independent to the selection algorithm behavior, as they are just interfaces supported by the framework. As can be seen in Figure 3.3a, the learning progress for the GEMM case using both fixed and range of sizes increases in a smooth curve until reaching, after 500 training iterations, 92 % and 81 % of the total accuracy, respectively. On the contrary, a random selection would have only selected the best implementation once every 6 selections, i.e., roughly a 16 % of hit rate, among the 6 possible combinations of implementations and devices. Therefore, comparing our framework with the random selection, our selector already takes accurate decisions after the 50^{th} iteration. Similarly, the learning progress for the STEREOBM application, depicted in Figure 3.3b, increases more rapidly than for the GEMM case, as it has less combinations of device-implementation available. This behavior finally leads to higher accuracy figures: 99 % and 93 % for fixed and ranges of sizes after 500 iterations, respectively. In general, we observe that the accuracy for fixed problem sizes is slightly higher than for range of sizes. This is due to the size parameter provided by the user is a more accurate indicator than freely selecting a size between a given ranges.

3.4.2 Evaluation of the adaptability

Next, we evaluate the adaptability the selector to make appropriate decisions each time a new device (and implementation) is attached to the heterogeneous platform after 100 training iterations. Figure 3.4 analyzes the quality of the selections in this scenario for both GEMM and STEREOBM use cases with fixed and range of sizes. Focusing on the GEMM case, the selector reaches about 98 % of accuracy after 100 iterations. Each time a new device-implementation is included in the platform we observe that the selector needs about 20 extra iterations to converge once again and reach similar accuracy figures than before the change. Looking at the STEREOBM application, we notice that each time we add a new version, the knowledge considerably drops compared to the GEMM case, however the selector needs approximately 25 iterations to stabilize once again. Certainly, using ranges of sizes instead of fixed sizes limits the information provided to the selector, thus



(a) Accuracy for GEMM.



(b) Accuracy for STEREOBM.

Figure 3.3: Progress of the accuracy of the selection algorithm for the GEMM and STEREOBM.

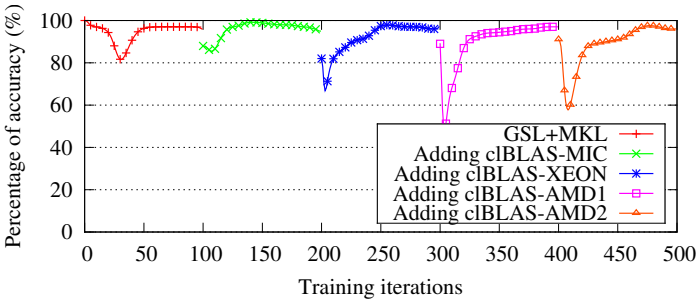
leading the selector to generate smoother learning curves and provide less accuracy. As a final remark, we observe that the accuracy reached by the selector when new devices are steadily attached to the platform is slightly higher than if all devices are attached right at the beginning of the training process. This is given because the selector needs to train less if only one extra device is attached each time.

3.5 Conclusions

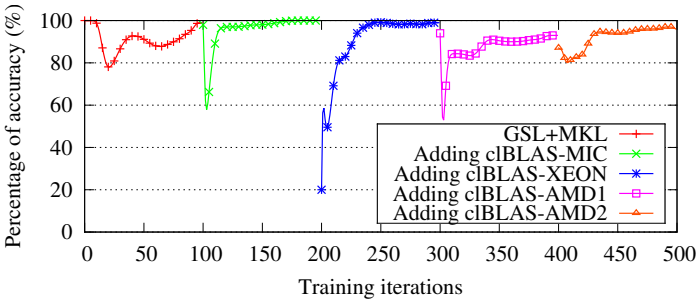
In this work we have presented an adaptive offline implementation selector for heterogeneous platform that *i)* provides two interfaces based on C++ attributes and concepts for providing multiple implementations offering the same functionality; and *ii)* selects automatically the tuple device-implementation that delivers the best performance according to the problem size using a static profile-guided optimization approach. Thanks to this approach, our framework shifts the decision-making process at compile time, so that overheads related to dynamic scheduling approaches are avoided. Furthermore, our framework is hardware independent, therefore, it is possible to freely add or remove devices of the platform without incurring significant overheads. Different to other approaches, our implementation selector leverages two novel C++ features (attributes and concepts), inherent to the standard C++ programming language, making it more portable and extensible.

To evaluate the benefits of this framework, we analyzed the global performance and convergence of the tool using two different use cases: the general matrix-matrix multiplication and an image-processing application. The experimental results demonstrate that the selector enhances performance while minimizes efforts to tune applications targeted to heterogeneous platforms. Furthermore, the profile-guided optimization approach leveraged requires only a few training iterations to guarantee that the implementations taken offer the maximum performance.

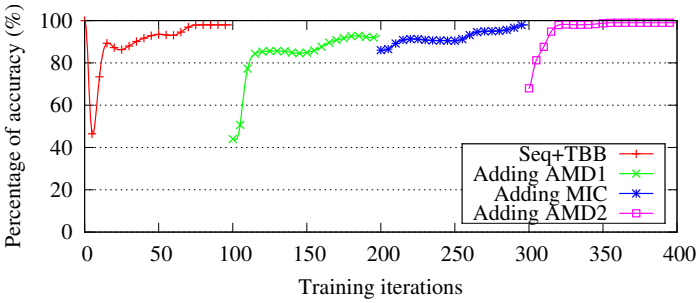
As future work, we plan to extend the set of C++ attributes in order to allow users specify other kinds of restrictions, such as memory usage or energy consumption. Furthermore, we also at incorporating a static partitioning module for supporting multiple devices in shared and distributed memory systems. Another goal is to incorporate the attribute-based selector tool as part of the Clang C++ compiler.



