Project no. 644235

REPHRASE

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

Software for the dynamic adaptivity for patterned applications on initial pattern set for homogeneous hardware architectures. D4.1

Due date of deliverable: 31st May 2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP4

Responsible institution: IBM
Editor and editor's address: Michael Vinov, IBM

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	07/10/16	Georgios Chasparis	SCCH	Changed link of PaRLSched library to the GIT repository (Section 4.4.2)
2	07/10/16	Georgios Chasparis	SCCH	Updating reference [6] in Bibliography section.
3	13/10/16	Michael Vinov	IBM	Fixing text errors

Executive Summary

This document is the initial 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 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 give the overview of the basic versions of the tools for static mapping, dynamic scheduling and performance monitoring, together with the JIT compiler.

Contents

1. Introduction

In this deliverable we describe 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 here provide basic dynamic adaptation infrastructure for homogeneous computing systems and generic parallel patterns, and will be extended in the course of the project with additional capabilities and adapted to domain-specific patterns (that are to be developed in WP2) and heterogeneous computing systems. In this way, we are addressing a crucial aspect of software engineering for parallel systems, which is automatic adaptation of software to the changing hardware and software setting.

In Chapter 2, we describe tools for static mapping of the patterned application components to the underlying hardware resources (T4.1). This involves the decisions about distributing/replicating data and mapping application threads to the processor cores. We also describe a mechanism for making decisions about instantiation of the pattern components (in terms of the number of components of each type to be created). In Chapter 3, we describe adaptive, just-in-time (JIT) compiler (T4.2) that can be used to dynamically, during the application execution, switch between alternative versions of the same function (or component) as a response to the changing behaviour of application and/or system, or additional knowledge gained during the application execution (e.g. knowledge about input data). Chapter 4 describes a dynamic scheduling mechanisms that uses machine learning to decide on re-mapping of the application's parallel threads to the processor cores (T4.3). This allows adapting the mapping decisions that were made by the static mapping tools to the changing settings. Finally, in Chapter 5 we present a performance monitoring infrastructure (T4.4), which allows collecting information about the application behaviour which can be used to drive the decisions made by the dynamic compilation and scheduling tools.

Chapter 2

Static Mapping of Software Components and Data to Hardware Resources

In this chapter, we briefly describe the work done so far in task 4.1. We first give an overview of the task, then describe a preliminary version of parallel patterns that contain static mapping of the pattern application's threads and data to the underlying hardware. Afterwards, we describe mechanisms for deriving (near-) optimal configuration of the application's pattern parameters, including the number of threads used by each farm and pipeline stage. The code for tools and use cases described in this chapter can be found in the RePhrase GitHub repository, at https :// github.com/uoscompsci/RePhrase, under the directory StaticMapping.

2.1 The Static Mapping Problem for Data Intensive Applications

Parallel patterns represent a very useful abstraction in bridging gap between the high level programming models in which programmers ideally want to code, and the complexity of modern multi-core/many-core hardware. However, in order to keep the high-level of abstraction, many low-level details related to thread and data management (such as the placement of threads and placement/replication of data) have to remain implicit to the pattern implementations. In many cases, these decisions can be left to the underlying language runtime system or even operating system itself, without impacting the parallel performance of patterns. However, in the case of applications with complex composition of patterns, making the decisions of how many threads (and of which type) to create for each component of the pattern structure and how to map these threads to the underlying hardware can be highly non-trivial, especially if the hardware is heterogeneous (i.e. comprising of a combination of CPUs and accelerators, such as GPUs). *Data-intensive* ap-

plications present additional complexity. Since they operate on potentially large volumes of data, careful decisions need to be made where in the memory to map the data to avoid problems like memory contention and false sharing. In addition to that, on Non-Uniform Memory Architectures (NUMA) access to some parts of the memory can be significantly more expensive for threads running on certain cores than for others. Therefore, *replication* of the data to multiple memory banks to reduce memory contention and memory access latency may be necessary to obtain good performance.

T2.1 deals with developing a software infrastructure for static mapping of patterned applications to the hardware resources. In static mapping, the decisions about how many threads to create and where to map threads and data are made at compile time, and not influenced by the runtime behaviour of the application and/or system (as in *dynamic* mapping). In this deliverable, we present a preliminary version of the infrastructure, focusing on the basic patterns and homogeneous computing systems, comprising only of multicore CPUs.

2.2 The Patterns with Explicit Data Mapping

We have extended some of the patterns presented in D2.1 with mechanisms to do explicit mapping and replication of the data on the NUMA architectures. For this purpose, we have used the libnuma library that is available on Linux and which allows different policies for memory allocation. In libnuma, main memory is divided into a number of *memory nodes*, and each node has a set of associated CPUs (which correspond to physical CPU cores). Latency in access to the remote nodes for CPUs is much more expensive than access to memory from their associated node. Memory can be explicitly allocated on a certain memory node using a numa_alloc_onnode function. It is also possible to set preferred nodes for memory allocation, or to set interleaved allocation so that memory nodes are selected in round-robin fashion. libnuma also allows pinning of threads to individual cores, or a group of cores that are closest to a certain memory node.

Based on these capabilities, we have developed a new version of the map parallel pattern that replicates the read-only data that it operates on to multiple memory nodes. In the preliminary version of the static mapping tool, we have used the Intel TBB library to demonstrate these new patterns, while we are also in the process of doing the same in the OpenMP and FastFlow libraries. The same technique can easily be applied to the farm and pipeline patterns. The code for the new version of map can be found in libs/parfor_numa.hpp, and it requires the libnuma libraries to be installed.

2.2.1 Map Pattern with Data Replication

We will focus on the most common form of the map pattern, parallel_for. For Intel TBB, we provide a MapReplicationClass class, a wrapper around the sequential

computation that is to be executed in parallel by the parallel_for loop. The constructor of this class, which is executed before the actual threads to perform the computation in parallel are created, copies the data to the multiple memory nodes, depending on the nodes available in the system and the number of threads to be created. The basic version of the constructor for this class is:

```
1  MapReplicationClass(
2      std::function<void(int,std::vector<void*>,R*)> f,
3      int nr_cpu_w,
4      std::vector<data_t> a,
5      R *r,
6      int chunk_size)
```

f is a sequential worker function that is the body of the for loop to be executed in parallel. It receives as an input a loop index (int), a vector of input data (std::vector (cont.)<void*>) and an output array of the type R (parameter of the class). nr_cpu_w is the number of parallel threads to be created, a is a vector of input data that is to be replicated, r is an array where the results of the computation are to be stored (which, of course, can be null in the case that the computation does not return a result), and chunk_size is a size of a chunk of for loop indices (in case that we use chunking). data_t type describes one array of data that will be replicated:

```
1 typedef struct {
2   void *p_data;
3   unsigned int size;
4 } data_t;
```

where p_data is a pointer to the array to be copied and size is the size of the array (in bytes). The constructor first determines the number of copies of the data to be created, which depends on nr_cpu_w and the number of available memory nodes and then iterates over the arrays in a and creates the appropriate number of copies on different memory nodes. The MapReplicationClass also contains an instance of the () operator, which is called separately for each parallel thread that is created by the parallel_for skeleton. There, the treads are distributed to processor cores close to the memory nodes where the data is replicated, and, for each thread, the actual worker function f from above is executed on a range of loop indices and with the data from the closest memory node. Later in this section, we will show two examples of the use of the MapReplicationClass class.

We also provide two alternative versions of the FarmReplicationClas constructor. In some cases, the same parallel_for pattern will be executed multiple times (for example, if it is nested within some outer loop). If the input data does not change between successive iterations, it would be very inefficient to replicate the same data each time parallel_for is executed. For these situation, we provide a variant of the

class constructor with an additional argument where the replicated data is stored:

```
1  MapReplicationClass(
2     std::function<void(int,std::vector<void*>,R*)> f,
3     int nr_cpu_w,
4     std::vector<data_t> a,
5     R *r,
6     int chunk_size,
7     std::vector< std::vector<void*> > *data)
```

The returned data can be used by the third version of the constructor which, instead of a vector of elements of data_t type accepts a vector of vectors of void* pointers, which contains the already replicated data that can be used by the parallel_for loop.

```
1  MapReplicationClass(
2     std::function<void(int,std::vector<void*>,R*)> f,
3     int nr_cpu_w,
4     std::vector< std::vector<void*> > data,
5     R *r,
6     int chunk_size)
```

2.2.1.1 Toy Example

As an example of the use of the MapReplicationClass class, consider the following loop:

```
1  void worker_function (int worker_index,
2                        int *in,
3                        double *res)
4 {
5  double x=0;
6  for (int i=0; i<arr_size; i++) {
7    x += in[i] + in [arr_size-1-i] - in [0] + in [arr_size-1];
8    x += in[i] + in [arr_size-1-i] - in [0] + in [arr_size-1];
9  }
10 res[worker_index] = x/1000;
11}
12
13for (int i=0; i<10000; i++)
14 res[i] = worker_function (i, *input_array)
```

The worker_function performs a simple operations on the input array in and stores the results in the output array res. The for loop at the line 11, that calls this function a number of times, can be trivially turned into parallel_for :

```
class WorkerClass {
public:
  void operator()(const blocked_range<size_t>& r) const {
    for (size_t i=r.begin(); i!=r.end(); i++) {
      worker_function (i, a, res);
    }
  }
  WorkerClass() {}
};

...

parallel_for(blocked_range<size_t>(0, 10000, chunk_size),
    (cont.)WorkerClass());
```

However, by doing this, we are faced with some problems. The worker function is constructed in such way that, in the same loop iteration, it accesses possibly distant parts of the input array. If the size of the array is large, this can lead to many cache misses, which turn into expensive accesses to the main memory. Furthermore, if there is only one copy of the array, in the case of larger number of threads in the parallel_for loop, we would have a situation where many threads access the same portion of the main memory at the same time, which leads to memory contention. To avoid the problem of memory contention, we can use the parallel_for with replication. This requires a slight change to the worker function which, due to the way MapReplicationClass is constructed, needs to accept a vector of arrays (i.e. void pointers) as an input:

```
  void worker_function (int worker_index,
                        std::vector<void*> data,
                        double *res)
{
  double x=0;
  int *in = (int *) data[0];
  for (int i=0; i<arr_dim; i++) {
    x += in[i] + in[arr_dim-1-i] - in[0] + in[arr_dim-1-i];
    x += in[i] + in[arr_dim-1-i] - in[0] + in[arr_dim-1-i];
  }
  res[worker_index] = x/1000;
}
```

We also need to pack the input data into a structure that will allow the MapReplicationClass (cont.) to replicate the input data appropriately:

```
std::vector<data_t> in_array(1);
data_t arr = {.p_data = (void *) a,
              .size = sizeof(int)*array_dim};
```

```
in_array[0] = arr;
```

where a is an input array, which is globally defined. Finally, provided that we have chunk_size and nr_cpu_w variables set up, and res array for storing results, we can call the parallel_for pattern in the same way as the regular parallel_for :

```
parallel_for(blocked_range<size_t>(0, 10000, chunk_size),
             MapReplicationClass<double>(worker_function,
                                         nr_cpu_w,
                                         in_array,
                                         res,
                                         chunk_size));
```

The MapReplicationClass will then determine the number of copies of the array a that should be created (which depends on the number of available memory nodes and the number of threads to be created, nr_cpu_w), copy the array to different memory nodes, create threads, pin the threads to the cores close to the appropriate memory nodes and call the worker function, supplying the nearest copy of the array a as a second argument to worker_function for each thread. The complete source code for this example can be found in the examples/toyexample−numa.cpp, with the basic TBB code without data replication in examples/toyexample−tbb.cpp.

2.2.1.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) benchmark, and its parallelisation using the Intel TBB library, was already described in the D2.2. As a reminder, ACO is a heuristic for solving hard optimisation problems where, over a number of iterations, individual *ants* compute solutions to the problem independently, biased by a *pheromone trail*. At the end of each iteration, there is a synchronisation step where the current best solution is picked as (currently) optimal and the pheromone trail is updated. We apply this technique to solve a scheduling problem, where a set of jobs, with corresponding processing times and deadlines, need to be scheduled on a single machine. For each job that ends after its deadline, a penalty (determined by its *weight*) is paid. Pheromone trail used to compute the individual solutions is stored in the tau matrix. The relevant part of the sequential is given below:

```
for (j=0; j<num_iter; j++) {
  for (i=0; i<num_ants; i++) {
    cost[i] = solve (i, weight, deadline, process_time, tau);
  }
  best_t = pick_best(&best_result);
  update(best_t, best_result, tau);
}
```

In the above code, individual solutions to the scheduling problem are computed using the solve function which also returns their costs (total penalties paid for all jobs that end after their designated deadline). Afterwards, the best solution is computed using the pick_best function and the pheromone trail is updated using the update function. Note that, as opposed to the Toy Example, here we have two nested for loops and only the inner loop can be parallelised:

```
1  for (j=0; j<num_iter; j++) {
2     /******* Intel TBB farm ********/
3     parallel_for(blocked_range<size_t>(0, num_ants, min_chunk_size
          (cont.)), CPU_Solve_Farm_Component());
4     /*******************************/
5     best_t = pick_best(&best_result);
6     update(best_t, best_result, tau);
7  }
```

Therefore, the inner parallel_for will be executed multiple times. At the same time, that loop operates on potentially very large amount of data (weight, deadline (cont.), process_time and tau arrays) and the same problem of memory contention as in Toy Example can occur. However, replicating each of these arrays each time parallel_for loop is invoked can be very expensive. We want to replicate the arrays only the first time parallel_for is executed, and to reuse the created copies on its each subsequent execution. For that purpose, we can use the alternative constructor for the MapReplicationClass class, where the pointer to the replicated data is available to the "outer world":

```
1  std::vector< std::vector <void*> > data;
2
3  for (j=0; j<num_iter; j++) {
4    /******* Intel TBB farm ********/
5    if (j==0)
6      parallel_for(blocked_range<size_t>(0,
7                                         num_ants,
8                                         min_chunk_size),
9                   MapReplicationClass<unsigned int>(f,
10                                                    nr_cpu_w,
11                                                    input_data,
12                                                    cost,
13                                                    min_chk_size,
14                                                    &data));
15   else
16     parallel_for(blocked_range<size_t>(0,
17                                        num_ants,
18                                        min_chunk_size),
19                  MapReplicationClass<unsigned int>(f,
20                                                    nr_cpu_w,
21                                                    data,
```

```
22                                                      cost,
23                                                      min_chk_size));
24      /*****************************/
25      best_t = pick_best(&best_result);
26      update(best_t, best_result, tau);
27    }
```

Here, in the first call to the parallel_for (line 6), we use the variant of a constructor that accepts an extra argument (in this case, &data) where the replicated data is stored. This data is then used in the subsequent instances of the inner loop (line 16), without the need for replication. The data that is replicated include the process_time (cont.), deadline, weight and tau arrays. However, there is a problem with the code above. If we use the MapRepliationClass class, the tau array is replicated, so there is no single instance of it as in the sequential code or the code that uses parallel_for without replication. Therefore, we need to modify the update function to work on an array of tau arrays:

```
1   for (i=0; i<nr_nodes; i++)
2     update(best_t, best_result, data, i)
```

We can also parallelise this code, since updating of different copies of the tau array can be done independently. Therefore, the final parallel version of the main loop of the program is:

```
1   std::vector< std::vector <void*> > data;
2
3   for (j=0; j<num_iter; j++) {
4     /******* Intel TBB farm ********/
5     if (j==0)
6       parallel_for(blocked_range<size_t>(0,
7                                          num_ants,
8                                          min_chunk_size),
9                    MapReplicationClass<unsigned int>(f,
10                                                     nr_cpu_w,
11                                                     input_data,
12                                                     cost,
13                                                     min_chk_size,
14                                                     &data));
15    else
16      parallel_for(blocked_range<size_t>(0,
17                                         num_ants,
18                                         min_chunk_size),
19                   MapReplicationClass<unsigned int>(f,
20                                                     nr_cpu_w,
21                                                     data,
22                                                     cost,
23                                                     min_chk_size));
```

```
24      /*****************************/
25      best_t = pick_best(&best_result);
26      parallel_for(blocked_range<size_t>(0, nr_mem_nodes, 1),
            (cont.)Update_Component())
27  }
```

The complete source code is given in examples/ant_colony_numa.cpp, with the examples (cont.)/ant_tbb.cpp containing the basic TBB version without data replication.

2.3 Static Mapping of Patterned Application Threads

The patterns for mapping threads and replicating data to the underlying hardware resources, that we described in the previous section, assume knowledge about the number of threads that will be used in each map in the application. Similarly, in the equivalent versions for farm or pipeline patterns, we will have to know exact number of workers in a farm or a number of threads/tokens used in the pipeline. If the pattern structure of the application is relatively simple (e.g. there is only a map/farm or a pipeline at the top level, and there is no nesting of patterns) then the situation is simple - we can just set the number of threads for each farm to be the number of physical processor cores on the system. However, in the cases where there is nesting of patterns or where computing system has a more complex structure (in terms of heterogeneous hardware) these decisions can be very non-trivial. For example, we can have a pipeline of three stages, each of which can have a farm inside of it. Using the notation of the Refactoring Pattern Language (RPL) that was described in D2.2, we can describe such application structure as

```
seq ( pipe ( farm (stage1), farm (stage2), farm (stage3) ) )
```

Then, the question is how many threads to assign to each of the farms inside of the pipe to get the best performance, energy consumption or whatever other metric we are interested in. While we can try profiling all possible options, i.e. consider every combination for the number of threads in each map/farm and pipeline stage, run the application and measure performance, this can be very expensive for long running applications. Furthermore, the more complex pattern structure the application has and the larger the system is (in terms of number of cores), the more combinations are there to be considered. Therefore, we have implemented several heuristics for computing a (near) optimal number of threads in each farm and pipeline stage in a patterned application. Our heuristics address a more general problem, which is deriving an optimal configuration of any extra-functional pattern parameters that can, besides the number of threads to use, include chunking and any application-specific parameters. In order to use these heuristics, the application needs to be parameterised, i.e. pattern parameters need to be exposed as the application command-line

arguments, so that, when we decide on the assignment of the parameters, we can actually run the application and measure the performance. In order to use the heuristics, user needs to provide an input file with the range for each of the parameters that the heuristic needs to estimate. The parameters that should not be estimated will have a fixed value. For example, in the Ant Colony Optimisation example above, the first parameter of the application can be the number of threads to use in the first parallel for (the main piece of work in the application, where the function solve is applied to find solutions), the second parameter can be the number of workers in the update parallel_for , whereas the remaining arguments are fixed (number of iterations, number of ants and the input file). The input file for each of the heuristics on a machine with 64 cores would then look like this:

```
1  ant_colony_numa
2  0-64
3  0-64
4  1
5  64000
6  inputs/wt1000.txt
```

This means that the application can be run from command like by, for example

```
1 ant_colony_numa 64 64 1 64000 inputs/wt1000.txt
```

In the following Sections, we describe the implemented heuristics.