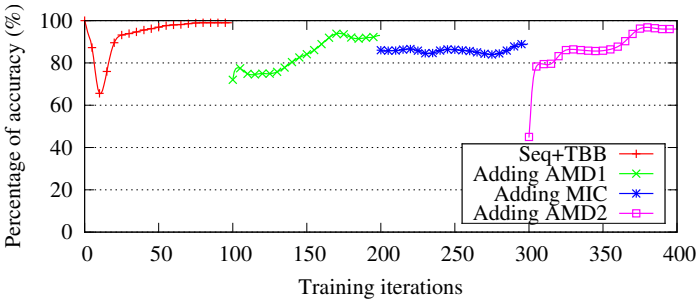
(a) Accuracy using fixed sizes for GEMM.



(b) Accuracy using ranges of sizes for GEMM.



(c) Accuracy using fixed sizes for StereoBM.



(d) Accuracy using ranges of sizes for StereoBM.

Figure 3.4: Progress of the accuracy of the selection algorithm for the GEMM and StereoBM when adding devices.

Chapter 4

Dynamic Scheduling of Patterned Application

4.1 Introduction

In this chapter, we describe the extension of our work conducted under T4.3 (D4.1), the primary goal of which is the design and implementation of a *Dynamic Scheduler* tailored specifically for Patterned (Parallelised) Applications.

The main goal of the Dynamic Scheduler is to exploit appropriately the available hardware resources (i.e., processing bandwidth and memory), so that it increases the *performance* (in a way to be defined) of a parallelised application during run-time. Moreover, the Dynamic Scheduler should also increase the robustness and resilience of the parallelised application, since a) the application should meet high performance standards relatively to the provided hardware configurations and b) other applications might share the same resources, which may lead to unpredictable variations in the performance of the running applications.

We are working towards the design of a fully *automatic* Dynamic Scheduler, which will be able to *discover* and *predict* optimal allocations of resources independently of the nature of the applications (advanced parallel patterns, data-intensive applications, etc.) and potential (dynamic) changes in the environments (e.g., availability of resources). As also described in deliverable D4.1, the main challenges involved in the design of the Dynamic Scheduler can be summarized as follows:

(G1) *"Optimality" through performance measurements.* A Dynamic Scheduler should be able to discover an optimal allocation of resources without assuming any knowledge of the application's details. Such optimisation may only be based on the performance of the application under prior resource assignments (i.e., *prior experience*).

(G2) *Fast response to changes in the availability of hardware resources*. When other applications also run on the same platform, the Dynamic Scheduler

should be able to adapt fast to the new situation and discover a new optimal resource allocation.

(G3) *Minimal computational complexity.* A Dynamic Scheduler should only assume minimum information available from the parallelised application, while the involved computations should also be efficient. The Dynamic Scheduler will also run on the same platform and therefore the required overhead (processing bandwidth and memory) should be as minimal as possible.

4.1.1 D4.1 (recap & open questions)

In D4.1, the Dynamic Scheduler addressed the problem of automatically and dynamically discovering the optimal placement of threads (pinning) into a homogeneous hardware platform (in fact, a set of identical CPU processing units). It was based on a distributed Reinforcement-Learning algorithm that learns the optimal pinning of threads into the set of available CPU cores based *solely* on the performance measurements of each thread. The proposed methodology emphasized the fact that the performance of a parallelised application can increase significantly under a) dynamic changes in the availability of resources and b) dynamic changes in the application's demand. These two points seem to be the two major weaknesses of modern operating systems, i.e., the ability to re-adjust under dynamic changes.

The algorithm presented and analysed in D4.1 addressed goals (G1)–(G3) above. It uses a Reinforcement-Learning scheme (developed in [2]), which formulates strategies for each one of the threads separately. It exhibits linear complexity with the number of threads, while it requires keeping in memory only a single strategy vector (of size equal to the number of CPU cores). Finally, it reacts relatively fast to changes in the availability of resources (i.e., when other applications start running on the same platforms) as demonstrated in the experiments of the dynamic scheduling section of D4.1.

We would like to emphasize though that the methodology proposed in D4.1 could improve with respect to the following aspects:

- *Multiple resources.* The proposed approach in D4.1 was concerned with the optimization of a *single* resource (i.e., the processing bandwidth through the allocation of CPU cores to threads). The question that naturally emerges is the following: *How the proposed methodology can be modified to accommodate multiple and possibly non-uniform resources?* For example, in NUMA architectures we may be concerned of both processing bandwidth as well as memory placement.

- *Hierarchical structure.* Resources in NUMA architectures may involve hierarchical structures. For example, placement of a thread into a CPU core constitutes a fine-grained allocation of the processing bandwidth. Allocation may instead be performed into NUMA nodes, which can be thought of as a higher-level allocation of processing bandwidth.

- *Non-uniform constraints/requirements.* Multiplicity in the number and nature of optimized resources may additionally impose non-uniform constraints and/or requirements. For example, switching the placement of a thread to a different CPU core may be performed more often compared to, for example, switching the placement of its memory pages among different NUMA nodes. Such differences in the constraints of placing non-uniform resources require special treatment and the algorithms provided should be able to accommodate alternative criteria.