2.3.1 Exhaustive Search

If the hardware system is relatively small, the pattern structure of the application is relatively simple or the application is short running, trying out all possible combinations of values for pattern parameters is a perfectly good option and, of course, the only one guaranteed to give us the optimal result (i.e. optimal values for all pattern parameters). Therefore, the first heuristic that we implemented is the Exhaustive Search over all possible pattern configurations in order to find the best one. As we have mentioned before, this is a very expensive method and should only be used for short running applications and where a range of values for each pattern parameter is relatively small. The source code for this heuristic is in libs/(cont.)exhaustive_search.cpp. For the above example, it can be executed via

```
1 exhaustive_search inputFile
```

2.3.2 Random Sampling

A method that works reasonably well in some situations is to just do random assignment of values for the pattern parameters a number of times and return the best one. In this way, we hope that we will 'hit' the assignment that is close to optimal one without having to evaluate all (or most) of the assignments. This works well in situations where there is a relatively small number of possible configurations. Our implementation of Random Sampling for static mapping is provided in the libs / (cont.)random_sampling.cpp file, and it accepts, together with the input file with the ranges for parameters, a number of random configurations that will be evaluated. Obviously, larger this number is, closer to the optimal will the result be but also more time it will take to evaluate the chosen configurations. It can be executed, for example, by calling

```
1 random_sampling inputFile 100
```

2.3.3 Hill Climbing

Hill Climbing is a heuristic based on incremental changing of an initial solution until we reach a (locally) best solution. In our case, we start from a pattern configuration that we approximate as a good one (for example, by assigning equal number of threads to each farm and pipeline stage in an application or using some simple cost model to calculate a good configuration) and then we incrementally change the individual parameters until we get a configuration that gives the worse performance than the previous one, at which point we stop and return the current best configuration as the overall best. The source code for our implementation of Hill Climbing is in libs / hill_climbing .cpp, and the heuristic can be executed via

```
1 hill_climbing inputFile
```

2.3.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) approach is used for generating and evaluating large game trees in Game theory. We adapted this method to the problem of finding the optimal configuration of pattern parameters. In our case, the nodes of the tree correspond to partial configurations (with some of the pattern parameters chosen) and the leaves of the tree correspond to complete configurations. The children of a node represent different possible assignments of a yet unassigned pattern parameter. We start from a tree that consists only of a single root node (i.e. a configuration where no parameters are chosen), and we proceed by repeating the following three

steps:

1. *Expansion step* – A node is selected, and one of its children is added to the tree. This is equivalent to assigning a value to a yet unassigned parameter;

2. *Selection step* – Starting from the newly added node, a complete configuration is generated by randomly assigning the remaining unassigned parameters. The resulting static mapping is evaluated by the evaluation function, Q, yielding the valuation v;

3. *Propagation step* – The valuation, v, is propagated back to the node added in step 1.

Steps 2 and 3 are repeated until a reliable statistical evaluation of Step 1 is attained. Then, Step 1 is repeated, adding a new value to the partial configuration. Finally, the overall best configuration (a leaf of the tree) is selected.

The MCTS heuristics is mostly useful when there is a large number of possible configuration, so that we can only realistically evaluate a small number of them. Also, as opposed to Hill Climbing, it does not assume that there is regularity in behaviour of the application with respect to changes of the pattern parameters. Hill Climbing is known to have a potential problem of getting stuck in a local maximum, so if a particular pattern configuration achieves a local maximum in terms of performance and is better than any configuration obtained by small modifications of parameters, then Hill Climbing might reach that configuration and return it as the best one, despite the fact that some other "distant" configuration (with more significant changes in the values of parameters) might be better. MCTS does not suffer from this problem, due to a random choosing of nodes in the selection step. Therefore, if the application exhibits irregular behaviour in terms of the effect of the pattern parameter changes to the performance, MCTS is better option than Hill Climbing. The source code for MCTS is in libs/monte_Carlo.cpp and, similar to Random Sampling, it requires an additional parameter, which is a number of random configuration to try when evaluating a particular partial configuration:

```
1 monte_carlo inputFile 100
```

Again, as in Random Sampling, the more random configurations we try, the closer to the optimal the result will be, but also the execution will take more time.

2.4 Next Steps

The immediate next steps in T4.1 is to implement the data replication mechanisms for the pipeline parallel pattern and to implement similar mechanisms as described in this deliverable in OpenMP and FastFlow frameworks. After that, we will add

further refinements of the patterns to deal with other common problems in parallelising applications, e.g. false sharing of data. We will also extend the mapping tools to deal with heterogeneous architectures that comprise of combinations of CPU and GPU processors. This will involve making decisions about which data to map to the GPUs and how to distribute work between CPUs and GPUs. Finally, we will extend all the mechanisms to advanced (domain-specific) parallel patterns that will be developed in the remainder of the project.

Chapter 3

Adaptive Compilation of Patterned Applications

In this section we describe the tool for dynamic recompilation. The tool is based on the LLVM compiler 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 versions of the function and measuring the appropriate metrics (e.g. execution time in the case of performance).

Using the dynamic compilation tool requires two stage compilation: First the program has to be compiled to its LLVM intermediate representation (IR), called bitcode. This stage is done statically using LLVM clang compiler. The IR is then fed into the dynamic recompilation tool which compiles and run the functions during the execution of the program. This tool is based on an initial version which can be found in

```
http://lists.llvm.org/pipermail/llvm-dev/2015-September/090536.
    (cont.)html
```

3.1 Building and Installing the tool

This section describes the steps of building and installing the tool. The dynamic compiler is built on top of LLVM compiler. The compiler should be built and run on x86.

3.1.1 Download and install GCC 4.9.2

In order to build the LLVM compiler, GCC needs to be installed first. Building of the LLVM was tested using the GCC version 4.9. To download this version, do

```
svn co svn://gcc.gnu.org/svn/gcc/tags/gcc_4_9_2_release gcc
```

After downloading GCC, follow the links below to install GCC:

https://gcc.gnu.org/svn.html

https://gcc.gnu.org/wiki/InstallingGCC

Here is an example of options to be passed to the GCC configure script:

```
../gcc/gcc/configure --prefix=`pwd`/../install_gcc_4_9_2
--disable-multilib --enable-languages=c,c++
--disable-bootstrap
```

3.1.2 Download and install LLVM

The next step is to download and install LLVM. The URL link below contains instructions for downloading and installing LLVM:

http://clang.llvm.org/get_started.html

Here is an example of options to be passed to the LLVM configure script:

```
cmake -DCMAKE_C_COMPILER=`pwd`/../install_gcc_4_9_2/bin/gcc
-DCMAKE_CXX_COMPILER=`pwd`/../install_gcc_4_9_2/bin/g++
-DGCC_INSTALL_PREFIX=`pwd`/../install_gcc_4_9_2/
 -DCMAKE_CXX_LINK_FLAGS="-L`pwd`//../install_gcc_4_9_2/lib64
-Wl,-rpath,`pwd`//../install_gcc_4_9_2/lib64 -l stdc++ -rdynamic"
-DLLVM_ENABLE_ASSERTIONS=ON -DLLVM_TARGETS_TO_BUILD="X86"
-DCMAKE_ASM_COMPILER=`pwd`//../install_gcc_4_9_2/bin/gcc ../llvm/
-DCMAKE_C_FLAGS=" -ggdb -O0 -g -rdynamic" -DCMAKE_CXX_FLAGS="
-ggdb -O0 -g -rdynamic"
-DCMAKE_C_LINK_FLAGS="-L`pwd`//../install_gcc_4_9_2/lib64
-Wl,-rpath,`pwd`//../install_gcc_4_9_2/lib64   -rdynamic
" -DCMAKE_INSTALL_PREFIX=`pwd`/../install_llvm/
-DCMAKE_LINKER=`pwd`/../install_binutils/bin/ld
```

3.1.3 Building the dynamic compiler

The dynamic compiler is based on a toy example provided in the LLVM examples directory, which can be found in llvm/examples/Kaleidoscope/Orc/fully_lazy/toy.(cont.)cpp The dynamic compiler code should reside under the LLVM examples directory. To build the compiler, the following steps should be taken:

1. *Extract the dynamic compiler directory under examples*

```
cp WPfolders/WP4_folder/tools/dynamic_recomp.tar.gz /path-
    (cont.)to-llvm-src/examples/Kaleidoscope/Orc
```

The dynamic_recom file then needs to be extracted.

2. *Add toy.cpp to the example path of examples/Kaleidoscope/Orc.* Add subdirectory fully_lazy_with_recompile_try to the bottom of /path-to-llvm-src/examples (cont.)/Kaleidoscope/Orc/CMakeLists.txt file.