- *Estimation & Optimization.* The proposed approach in D4.1 provided a unified methodology for concurrent *estimation* and *optimization*. In particular, *estimation* was performed through the reinforcement-learning updates of a strategy/probability vector that summarizes prior experience of a thread over the most beneficial allocation. Moreover, *optimization* was achieved by randomly selecting the destination of a thread according to the corresponding probability vector.

  However, it might be desirable that estimation and optimization are separated from each other, in order for us to be able to incorporate alternative approaches (either for estimation or for optimization). Furthermore, alternative estimation methods might be available at the same time and the role of the optimizer should be to optimally integrate their predictions.

4.1.2 D4.2 (objective & new features)

In this deliverable (D4.2), we provide an extension of the Dynamic Scheduler developed under D4.1 in two main directions (in accordance to the tasks described in T4.3):

(F1) ***Advancement of architecture.*** We wish to provide a Dynamic Scheduler that may easily accommodate more than a single resource at the same time (e.g., both processing bandwidth and memory). However, resources may not necessarily be uniform in nature, optimization criteria and constraints, while they may be organized in hierarchical structures. To this end, we introduced a rather abstract structure in the dynamic scheduler, which is characterized by the following features:

  (a) *Multiple resources.* The user may define alternative resources to be optimized (i.e., processing bandwidth in the form of thread placement, and cache-memory allocation).

  (b) *Hierarchical structure.* The resources may accept *child* resources, a term introduced to establish hierarchical dependencies between the resources. For example, thread placement may be performed with respect to NUMA nodes, however therein a subsequent placement may also be performed with respect to a CPU core.

(c) *Distinct optimization criteria.* Each one of the optimized resources and/or their child resources, may accept a distinct method for estimation and optimization, as well as a distinct optimization criterion.

(F2) ***Separating estimation from optimization.*** We advanced our framework for generating strategies for threads by separating the role of *estimation/prediction* from the role of the *optimization*. The reason for this distinction comes from the need to incorporate alternative prediction schemes over optimal allocations without necessarily imposing any constraint in the way these predictions are utilized in the formulation of an optimal strategy.

(F3) ***Advancement of learning dynamics.*** When optimizing memory placement in run-time, we wish to minimize the number of placement switches necessary for approaching an optimal allocation. At the same time, we wish to increase the reaction speed to rapid variations in the performance. To this end, we introduced a novel learning dynamics that is based in the formulation of benchmark performances/actions. This class of dynamics closely follows the evolution of the performance and triggers the appropriate responses (e.g., experimentation).

It is important to note that the above framework is *independent of the specifics of the underlying hardware structure as well as the details of the application itself*. Thus, in practice there is no limitation imposed from possibly varying pattern implementation.

4.2 Dynamic Scheduler

Below we provide a detailed description of the new features of the updated Dynamic Scheduler. Figure 4.1 also provides a general picture over the main parts of the scheduler.

4.2.1 Advancement of architecture

As briefly discussed in Section 4.1.2, the main goal of the updated architecture is to provide a straightforward integration of (a) *multiple resources*, (b) *hierarchical structure of resources*, and (c) *alternative optimization criteria*.

According to the new architecture, the user may define the resources to be optimized as well as the corresponding methods used for establishing predictions and for computing optimal allocations. In particular, the initialization of the scheduler accepts the parameters depicted in the following table.

```
RESOURCES={"NUMA_PROCESSING","NUMA_MEMORY"}
OPT_CRITERIA={"PROCESSING_SPEED","PROCESSING_SPEED"}
RESOURCES_EST_METHODS={"RL","RL"}
RESOURCES_OPT_METHODS={"RL","AL"}
```

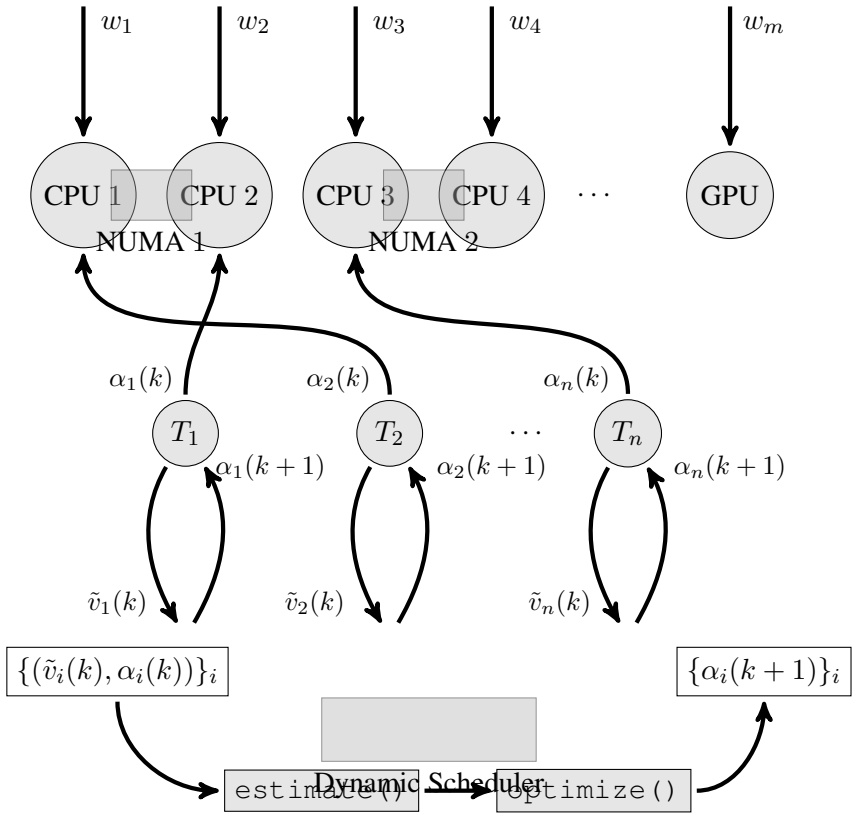


Figure 4.1: Schematic of the *dynamic* resource allocation framework.