3. *Copy the latest version of toy.cpp.* copy the latest version of toy.cpp in WP4_folder/tools/dynamic_recomp_versions/ to /path-to-llvm/examples/Kaleidoscope (cont.)/Orc/fully_lazy_with_recompile_try/toy.cpp

4. *Adjust toy.cpp to the latest changes in LLVM.* This step is optional. It might be that toy.cpp version provided under dynamic_recomp_versions should be adjusted to be compatible with the newest version of LLVM in order to pass compilation. Here are some known changes that might be needed:

```
CHANGE #include "llvm/ExecutionEngine/Orc/OrcTargetSupport
    (cont.).h"
TO  #include "llvm/ExecutionEngine/Orc/
    (cont.)OrcArchitectureSupport.h"

CHANGE  JITCompileCallbackManager<OrcX86_64>
    (cont.)CompileCallbacks;
TO  LocalJITCompileCallbackManager<OrcX86_64>
    (cont.)CompileCallbacks;
```

5. *Download and install OpenMP library* Download and install the OpenMP libraray from http://openmp.llvm.org/

6. *Update the dynamic libraries path in toy.cpp.* Edit the libraries path in function main in toy.cpp file with the machine path. For example:

```
llvm::sys::DynamicLibrary::LoadLibraryPermanently("/path/to
    (cont.)/libc.so.6", &ErrMsgStr);
llvm::sys::DynamicLibrary::LoadLibraryPermanently("/path/to
    (cont.)/libomp.so", &ErrMsgStr);
```

7. *Compile.*

```
cd /path/to/llvm/build/
make Kaleidoscope
```

When compilation passes OK,

```
/path-to-llvm-build/bin/Kaleidoscope-Orc-
    (cont.)fully_lazy_with_recompile_try
```

8. Run the examples. Before running examples in

```
RePhrase/Papers/Drafts/EuroPar2016-Mapping/examples/
```

you have to adjust PATHS in Makefile.common.user to reflect your build/install location.

3.2 Description of the code

The Initial version of the dynamic compiler was provided by Lang Hames and is available in

```
http://lists.llvm.org/pipermail/llvm-dev/2015-September/090536.
    (cont.)html
```

This version is based on toy examples provided in the LLVM code under llvm/(cont.)examples/Kaleidoscope/Orc directory.

The *Orc (On Request Compilation)* mechanism does lazy compilation, which is the ability to compile the code for a function at the point when it is called. Up to that point, the code is saved in its IR (Intermediate Representation), called *bitcode*. The Orc directory contains examples that use the lazy compilation scheme. The dynamic compiler we used also relies on the Orc mechanism for doing lazy compilation.

High level description of the dynamic compiler was provided by Lang Hames in

```
http://lists.llvm.org/pipermail/llvm-dev/2015-July/088665.html
```

The initial version described in the link recompiles the "hot" functions by first creating their instrumented versions, which include a counter at the beginning of each function so that after pre-defined threshold an optimised version of the function will be generated and run. The idea is that only the hottest function will be recompiled with special compilation options that could take advantage of run-time

information collected in the instrumentation phase. The version of the code we worked on for the RePhrase project relies on this idea, but contains some adjustments. The dynamic recompiler accepts as input different versions of the same function; each one is compiled separately, and the aim is to choose, at run-time, the best version to execute. To do that, the dynamic recompiler first saves the different versions of the function in an internal data-structure. Once a call to a function with multiple versions is made, the first version of the function is compiled to its instrumented version. The instrumented version contains a counter, so after several executions, a call to the dynamic recompiler will be issued in order to compile and run the more suitable version. Every time a function version has been run, its start and end times are measured, the purpose of which is to choose the version with the smallest execution time when no new versions are left to be compiled.

3.2.1 List of functions

Following is the description of the relevant functions for dynamic recompilation.

- main function iterates over the files intermediate representation (IR), loads them and fires up the run of the program by executing main.
- parseInputIR function iterates over the functions IRs in a specific file and insert them to the internal structures from which they are later retrieved.
- getAddress function returns the address of a symbol. In the LLVM lazy compilation scheme, a function is compiled to the executable code only when
- getAddress has been called on its name symbol. Otherwise, its code is saved as IR.
- addModule adds a module to the lazy compilation scheme. This means that a module will be compiled only when getAddress is called on it.
- createLambdaResolver is executed when we try to compile a function. It tries to find out how to compile the function, e.g. by searching whether it is in a dynamic library. When the function has multiple versions, its name will be recompile_version and the recompileNextVersion function will be called to compile the next version of the function if it exists.
- recompileNextVersion function is responsible for recompiling the next version of the function. In case all the versions have been compiled, it will call recompileSelectedVersion1 for the final compilation of the selected function.
- searchModule2 function returns the address of the compiled function. When a function has more than one version, it will generate a stub for it and return the stub's address.

- irGenStubIR function generates a stub which, when called at run-time, compiles the instrumented version of the function. The instrumented version is identical to the original version, except that it has a counter in its prologue, such that the function returns to the recompilation mechanism when the counter has reached a threshold.

- instrumentFunctions1 returns the instrumented version of a function. The instrumented version is identical to the original function except that, after certain number of times it has been called, it returns to the recompilation mechanism. The condition which checks if the counter exceeds the threshold is

```
Value *Condition = B.CreateICmpUGT(CounterVal,
    (cont.)ConstantInt::get(Int32, threshold));
```

A snippet from the dump of the IR of an instrumented function is shown in the box below. A global counter variable called foo$counter holds the number of times the function has been executed. If counter exceeds a threshold, a call to a function called recompile_version is generated. The argument to this call will be the original function. The call to recompile_version will trigger a call to createLambdaResolver to resolve its symbol.

```
1  define i32 @foo(i32 %x, i32 %y, i32* %res) #-1 !dbg !0 {
2  entry:
3    \%x.addr = alloca i32, align 4
4    \%y.addr = alloca i32, align 4
5    \%res.addr = alloca i32*, align 8
6    br label %entry_cont
7    \%counter = load i32, i32* @"foo$counter"
8    \%0 = icmp ugt i32 %counter, 1
9    \%1 = add i32 %counter, 1
10   store i32 %1, i32* @"foo$counter"
11   br i1 %0, label %recompile, label %entry_cont
12 .
13 recompile:                                          ; preds =
     (cont.) %entry
14   \%11 = call i64 @"$recompile_version"(i64
       (cont.)140733799726640, i64 73249448)
15   \%12 = inttoptr i64 %11 to i32 (i32, i32, i32*)*
16   \%13 = tail call i32 %12(i32 %x, i32 %y, i32* %res)
17 }
```

3.2.2 Data structures

Following are data-structures used to save and restore information of the functions versions:

- hashFunc is a hash table that holds, for each entry, an array of function pointers, each for a different version of a function. The entries are specific to a function and are thus accessed with the function name. To retrieve the next version in process, MapVersionNum array is used.
- MapVersionNum is a map between function name and the index in the hashFunc (cont.) vector that belongs to that function.
- hashFuncStartTime and hashFuncEndTime are the data structures where information of the version start and end time is stored. Each has a vector of doubles in their entries, which, in combination with the function name and the version number, can retrieve and store the times. The times are updated in the irGenStubIR and recompileNextVersion functions. The first function records the start time of the first version of the function before its compilation and run, while the second function records the end time and time and the start time for the rest of the versions. The version with the smallest run-time execution is chosen in the recompileNextVersion function.

3.2.3 Examples

The RePhrase GitHub repository (https :// github.com/uoscompsci/RePhrase) contains toy examples in DynamicCompilation directory that demonstrate the dynamic recompilation tool. The compilation process starts with generating the .bc (bitcode) files of the program. The .bc files contain the intermediate representation of the functions that are to be dynamically compiled. Later on, they are processed and compiled by the dynamic recompilation mechanism. The functions that are considered for recompilation are defined in separate files, one file per each function version. For example, when compiling program that calls the function foo which has two versions, the dynamic recompiler will be invoked in the following way:

```
./Kaleidoscope-Orc-fully_lazy_with_recompile_try
main.bc foo_v1.bc foo_v2.bc
```

3.2.3.1 Building and running the examples

Build:

```
cd /to/the/examples/directory
make build
```

Run:

```
cd /to/the/examples/directory
make run
```

Chapter 4

Dynamic Scheduling of Patterned Application

4.1 Introduction

In this chapter, we describe the work conducted under T4.3, the primary goal of which is the design and implementation of a Dynamic Scheduler tailored specifically for Patterned (Parallelised) Applications. Such scheduler should be able to increase the robustness and resilience of parallel and data-intensive applications when running under non-homogeneous and possibly shared hardware resources. We are moving towards a fully *automatic* Dynamic Scheduler, which will be able to automatically discover 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).

In particular, the main challenges involved in the design of such Dynamic Scheduler are dealing with the following aspects:

(1) *Discover an "optimal" allocation of resources 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*).

(2) *Fast response to changes in the availability of hardware resources*. When other applications are running on the same platform, the Dynamic Scheduler should be able to adapt fast to the new situation and discover a new optimal resource allocation.

In this deliverable, we provide a description of the first prototype of the Dynamic Scheduler. In this first prototype, *the Dynamic Scheduler addresses the problem of automatically and dynamically discovering the optimal assignment of threads*

(pinning) into a homogeneous hardware platform (in fact, a set of identical CPU processing units). It is 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.

Our goal was to investigate whether a dynamic discovery tool of the optimal allocation of CPU cores into threads can outperform the mechanisms that are currently available in operating systems. This was indeed the case, as we will demonstrate in detail in the forthcoming experiments section. In this first prototype, we are experimenting with basic patterns (that is, farm and map patterns); however, the methodology described is generic enough to accommodate any pattern structure. Furthermore, in this first prototype, we are not addressing the problem of dynamic memory allocation, since this is primarily a concern for non-homogeneous hardware architectures, and it will be part of the next deliverable (D4.2).

The material presented in this deliverable is based upon the recent publications conducted under T4.3 [7, 6].

4.2 Related Work & Contribution

4.2.1 Related work

When resource allocation is performed online and the number, arrival and departure times of the tasks are not known a-priori (as in the case of CPU bandwidth allocation), the role of a Dynamic Scheduler is to guarantee an *efficient* operation of *all* tasks by appropriately distributing resources. However, guaranteeing efficiency through the adjustment of resources usually requires the formulation of a centralised optimisation problem (e.g., mixed-integer linear programming formulations [3]), which further requires information about the specifics of each task (i.e., the application's details). Such information may not be available to neither the Dynamic Scheduler nor the task itself.

Given the difficulties involved in the formulation of centralised optimisation problems, not to mention their computational complexity, feedback from the running tasks in the form of performance measurements may provide valuable information for the establishment of efficient allocations. Such (feedback-based or measurement-based) techniques have recently been considered in several scientific domains, such as in the case of application parallelization (where information about the memory access patterns or affinity between threads and data are used in the form of scheduling hints) [4], or in the case of allocating virtual processors to time-sensitive applications [5].

Dynamic resource allocation problems have also been addressed through *distributed* or *game-theoretic* optimisation schemes. The main goal of such approaches is to address a centralised (*global*) objective for resource allocation through agent-based (*local*) objectives, where, for instance, agents may represent the tasks to be allocated. For example, when addressing the problem of dynamically binding threads to CPU cores, each thread may be treated as an independent *decision*

maker or *agent* which makes its own decisions regarding which CPU core to run on. Examples of such approaches include the cooperative game formulation for allocating bandwidth in grid computing [11], the non-cooperative game formulation in the problem of medium access protocols in communications [12] or for allocating resources in cloud computing [14].

The main advantage of distributing the decision-making process is the considerable reduction in computational complexity (a group of N tasks can be allocated to m resources with m^N possible ways, while a single task may be allocated with only m possible ways). This further allows for the development of online selection rules where tasks/agents make decisions often using current observations of their *own* performance.

In this work, we propose a *distributed learning scheme* specifically tailored for addressing the problem of dynamically assigning/pinning threads of a Parallelised application to the available processing units. Prior work has demonstrated the importance of thread-to-core bindings in the overall performance of a Parallelised application. For example, [8] describes a tool that checks the performance of each of the available thread-to-core bindings and searches for an optimal placement. Unfortunately, the *exhaustive-search* type of optimisation that is implemented may prohibit runtime implementation. Reference [4] combines the problem of thread scheduling with scheduling hints related to thread-memory affinity issues. These hints are able to accommodate load distribution given information for the application structure and the hardware topology. The HWLOC library is used to perform the topology discovery which builds a hierarchical architecture composed of hardware objects (NUMA nodes, sockets, caches, cores, etc.) and the BubbleSched library [13] is used to implement scheduling policies. A similar scheduling policy is also implemented by [10].

The aforementioned scheduling strategies exhibits several disadvantages when dealing with dynamic environments (e.g., varying availability of resources). In particular, retrieving the exact affinity relations during runtime may be an issue due to the involved information complexity. Furthermore, in the presence of other applications running on the same platform, the above methodologies will fail to identify irregular application behaviour and react promptly to such irregularities.

4.2.2 Contribution

In dynamic environments, it is more appropriate to consider *learning-based optimisation techniques* where the scheduling policy is being updated based on the performance measurements from each running thread. Furthermore, since performance measurements can be collected independently from each thread, this further allows for distributing the decision-making process. Through such *distributed-learning-based* scheme, we can

(a) *reduce information complexity* (i.e., when dealing with a large number of possible thread/memory bindings) since only performance measurements need to be collected during runtime,

(b) *reduce computational complexity*, since decisions with respect to the resources assigned to each thread are taken independently for each thread.

(b) *adapt to uncertain/irregular application behaviour* (e.g., when threads' bandwidth demand changes with time, when the number of threads changes with time, etc.),

(c) *adapt to changes in the environment* (e.g., the resource availability changes with time when other applications are also running on the same platform).

To understand the differences between a distributed learning scheme as compared to centralised (static) schemes, consider the following scenario. Let us assume that thread i of a Parallelised application is running on CPU core j. Let us also assume that at some time the load of the jth CPU core increases (due to, e.g., other applications starting running on the same core). Then, thread i will experience a drop in its performance. If the decision-making process is independent for each thread, then thread i may immediately react to this change by changing its CPU affinity without necessarily altering the assignment of the remaining threads. Such immediate reaction to changes in the environment constitutes a great advantage in comparison to static schemes. In the absence of an explicit form of a centralised objective, a centralised framework would require a testing period over which all possible assignments are tested and then compared with respect to their performance. When all possible assignments have been tested and evaluated, then the best one can be selected. Even if such optimisation is repeated often, it is obvious that such *exhaustive-search* type approach will suffer from slow responses to possible changes in the environment.

To this end, we developed a *distributed learning-based* scheme for optimally allocating threads of a Parallelised application into a set of homogeneous CPU cores. The proposed methodology is implemented through a Dynamic Scheduler which runs in parallel to the application and it is responsible for all the necessary computations and executions related to the assignment of threads into CPU cores. We will briefly refer to this scheduler as *PaRLSched*, and in the remaining sections we will describe in detail its architecture and functionality.

4.3 Dynamic Scheduler

4.3.1 Architecture

Parallelised applications consist of multiple threads that can be controlled independently with respect to their CPU affinity (at least in Linux machines). Thus, decisions over the assignment of CPU affinities can be performed independently for each thread, allowing for the introduction of a *distributed learning* framework. Under such a framework, the Dynamic Scheduler treats each thread as an independent decision maker and provides each thread with an independent decision rule.

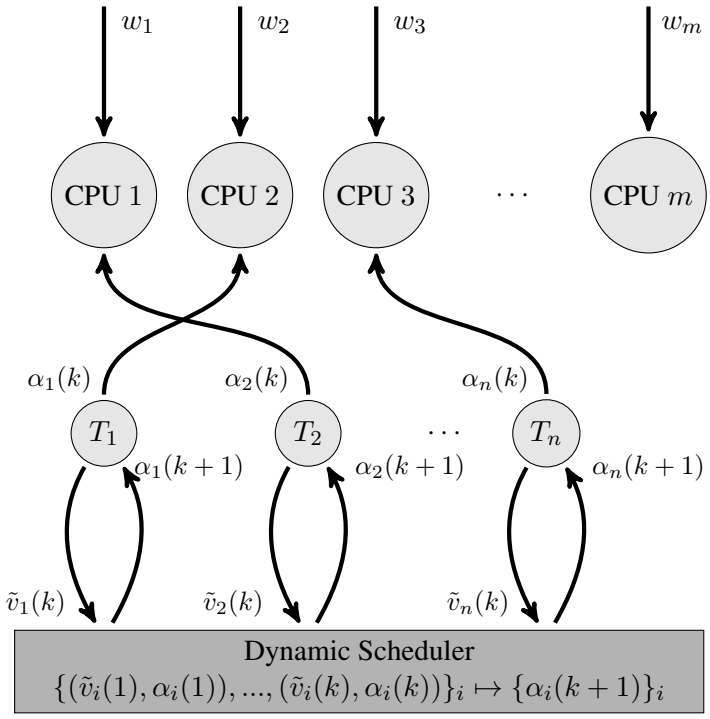

Figure 4.1: Schematic of *dynamic* resource allocation framework.

A schematic of the architecture of the dynamic resource allocation framework is provided in Figure 4.1.

4.3.2 Strategy

We will consider a set $\mathcal{I} = \{1, ..., n\}$ of threads which need to be assigned to a set $\mathcal{J} = \{1, ..., m\}$ of available CPU cores. We may view the decision over which CPU thread i will run as the realisation of a finite probability distribution σ_i, such that $\sigma_{ij} \in [0, 1]$, $j \in \mathcal{J}$, denote the probability that thread i selects CPU j. We set $\sum_{j \in \mathcal{J}} \sigma_{ij} = 1$, so that $\sigma_i \doteq (\sigma_{i1}, ..., \sigma_{i|\mathcal{J}|})$ is a probability distribution over the set of CPU's (or *strategy* of agent i). To provide an example, consider the case of 3 available CPU cores, i.e., $\mathcal{J} = \{1, 2, 3\}$. In this case, the strategy σ_i of thread i may take the following form:

$$\sigma_i = \begin{pmatrix} 0.2 \\ 0.5 \\ 0.3 \end{pmatrix},$$

such that 20% corresponds to the probability of assigning itself to CPU core 1, 50% corresponds to the probability of assigning itself to CPU core 2 and 30% corresponds to the probability of assigning itself to CPU core 3. Briefly, the assignment

of a thread i (or *action*) will be denoted by

$$\alpha_i = \mathsf{rand}_{\sigma_i}[\mathcal{J}].$$

4.3.3 Strategy update (reinforcement-learning)

We briefly provide here the core of the Dynamic Scheduler, which is based upon a *perturbed reinforcement-learning* scheme. According to the proposed scheme, the probability that thread i selects action j at time $k = 1, 2, ...$ is:

$$\sigma_{ij}(k) = (1 - \lambda)x_{ij}(k) - \frac{\lambda}{|\mathcal{J}|} \tag{4.1}$$

where $\lambda > 0$ corresponds to a perturbation term (or *mutation*) and $x_i(k)$ corresponds to the *nominal strategy* of agent i. The nominal strategy is updated according to the following update recursion:

$$x_{ij}(k+1) = \begin{cases} x_{ij}(k) + \epsilon \cdot u_i(\alpha(k)) \cdot [1 - x_{ij}(k)], & \text{if } \alpha_i = j \\ x_{ij}(k) + \epsilon \cdot u_i(\alpha(k)) \cdot x_{ij}(k), & \text{else.} \end{cases} \tag{4.2}$$

for some constant step-size $\epsilon > 0$. The function $u_i(\cdot) > 0$ represents the *utility* of thread i which usually corresponds to a performance indicator. It is a function of the actions of all threads or *action profile*, denoted $\alpha(k)$, since the performance of each thread depends on the allocations of all threads. For example, the utility function assigned to thread i may correspond to its processing speed.