In the above example, we have defined two distinct resources to be optimized, namely "`NUMA_PROCESSING`", which refers to the placement of threads to specific NUMA nodes, and "`NUMA_MEMORY`", which refers to the placement/binding of the thread's memory pages into specific NUMA nodes. For each one of the resources to be optimized, there might be alternative optimization criteria, which summarize our objectives for guiding the placements. The selection of the optimization criteria is open-ended and directly depends upon the available performance metrics. Currently, we evaluate allocations based on their impact in the *average processing speed* over all running threads of the parallelized application.

In parallel to the selection of the optimized resources, we need to also define the corresponding "methods" for establishing predictions (which are used under the `estimate()` part of the Dynamic Scheduler, Figure 4.1). For example, we may use the Reinforcement-Learning (RL) algorithm developed and implemented in D4.1 to formulate predictions based on prior performances. The outcome of the `estimate()` part of the scheduler, which a probability or strategy vector over the available placement choices, represents a prediction over the most beneficial placement.

Similarly, we need to define the corresponding "methods" for the computation of the next placements (which are used under the `optimize()` part of the Dynamic Scheduler, Figure 4.1). For example, we may use again the Reinforcement-Learning (RL) selection criterion which is based upon the strategy vectors developed in the `estimate()`. For the case of "`NUMA_MEMORY`", we may use an alternative optimization method, *aspiration learning* (AL), briefly described in the forthcoming Section 4.2.4.

4.2.2 Hierarchical structure

Apart from the ability to optimize over more than one resources, it might be the case (as evident in NUMA architectures) that placement of resources can be specialized over several levels. For example, a thread may be bound for processing into one of the available NUMA nodes, however placement can further be specialized over the underlying cores of this node. Thus, the nested hardware architecture naturally imposes a nested description of the resource assignment. That is, the decision $\alpha_1(k)$ of thread T_1 in Figure 4.1 may consist of two levels: in the first "0" level, the NUMA node is selected, while in the second "1" level, the CPU-core of this NUMA node is selected.

The Dynamic Scheduler has been redesigned so that it accepts a nested description of resources. The depth of such description is not limited, although in the current implementation we have been experimenting with a single type of a child resource. An example of how child resources can be defined by the user is depicted in the following table.

```
CHILD_RESOURCES={"CPU_PROCESSING","NULL"}
CHILD_OPT_CRITERIA={"PROCESSING_SPEED","
   (cont.)PROCESSING_SPEED"}
CHILD_RESOURCES_EST_METHODS={"RL","RL"}
CHILD_RESOURCES_OPT_METHODS={"AL","AL"}
```

We have defined a child resource for the NUMA processing resources, which corresponds to a CPU-based description of the placement. On the other hand, for the NUMA memory resources, we have not defined any child resources. For each one of the child resources, we may define separate estimation and optimization methods. Note that the decisions and algorithms over child resources may not coincide with the corresponding methods applied for the case of the original resources. In other words, decisions over original and child resources are completely separated from each other.

4.2.3 Separating estimation from optimization

An additional feature of the updated Dynamic Scheduler is the separation between *estimation/prediction* and *optimization*. These two distinct functionalities of the scheduler are depicted in Figure 4.1. The reason for this separation in the scheduler

was the need for incorporating alternative methodologies for the establishment of both predictions and optimal decisions.

For example, in the learning dynamics (Reinforcement Learning) implemented for optimizing the overall processing speed of a parallelized application in D4.1, we have implicitly incorporated the functionalities of estimation and optimization within a single learning procedure. Recall that the first part of the Reinforcement Learning methodology is devoted in updating the strategy vectors for each one of the threads (which summarize our predictions over the most beneficial placement), while the second part is devoted in randomly selecting a decision based on a slightly perturbed strategy vector. Under the updated architecture of the scheduler, these two functionalities (estimation and optimization) are separated.

The only constraint that we impose in the design of the estimation/prediction scheme is that its outcome should always be provided in the form of a strategy/probability vector.

4.2.4 Advancement of learning dynamics

As briefly described in Section 4.1.2, we wish to provide a class of learning dynamics which provides a better control over two important features: a) speed of response to rapid performance variations, and b) experimentation rate.

The Reinforcement Learning dynamics presented in D4.1 and also described in [3], provide a class of learning dynamics that requires an experimentation phase controlled through a rather small perturbation factor $\lambda > 0$. Essentially, this factor represents a very small probability that a thread will not be placed according to the formulated estimates, rather it will be placed according to a uniform distribution. Such perturbation is essential for establishing a search path towards the current best placement.

However, under this learning dynamics, the times at which the experimentation phase occurs are independent from the current performance of a thread. Thus, situations may occur at which a) experimentation occurs frequently at allocations at which rather high performance is currently observed (something that would not be desirable), and b) experimentation may delay when needed the most (i.e., when performance is dropping). In such cases, it would have been desirable to also exploit the available performance metrics.