Note that according to the above recursion, the new nominal strategy will increase in the direction of the currently selected action $\alpha_i(k)$ and it will increase proportionally to the utility received from this selection.

The asymptotic behaviour of this update recursion depends primarily on the characteristics of the utility function assigned to each thread. The goal is to assign a utility function so that when each thread records its own performance and updates its strategy as above, then gradually the strategies of all threads approach an allocation that maximises a *global* objective. In the context of this deliverable, we have experimented with two alternative utility functions:

(O1) $u_i(\alpha) \doteq \sum_{i=1}^{n} v_i/n$, where v_i is the processing speed of thread i;

(O2) $u_i(\alpha) \doteq \sum_{i=1}^{n} [v_i - \gamma(v_i - \sum_{j \in \mathcal{I}} v_j/n)^2]/n$, for some $\gamma > 0$.

Note that the first utility function corresponds to the average processing speed of all threads, i.e., each thread prefers to select actions that increase the average processing speed of all threads. On the other hand, the second utility function corresponds to the average processing speed minus a penalty that is proportional to the speed variance among threads. In other words, according to the second utility function, each thread prefers an allocation that maximises the average processing speed while maintaining small speed differences with all other threads.

Finally, it is important to note that the Dynamic Scheduler may not know the explicit dependence of the processing speed $v_i = v_i(\alpha)$ of thread i to the action profile α. Instead, it may only have access to measurements of this function, denoted $\tilde{v}_i$.

4.4 Dynamic Scheduler Library (`PaRLSched`)

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

4.4.1 Description

The PaRLSched library is currently developed for Linux platforms and utilises the `sched.h` library for CPU-affinity bindings and the PAPI interface [9] for real-time collection of performance counters during the execution time of a thread. 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 main components:

- `thread_info.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).
- `thread_control.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 with all the necessary computations for determining the next allocation strategy for all active threads (as determined in detail in Section 4.3.3).

4.4.2 Provided software

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

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

The PaRLSched library is located at the following folder:

```
Scheduler/PaRLSched/ver_1.0_w_combs_example/
```

The source code provided 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.

4.4.3 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 ..
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` $\rightarrow$ `Import` $\rightarrow$ `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.

4.5 Experiments

In this section, we present an experimental study of the proposed reinforcement learning scheme for dynamic pinning of parallelised applications. Experiments were conducted on `20×Intel©Xeon©CPU E5-2650 v3 at 2.30 GHz` running Linux Kernel 64bit 3.13.0-43-generic. The machine divides the physical cores into two NUMA nodes (Node 1: 0-9 CPU's, Node 2: 10-19 CPU's).

4.5.1 Experimental Setup

We consider the `SchedulingExample.cpp` example described in the previous section that executes a fixed number of computations (corresponding to the combinations of M out of a set of $N > M$ numbers). The routine is being parallelised using the `pthread.h` (C++ POSIX thread library), where each thread is executing a replicate of the above set of computations. The nature of these computations does not play any role and in fact it may vary between threads (as we shall see in both the forthcoming experiments).

Throughout the execution, and with a fixed period of 0.3 sec, the Dynamic Scheduler collects measurements of the total instructions per sec (using the PAPI library [9]) for each one of the threads separately. Given the provided measurements, the update rule of Equation (4.2) with the utility function under (O2) is executed by the Dynamic Scheduler. In the following, we demonstrate the response of the Dynamic Scheduler in comparison to the Operating System (OS) response (i.e., when placement of the threads is not controlled by the Dynamic Scheduler). We compare them for different values of $\gamma \geq 0$ in order to investigate the influence of more balanced speeds to the overall running time.

In the forthcoming experiments, the Dynamic Scheduler is executed within the master thread which is always running in the first available CPU (CPU 1). Furthermore, in all experiments, only the first one of the two NUMA nodes are utilised, since our intention is to demonstrate the potential benefit of an efficient thread placement when the effect of memory placement is rather small.

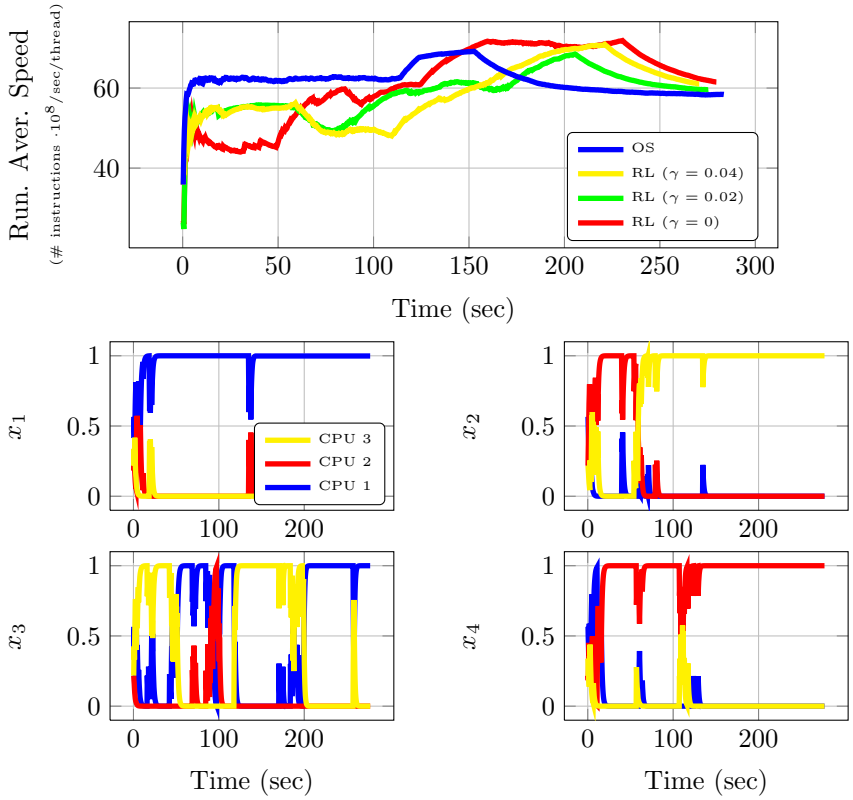

Figure 4.2: Running average execution speed when 4 threads run on 3 identical CPU's. Thread 3 requires about half the computing bandwidth compared to the rest of the threads which are identical. The strategies correspond to the RL scheme with $\gamma = 0.04$. The RL schemes run with $\epsilon = 0.005$ and $\lambda = 0.005$.

4.5.2 Experiment 1: Non-Identical Threads under Limited Resources

In this experiment, we consider the case of limited resources (i.e., when the number of threads is larger than the number of available CPU's). However, one of the threads requires CPU time with smaller frequency than the rest of the threads (i.e., executes its computations with smaller frequency). We should expect that in an optimal setup, threads that do not require CPU time often should be the ones sharing a CPU. On the other hand, threads that require larger bandwidth, they should be placed alone.

In particular, in this experiment, Thread 3 requires about half the computing bandwidth compared to the rest of the threads (Thread 1, 2 and 4). The resulting performance is depicted in Figure 4.2.

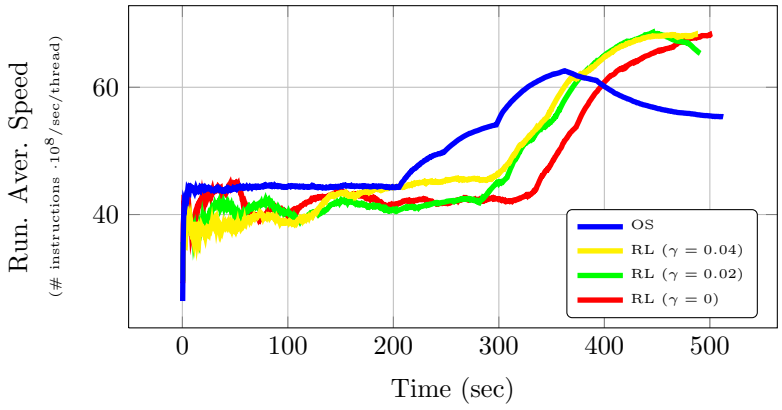

Figure 4.3: Running average execution speed when 7 non-identical threads run on 3 CPU cores. Threads 1 & 2 require about half the computing bandwidth compared to the rest of the threads (which are identical). Thread 3 is joining after 120 sec. The RL schemes run with $\epsilon = 0.003$ and $\lambda = 0.005$.

We observe indeed that Threads 1, 2 & 4 (which require identical bandwidths) are allocated to different CPU's (CPU 1, 3 and 2, respectively). On the other hand, Thread 3 is switching between CPU 1 and CPU 3, since both provide almost equal processing bandwidth to Thread 3. In other words, the less demanding application is sharing the CPU with one of the more demanding threads.

Note, finally, that the difference with the processing speed of the OS scheme is small, although a more balanced processing speed ($\gamma = 0.04$) improved slightly the overall completion time.

4.5.3 Experiment 2: Non-Identical Threads in a Dynamic Environment

In this experiment, we demonstrate the robustness of the algorithm in a dynamic environment. We consider 7 threads. The first two (Threads 1 & 2) require about half the computing bandwidth compared to the rest. The rest of the threads (Threads 3, 4, 5, 6 and 7) are identical. However, Thread 3 starts running later in time (in particular, after 120 sec).

Figure 4.3 illustrates the evolution of the RL scheduling scheme under different values of γ. Again in this case, a fastest response of the overall application can be achieved when higher values of γ are selected. The difference should be attributed to the fact that the OS fails to distinguish between threads with different bandwidth requirements. Table 4.1 presents a statistical analysis of these schemes where the speed difference between the RL ($\gamma = 0.04$) and the OS reaches approximately 5% on average.

Run #	OS	RL ($\gamma = 0$)	RL ($\gamma = 0.02$)	RL ($\gamma = 0.04$)
1	513 sec	505 sec	492 sec	489 sec
2	530 sec	506 sec	489 sec	494 sec
3	536 sec	517 sec	518 sec	515 sec
4	533 sec	507 sec	515 sec	509 sec
5	523 sec	502 sec	491 sec	496 sec
6	513 sec	523 sec	501 sec	492 sec
7	520 sec	514 sec	497 sec	492 sec
8	530 sec	518 sec	499 sec	497 sec
9	520 sec	532 sec	500 sec	497 sec
10	528 sec	517 sec	493 sec	492 sec
aver.	**524.6 sec**	**514.10 sec**	**499.5 sec**	**497.3 sec**
s.d.	**8.06 sec**	**9.29 sec**	**9.85 sec**	**8.27 sec**

Table 4.1: Comparison between the OS performance and RL schemes when $\epsilon = 0.003$ and $\lambda = 0.005$ for different values of γ under Experiment 2.

4.6 Discussion and Next Steps

In this chapter, we described the work conducted under T4.3, and, in particular, the design and implementation of a Dynamic Scheduler tailored specifically for Patterned (Parallelised) Applications (`PaRLSched` library). Next steps include the following tasks:

- *Automatic tuning:* Currently the parameters of the Dynamic Scheduler has been tuned for the specific Linux platform under which the experiments were conducted. It will be necessary that the Dynamic Scheduler is tuned automatically, so that it is independent of the platform under which it runs.

- *Integrating the Performance Monitoring Tool of T4.4:* Currently performance counters are provided through a standard implementation of the PAPI interface. However, it is necessary that the Performance Monitoring Tool developed in T4.4 is also integrated since it provides extra-functional properties.

- *Efficiently allocating memory:* Currently, the Dynamic Scheduler is addressing the problem of efficiently allocating CPU cores into the running threads. Our next goal is to extend the resource allocation tool to also accommodate memory allocation issues in a dynamic fashion.

- *Incorporating advanced patterns:* The developed Dynamic Scheduler is generic enough to accommodate advanced parallel patterns. However, underlying information about the pattern structure may enhance the resulting performance when searching for optimal allocations. We will advance the Dynamic Scheduler to exploit such additional information.

- *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.

Chapter 5

Monitoring of Patterned Applications

In this section we provide an introduction to the monitoring mechanisms for patterned parallel applications, developed in the project as a part of T4.4. These mechanisms can be used to profile running applications by gathering different kinds of measurements (e.g., performance, power/energy consumption), which can later be used to drive decisions made by static/dynamic mapping engine and dynamic recompilation framework, developed in T4.1, T4.2 and T4.3.

This section is organised as follows. In Section 5.1, we describe the performance monitoring mechanisms implemented for the FastFlow parallel programming framework[1]. These mechanisms are native, i.e. they are implemented as part of the parallel runtime system. In Section 5.2, we describe the Mammut library[2] (MAchine Micro Management UTilities), which can be used to profile the power-based statistics of running applications. This library is general, and can be used to profile any sequential/parallel code. It will be used to gather power-related measurements from the parallel patterns implemented in the project. Finally, in Section 5.3, we introduce the PMLib library for power profiling of scientific codes. This library takes the problem of power/energy profiling of sequential codes by offering a different interface. Both Mammut and PMLib will be synergically used within the project and integrated properly in the pattern implementation that will be developed in WP2.

5.1 Performance Monitoring in FastFlow

In this section we briefly describe how to compile a FastFlow program in order to effectively use the run-time performance monitoring mechanisms to collect online profiling measurements of the parallel code. First, we provide a short guide

[1] http://mc-fastflow.sourceforge.net/

[2] http://danieledesensi.github.io/mammut/

to install Fastflow. Then, we describe the possible ways to use the performance monitoring mechanisms and we show their potential through some examples.

5.1.1 Building and installing the tool

This section describes the steps for installing Fastflow. FastFlow is provided as a set of header files (`*.hpp`). The installation process requires downloading the latest version of the FastFlow source code from the SourceForge:

```
svn co https://svn.code.sf.net/p/mc-fastflow/code fastflow
```

The library does not require compilation, but the header files must be directly included by the application programs and compiled properly. It is highly advised to always compile programs that use the FastFlow library with the `-O3` optimisation flag enabled, to achieve good performance. Also, remember that the correct compilation of FastFlow programs requires linking the *pthread* library (`-pthread` flag).

To enable the performance monitoring system, it is necessary to compile all the FastFlow programs with the macro TRACE_FASTFLOW defined. An example of a compilation of a program test.cpp, which uses the FastFlow library with monitoring mechanisms enabled, is the following:

```
g++ -std=c++11 -I$FF_ROOT -O3 -DTRACE_FASTFLOW test.cpp
-o test -pthread
```

5.1.2 Monitoring mechanisms

FastFlow provides several methods to acquire statistics from the current application execution. Such mechanisms can be invoked on a single `ff_node` instance (the FastFlow abstraction for a thread), or on complete patterns and their composition/nesting (e.g., pipe, farm, map, stencils). The information that can be gathered is the following:

- the *working time* of a FastFlow node. It is the total time passed in executing user-code by a FastFlow node. This time is counted from the beginning of the execution of the node, and it does not include wasted time periods, e.g., time periods in which the node is waiting for the reception of a new incoming task;

- the *total number of processed tasks* by the FastFlow node from the beginning of the execution;

- the number of failed pop/push operations from/to the input/output queues of a node. We recall that a push fails if the destination queue is full, while a pop fails just in case the input queue is empty.

After the termination of a pattern/node (by invoking the run_wait_end() method), the programmer can inspect the statistics by calling the ffStats () method of the target node/pattern. The method has the following signature:

```
virtual void ffStats(std::ostream &)
```

the user can provide any output character stream, and the monitoring mechanism will write on it the global statistics collected during the application execution in a human-readable form. A code example that uses the monitoring mechanics in this way is the following:

```
19 int main() {
20  ...
21  // create a pipeline object with two stages
22  ff_node *source = new Source(...);
23  ff_node *sink = new Sink(...);
24  ff_Pipe<> pipe(source, sink);
25  // the threads of the pipeline runs synchronously
26  if(pipe.run_and_wait_end() < 0) {
27   cerr << "Error" << endl;
28   return -1;
29  }
30  pipe.ffStats(std::cout);
31  return 0;
32 }
```

In this example the monitoring mechanism is used *synchronously*, to collect information of the pipeline execution after the end of the processing. The application is a pipeline of two sequential stages. The first stage, (source), generates a stream of $1M$ integers with a rate of $50,000$ elements per second. The second stage, (sink), absorbs the tasks with a processing time of 100ms per task. In this simple case the programmer provides the std::cout (standard output) stream to have a graphical representation of the monitoring output on the console. The following snippet shows a possible output of the execution of the program.

```
--- pipeline:
ID: 0  work-time (ms): 155478
  n. tasks       : 1000001
  svc ticks      : 53401802932 (min= 43 max= 514197)
  n. push lost   : 24847639 (ticks=24847639000)
```

```
  n. pop lost    : 0 (ticks=0)
ID: 1  work-time (ms): 155560
  n. tasks       : 1000000
  svc ticks      : 79449442269 (min= 55228 max= 2123865)
  n. push lost   : 0 (ticks=0)
  n. pop lost    : 56 (ticks=56000)
...end ff_pipe.
```

In the output, the two stages are marked with a progressive identifier. The output shows, for each node, the monitored measurements with additional information (e.g., maximum and minimum values of some measurements). In this example, the second stage is slower than the first one. This is evident in the monitoring results, as the number of lost push operations performed by the first stage is quite large. In fact, since all the FastFlow queues are by default bounded (of a finite size specified when the pipeline is built), the input queue of the second stage becomes full quickly and the first stage attempts to deliver new tasks many times by finding the destination queue full.