To this end, we developed a novel learning scheme that is based upon the notions of benchmark actions/performances and bears similarities with the so-called *aspiration learning*. In words, the basic steps of this learning scheme can be summarized as follows: Let k denote the time index of the optimizer update (it may or may not coincide with the update index of the scheduler).

1. **Performance update.** At time k, update the (discounted) running average performance of the thread (with respect to the optimized resource). Let us denote this average performance by $\bar{v}_i(k)$.

2. **Benchmark update.** Define the *upper benchmark performance* $\bar{b}_i(k)$ as follows:

$$\bar{b}_i(k) = \begin{cases} \bar{v}_i(k), & \text{if } \bar{v}_i(k) > \bar{b}_i(k-1) \\ \bar{b}_i(k-1), & \text{else.} \end{cases} \tag{4.1}$$

and the low benchmark performance $\underline{b}_i(k)$ as follows: $\underline{b}_i(k+1) = \bar{b}_i(k+1) \cdot (1-\Delta)$, where Δ expresses a security interval from the maximum benchmark that is subject to design.

3. **Action update.** Given the current benchmark and performance, a thread i selects actions according to the following rule:

   (a) if $\bar{v}_i(k) < \underline{b}_i(k)$, i.e., if the current average performance is larger than the low benchmark performance, then thread i will perform a random switch to an alternative selection with large probability.

   (b) if $\bar{v}_i(k) \geq \underline{b}_i(k)$, i.e., if the current average performance is larger than the low benchmark performance, then thread i will keep playing the same action with high probability and experiment with a very small probability.

It is important to note that the above learning scheme will react immediately when a rapid decrease is observed in the performance (thus, we indirectly increase the response time to large performance variations). At the same time, the small probability of experimentation is necessary under situations of rather constant performance in order to explore a more beneficial allocation. However, we may direct the search of the experimentation towards allocations which we believe will provide a better outcome. This can be done by directly incorporating the outcome of an estimation method directly in step (3a) of the learning scheme. Thus, such learning scheme can easily be incorporated in the updated architecture of Figure 4.1 and make use of the outcome of any estimation method.

4.3 Dynamic Scheduler Library (`PaRLSched_2.0`)

In this section, we provide a description of version 2.0 of the Dynamic Scheduler Library (briefly, PaRLSched, which stands for *Reinforcement-Learning-based Scheduler for Parallelised applications*).

4.3.1 Description

The PaRLSched library is currently developed for Linux platforms and utilises the `sched.h` library for CPU-affinity bindings, the PAPI interface [6] for real-time collection of performance counters during the execution time of a thread and the `libnuma` for memory bindings. Currently, the library has only been used and tested when threads are defined using the C++ POSIX thread library, because it provides

enough freedom to incorporate the PAPI interface for capturing performance measurements. However, the scheduler is independent of the way threads have been created.

The PaRLSched library consists of the following header files:

- `ThreadInfo.h` captures general properties of each thread (e.g., thread ID) as well as running-time performance measurements during the last evaluation interval of the scheduler (e.g., total instructions completed).
- `ThreadControl.h` defines a generic class that includes the functions responsible for initialising and recording the performance counters for each one of the threads.
- `Scheduler.h` incorporates the core of the PaRLSched scheduler as depicted in Figure 4.1.
- `MethodsEstimate.h` provides the necessary definitions and methods for performing the estimation/prediction over the most beneficial placement of resources.
- `MethodsOptimize.h` provides the necessary definitions and methods for performing the optimal placements of resources.
- `MethodsActions.h` provides the necessary definitions for describing hierarchical placement of resources. This is particularly relevant when child resources are present that naturally imposes a nested description of placements.
- `MethodsPerformanceMonitoring.h` provides the necessary definitions for pre-processing performance metrics before their use in the learning schemes. Such pre-processing may involve, for example, the computation of (discounted) running -average performance, or the computation of an average performance among all running threads.

4.3.2 Provided software

The PaRLSched (ver 2.0) library can be cloned or checked out from the project Git repository using the following link:

```
https://github.com/uoscompsci/RePhrase.git
```

The PaRLSched (ver 2.0) library is located at the following folder:

```
Scheduler/libs/PaRLSched_2.0/
```

An example of how the library can be used can be found at:

```
Scheduler/Examples/PaRLSched_2.0_w_combs_example/
```

The source code provided in the example consists of two sub-directories, namely:

- `src`
- `PaRLSched`

The `PaRLSched` provides the source code for building the scheduler library, while the `src` directory provides an example of how the library can be linked and utilised. In particular, the provided example is `src/SchedulingExample.cpp`). In this example, first, a number of parameters are initialised necessary for running the scheduler. Then, a number of parallel (POSIX) threads following a standard farm pattern are created, while the computational task assigned to each thread corresponds to the computation of combinations of numbers. In fact, the pattern over which the threads are defined and the computational tasks executed is subject to the user and does not constrain the applicability of the scheduler. Furthermore, the details and nature of these computational tasks could be different for each thread and can be defined by the user within the `src/thread_routine.h` header file which is also provided.

4.3.3 Installation

Currently, the provided scheduler has been developed and tested over Linux Kernel 64bit 3.13.0-43-generic. For compiling the code, `gcc` (version 4.9.3) compiler has been used under the C++11 standard.

Before building the library, it is necessary that the PAPI-5.4.3 library interface for performance monitoring is being downloaded and installed. Download links and instructions can be found at [1]. Furthermore, the `pthread` library should be installed, which is available in several Linux distributions.

For building the example `src/SchedulingExample.cpp` and linking to the PaRLSched library, CMakeLists files are also provided. To build and run the scheduling example, simply run the following commands:

```
mkdir build
cmake -G "Unix␣Makefiles" -D CMAKE_BUILD_TYPE=Release
   (cont.)..
make
cd ./src
./SchedulingExample
```

In case it is desirable to edit the `SchedulingExample.cpp` provided and/or create a new example, the following commands generate the corresponding C++ project in Eclipse IDE.

```
mkdir build
cmake -G"Eclipse␣CDT4␣-␣Unix␣Makefiles" -D
  CMAKE_BUILD_TYPE=Debug ..
make
```

For example, in Eclipse IDE (Version: Mars.1 Release (4.5.1)) the project can be imported as follows: `File` → `Import` → `C/C++` and then select "*Existing Code as Makefile Project*". Then, enter the root directory of the project and select the Linux GCC tool-chain.

4.4 Discussion and Next Steps

In this chapter, we described the advancement of our work conducted under T4.3, and, in particular, the design and implementation of a Dynamic Scheduler tailored specifically for Patterned (Parallelised) Applications (`PaRLSched` library). The Dynamic Scheduler was advanced in three main directions: a) Hierarchical architecture for multiple resource optimization, b) Separating estimation from optimization, and c) Benchmark-based learning dynamics. These advancements provide a unified framework for dynamic resource allocation of patterned parallelised applications that is independent from the application's details (e.g., pattern structure).

Next steps include the following tasks:

- *Extensive testing:* We are planning to perform extensive tests with the new scheduler architecture when both the processing bandwidth and memory are being optimised. This will provide insights in possible modifications in the design of the estimation/optimization algorithms. We are also planning to perform tests with more demanding and data-intensive applications.

- *Advancing estimation methods:* We would like to incorporate the possibility that the estimator accepts hints from the application itself (e.g., pattern structure), to guide appropriately the formulation of the predictions. In general, it is necessary to identify the type of information that might be available from the application and how this can be explored within the scheduler.

- *Adapting parallel pattern:* Under certain circumstances, it might be desirable that the parallel pattern and its details also adapt to the resource availability. When the resource availability decreases due to other applications running on the same platform, reducing the number of threads may improve the application's performance. It would be desirable that the dynamic scheduler also initiates adaptation over the application's parallel pattern and/or its characteristics.

- *Integrating the Performance Monitoring Tool of T4.4:* Currently performance counters are provided through a standard implementation of the PAPI inter-

face. However, it is necessary that the Performance Monitoring Tool developed in T4.4 is also integrated since it provides extra-functional properties.

Chapter 5

Monitoring of Patterned Applications

Monitoring of pattern applications is an activity that turns out to be necessary for different purposes.

- *In primis*, tuning of patterned applications requires some precise knowledge concerning the "performances" (e.g. latency, bandwidth, consumed energy or power) achieved in the actual run of the application on the target hardware.
- Then, monitoring of the ongoing "performances" is required to trigger adaptivity in all those cases where the run time support may have alternative implementations of the same patterned application that better fit the current available conditions in terms of application parameters and/or target hardware availability.

In the former case, as an example, monitoring results may be used to tune the non functional parameters of the patterns used in the application at hand. In case we used a parallel pattern with a non functional parameter setting the parallelism degree at n_w and we verified the parallel pattern always waits for tasks to be computed from the previous computation stage (this can be verified by looking at the values relative to the monitored service times) we may conclude that the parallel pattern is over dimensioned and try improved program efficiency by decreasing its parallelism degree. In this case we may state that monitoring supports the decision process tuning an application running as a

$$\mathsf{pipe}(S_1, \mathsf{farm}(W, n_w))$$

to one running as a

$$\mathsf{pipe}(S_1, \mathsf{farm}(W, n_w - k))$$

In the latter case, monitoring results may be used to refactor "on-the-fly" the application parallel structure. When using a pipeline pattern where monitoring

reveals a bottleneck stage (i.e. a stage computing a task in a time much longer than the time needed to compute the other stages) we can easily refactor the parallel application structure to a pipeline with a farm stage, provided the stage business logic is stateless. In this case we may state the monitoring activity supports the decision process refactoring the application run as a

$$\mathsf{pipe}(S_1, \mathsf{pipe}(S_2), \mathsf{pipe}(S_3))$$

to one running as a

$$\mathsf{pipe}(S_1, \mathsf{farm}(\mathsf{pipe}(S_2), n_w), \mathsf{pipe}(S_3))$$

All these examples deal with monitoring of "time performance" but there will be similar activities and decisions that will deal with "power/energy performance" instead. The key point is monitoring activities provide from the execution of patterned applications those feed-backs needed to trigger and ensure the tuning and optimization processes.

When monitoring generic applications, monitoring performances requires the proper insertion of `probes` in different code locations and the gathering of the probed data in some place and format that may be used by the tuning and dynamic optimizer tools. Placing the necessary and correct probes in the code is quite an art, however, subject to errors and to the concrete possibility to introduce more and significant overhead possibly impairing the overall application performance.

When patterned applications are considered, however, monitoring turns out to be a much easier task due to the fact the parallel structure of the application may be completely exposed to the compiler/runtime/tuning/optimizing tools. Furthermore, the implementation of each one of the patterns provided by the programming framework may be designed *ab initio* to provide monitoring and, in particular, to provide the monitoring probes needed to figure out what's going on with the pattern at hand. The application specific activities relative to the monitoring of generic parallel applications may be turned into pattern specific activities and, as a consequence, moved from the responsibility of the application programmer to the responsibility of the system programmer providing the actual parallel pattern implementation.

In the previous WP4 deliverable D4.1, we provided different software components suitable to implement the monitoring of patterned applications targeting homogeneous (CPU only) architectures. The software components where basically embedded in FastFlow and in the support library `Mammut`.

Within FastFlow we provided basic mechanisms to gather the measures of interest in the execution of a given pattern, including the time spent in the computation of the tasks, the number of tasks processed, the parallelism degree, the amount of time spent awaiting for non existing input tasks or for a free output buffer. All these measures may be used to compute either completion time or service time for the pattern.

The mechanisms provided in FastFlow work "per pattern" and in D4.1 they were only providing measures for pattern implementations targeting CPU cores.

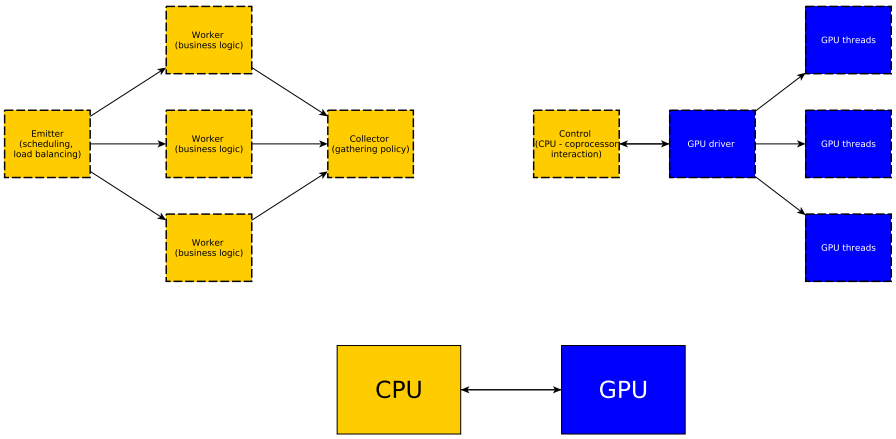


Figure 5.1: Pattern implementations of a map targeting CPU (left) and GPU (right)

The mechanism is a general mechanism, however, and may be used to implement the very same measures also for pattern implementations targeting GPUs.

Fig. 5.1 outlines the main features of two implementation of a map pattern, one targeting CPU cores and the other one targeting GPU(s), instead.

When monitoring the pattern performances in the pattern targeting the CPU cores the "emitter" thread is in charge of gathering all the information needed to expose the monitored measures, possibly interacting with the worker and collector threads. In case of the pattern targeting the GPU(s) the "control" thread interacting with the GPU(s) through the proper driver(s) may collect the very same kind of information using the driver specific mechanisms and provide the monitored measures through the very same interface implemented in the pattern implementation targeting the CPU cores.

As an example, the control thread in the GPU implementation knows how many map tasks have been scheduled to the GPU, how much time passed from the moment the tasks where sent to the GPU to the moment the results eventually come back from the GPU to the CPU, etc. and therefore it may easily pack that measured values in the proper format and deliver them upon requests issued by the application "manager" using the proper `ffStats` interface.

For all these reasons, we are not going to release any new API with this deliverable to access the monitored values of patterned applications targeting heterogeneous hardware.

The monitoring of energy or memory accesses in heterogeneous architectures are worth some extra words. Energy consumption is monitored using the APIs provided by the `Mammut` library. The library basically provides energy consumption using two distinct mechanisms:

- reading the counters available from the monitored micro-architecture, or
- reading the values measured by some kind of external "energy meter" through

some kind of interface (e.g. via USB, if the meter is directly connected to the machine using this interface, or via IPMI, in case the meter supports remote network access).

The version of `Mammut` (available at the `Mammut` git site https://github.com/DanieleDeSensi/mammut) will include all those mechanisms needed to read the relevant energy counters for Intel CPUs and nVidia GPUs. The FastFlow patterns will be eventually updated to include in the `ffStats` mechanisms the possibility to get the energy stats along with the time performance stats.

However, after delivering D4.1 at M14, at M16 we delivered in WP2 D2.4 "Software for pattern implementations for the initial pattern set" where we provided the initial design of a *generic pattern interface* fully compliant with the recent C++ (11 and 14) standards. The generic pattern interface (GrPPI) as initially designed in D2.4 only include the "functional" part of the pattern interface. Within WP2 we are currently refining and extending the GrPPI in such a way it includes also the non functional parameters specifying the pattern behaviour. The non functional part of the GrPPI will include also some support for the monitoring of the patterned applications, e.g. some way to specify which protocol has to be used to retrieve the monitored values from the running application or "post mortem".

At the moment being, the design of this part of GrPPI is not yet completed. It will be eventually delivered within D2.8 at M33. However, we agreed to deliver (partial) versions of the GrPPI as soon as they will available, in particular to the colleagues working on the use cases actually assessing the results of the RePhrase project. We plan to have the first partial release of the new GrPPI next early spring (somewhere close to M24) and that release will already have the monitoring stuff included, at least for part of the patterns implemented.

Last but not least, under the GrPPI umbrella different implementations of the patterns of the RePhrase extended pattern set (as described in D2.5) will be included. In D2.4 we already provided optimized implementations of the patterns include in D2.1 (the initial RePhrase pattern set) in FastFlow and preliminary implementations of the same patterns on top of C++11 threads and concurrency mechanisms, of OpenMP and of Intel TBB. Within the same deliverable, some relevant pattern have been implemented to target GPUs, notably in FastFlow through OpenCL and in C++11 through the Thrust library. As of RePhrase DoW and amendments, patterns implemented using different back-end frameworks (e.g. FastFlow, OpenMP or TBB) will not support interoperability. However, the possibility for the programmer to use GrPPI to write application code "portable" across different back end constitutes a big added value for the project, not identified in the original formulation of the RePhrase DoW. The consequences of the adoption of the GrPPI affect also the monitoring interface, indeed, and that's why the actual monitoring interface described in this document still refers only to FastFlow and `Mammut`.

Tool download The interested reader may download the FastFlow and `Mammut` tools from the `svn` at https://sourceforge.net/p/mc-fastflow/code/HEAD/tree/[1] and on the `git` at https://github.com/DanieleDeSensi/mammut respectively, and install and use them according to the instructions appearing on the web side and/or to the ones already included in D4.1.

[1] svn checkout svn://svn.code.sf.net/p/mc-fastflow/code/ mc-fastflow-code

Chapter 7

Conclusions

This deliverable described the extension of the work that was reported in D4.1 for the basic dynamic adaptation infrastructure. The set of tools described here allows adaptation of the application execution to the changes in the application behaviour and/or computing environment. Compared to the earlier versions of the tools reported in D4.1, in this deliverable we described additional mechanisms to deal with adaptation of the applications to the changes in input data of an application and to heterogeneous resources in the computing environment.

In particular, Chapter 3 described a linear-interpolation based mechanism for chosing the most suitable implementation of a parallel component of an application for a given scenario (in terms of the application input data and the availability of resources), based on novel features of the C++ language, concepts and attributes. This is a part of the static mapping framework and the mechanisms that we presented are hardware-independent, allowing the application to adapt to addition or removal of devices from the underlying computing environment. We also presented evaluation on the matrix multiplication and image processing applications, demonstrating that we are able to achieve good execution time while relieving the programmer from the daunting task of having to manually choose the component implementation. In the Chapter 4 we described a new version of the library for dynamic scheduling (`PaRLSched_2.0`) that extends the initial version reported in D4.1 in several directions. Firstly, in addition to targeting only dynamic (re)mapping of threads to processors, we the scheduler now also supports mapping of the application data to different memory nodes, and, in general, organising resources that need to be controlled (memory, processors etc.) into a hierarchical structure suitable e.g. for NUMA machines. Secondly, we decouple the estimation of the application performance from computing an optimisation strategy for making the dynamic scheduling decisions. This allows us a modular approach, where different optimisation strategies can be plugged in the scheduler without affecting the estimation of the performance, and vice versa. We demonstrate this by providing additional *aspiration learning* optimisation strategy that can be plugged instead of reinforcement learning, described in D4.1. Finally, the scheduler was

extended with adaptive mechanisms for making decisions about the frequency of estimation/optimisation decisions. These mechanisms allow faster response of the scheduler in the situation when rapid fluctuation of the application performance is observed, and slower response in other cases. In the Chapter 5, we have described how the performance monitoring infrastructure described in D4.1 can be used on heterogeneous computing systems, as well as our plans for adapting the same infrastructure to use the generic pattern interface (GrPPI) described in D2.4.

In the remainder of the project, we plan to extend the scheduling infrastructure to deal with heterogeneous computing systems. We also plan to extend all parts of the dynamic adaptation infrastructure to deal with issues that will arise from using *domain-specific* parallel patterns. We will also enable interoperability of the tools in this work package, so that they can be used together as a part of the **RePhrase** methodology.

Bibliography

[1] *Performance Application Programming Interface (PAPI 5.4.3) @ONLINE*, 2016. http://icl.cs.utk.edu/papi/software.

[2] Georgios C. Chasparis. Stochastic stability analysis of perturbed learning automata with constant step-size in strategic-form games. In *American Control Conference (submitted for publication)*, Seattle, WA, June 2017.

[3] Georgios C. Chasparis and Michael Rossbory. Efficient dynamic pinning of parallelized applications by distributed reinforcement learning. In *9th International Symposium on High-Level Parallel Programming and Applications (HLPP) (accepted for publication)*, Münster, Germany, July 2016.

[4] M.I. Daoud and N. Kharma. Efficient compile-time task scheduling for heterogeneous distributed computing systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 9 pp.–, 2006.

[5] Jens Maurer and Michael Wong. Towards support for attributes in C++ (Revision 6). In *JTC1/SC22/WG21 - The C++ Standards Committee*, 2008. N2761=08-0271.

[6] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[7] REPARA project. Target Platform Description Specification. https://repara-project.eu/wp-content/uploads/2014/04/ICT-609666-D3.1.pdf, February 2014.

[8] Luis Miguel Sanchez, David del Rio Astorga, Manuel F. Dolz, and Javier Fernández. CID: A Compile-time Implementation Decider for Heterogeneous Platforms based on C++ Attributes. In *2016 Intl IEEE Scalable Computing and Communications*, pages 1149–1156, 2016.

[9] LuisMiguel Sanchez, Javier Fernandez, Rafael Sotomayor, Soledad Escolar, and J.Daniel. Garcia. A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures. *New Generation Computing*, 31(3):139–161, 2013.

[10] Andrew Sutton. Working Draft, C++ Extensions for Concepts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4040.pdf, May 2014.

[11] Andrew Sutton and Bjarne Stroustrup. Concepts Lite: Constraining Templates with Predicates. http://concepts.axiomatics.org/~ans/concepts-lite.pdf, January 2012.