In order to perform an online monitoring of the running FastFlow patterns, i.e. to profile the actual Quality of Service while the application is running, the monitoring mechanisms can be used as shown in the following piece of code:

```
1  int main() {
2   ...
3   // create a pipeline object with two stages
4   ff_node *source = new Source(...);
5   ff_node *sink = new Sink(...);
6   ff_Pipe<> pipe(source, sink);
7   // the threads of the pipeline runs synchronously
8   if(pipe.run) {
9    cerr << "Error" << endl;
10   return -1;
11  }
12  while(!pipe.done()) {
13   sleep(1);
14   pipe.ffStats(std::cout);
15  }
16  pipe.wait();
17  std::cout << "DONE\n";
18 }
```

Instead of executing the run_and_wait_end() method of the pipeline pattern, we can execute the pattern *asynchronously* with respect to the main thread. In this way the program periodically prints the monitoring results on the screen at every sampling interval (in this case of 1 second). In particular, the pipe.done() method call will

return true only after the pipeline has actually finished computing every task appearing onto its input stream. The subsequent pipe.wait() call will also wait for the completion of all the pipeline operations, including the thread management operations (e.g. shutdown of the pipeline threads). Furthermore, instead of using the ffStats() method to gather monitoring data as a stream of characters, we can use the following methods offered by the ff_node class:

```
double getworktime() const;
size_t getnumtask()  const;
ticks  getsvcticks() const;
size_t getpushlost() const;
size_t getpoplost()  const;
```

The semantics of the methods is intuitive: they allow the programmer to gather the profiling measurements described above from single ff_node objects. Such methods can be used to gather data that can be saved into a repository, or made available through a proper interface to the other components of the WP4 tool-chain (e.g., static/dynamic mapping engine or dynamic recompilation infrastructure).

5.1.3 Use Cases

In this section we show the potential of the FastFlow monitoring infrastructure through a simple set of examples. The idea is to monitor the performance of a patterned application in order to discover in real-time which node/pattern is a bottleneck. This use case is particularly important for the WP4 goals, since the possibility to detect bottleneck threads is a precondition to apply dynamic re-scheduling strategies of running programs on homogeneous/heterogeneous systems.

The examples can be found in the tar-ball ff_monitoring.tgz, downloadable at the following url

```
https://svn.code.sf.net/p/mc-fastflow/code/tests/monitorning
```

5.1.3.1 Pipeline example

This first example is a simple pipeline program with four stages. *Source* produces a stream of tasks, *Stage2* receives tasks and simply forwards them, *Stage3* has the same behaviour of *Stage2*, and *Sink* absorbs the tasks. Each task has a structure of two integers: a unique identifier and a value. Each stage emulates a generic computation by executing a usleep with a fixed waiting time[3]. This is to repro-

[3]Since threads are locked to cores in FastFlow and the threads are mapped to cores 1-to-1, if enough cores are available, the usage of the usleep conveniently emulates a true computation, as far as computation time is concerned.

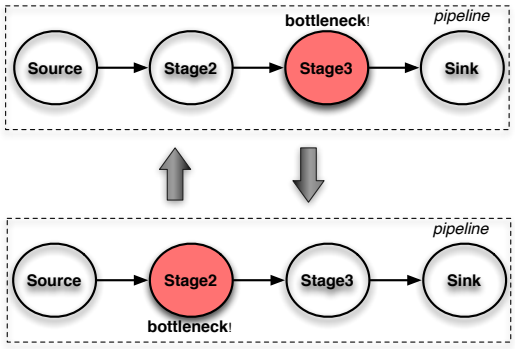

Figure 5.1: First example of performance monitoring in FastFlow.

duce stages with different computational grains. The main thread creates and starts the pipeline and performs monitoring activities asynchronously with respect to the computation. Instead of executing the ffStats() method, the main executes periodically a function which:

- inspects the structure of the pattern. In this case it is a pipeline of four sequential nodes. For each node, monitoring data are gathered;
- the function derives the ideal service times of the four stages (i.e. the average running time per task measured in the last sampling interval);
- by comparing the ideal service times, the function prints on the screen which node of the pipeline is a bottleneck during the last sampling interval.

The program can be executed by passing the following command-line arguments ($-l$ option specifies the stream length in terms of tasks, $-s$ the sampling interval in seconds):

```
$ ./bin/pipe -l <stream len> -s <sampling interval in secs>
```

Typical parameters to be used in this case are -l 100000 and -s 1. For a proper translations of the service time measured in ticks, it is necessary to edit the Makefile and change the value of the FREQ macro (the CPU frequency is expressed in Mhz, the default value is $2,000$ Mhz).

The behaviour of the stages is configured to emulate a dynamic situation in which, for the first part of the execution, the third stage is the bottleneck, then the slowest node becomes the second stage and finally the third stage becomes the bottleneck again. This is obtained by changing dynamically the waiting times per task used by the nodes of the pipeline. Fig. 5.1 shows a representation of this first example.

Fig. 5.2 show the results of this experiment by depicting the service time of the four stages at each sampling interval.

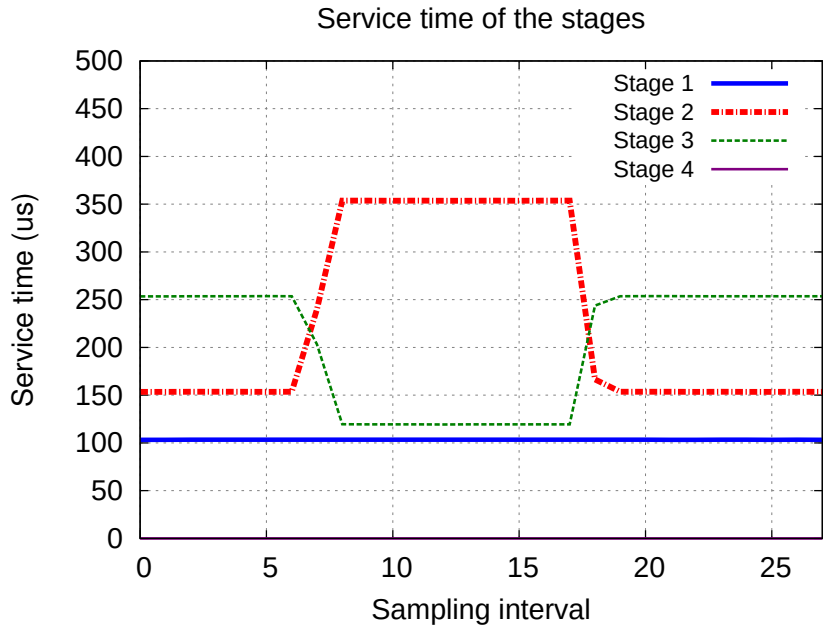

Figure 5.2: Results of the first use case.

5.1.3.2 Pipeline and Farm example

The second example has the same structure as the previous one except that the third stage is a farm of two workers. The farm has also an emitter and a collector node. The idea of this example is the same as before: the monitoring thread inspects the performance statistics at regular sampling intervals. In the first part of the execution the second stage is the bottleneck, so the farm resources are underutilised. Then, at a certain point of the execution, the second worker of the farm increases its service time and becomes the new bottleneck of the application. Finally, in the last part of the execution the emitter of the farm becomes the bottleneck, as shown in Fig. 5.3. The running command is the following with the same command-line arguments of the previous example:

```
$ ./bin/farm -l <stream len> -s <sampling interval in secs>
```

5.2 The Mammut Library

In this section we describe the Mammut library[4] for real-time power monitoring of sequential/parallel computations on multicores. Mammut provides an object-oriented abstraction of architectural features normally exposed by means of sysfs files or CPU hardware counters on Linux/UNIX OSs. In the last release, the library

[4] https://github.com/DanieleDeSensi/mammut

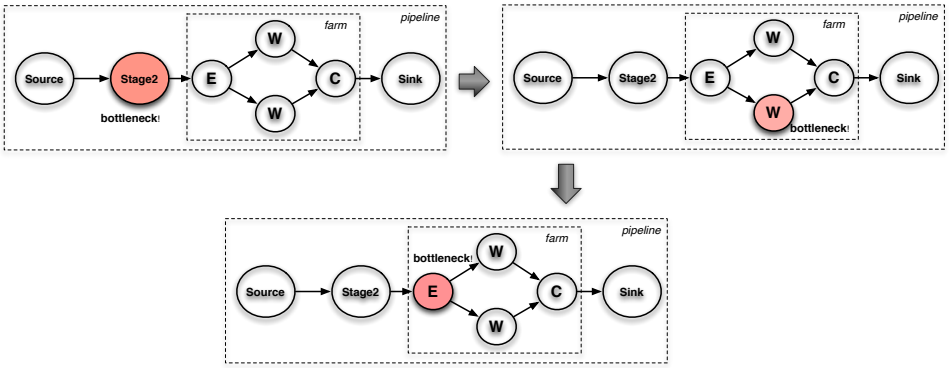

Figure 5.3: Second example of performance monitoring in FastFlow.

also provides the possibility to manage distributed nodes by using a client-server mechanism.

5.2.1 Building and installing the tool

Mammut is a C++ library that can be installed on multicore machines equipped with a UNIX-like OS. To download the last release the user can execute the following commands on the shell:

```
$ git clone git://github.com/DanieleDeSensi/mammut.git
$ cd mammut
$ make
$ make install
```

By default Mammut is installed under the /usr/local (root privileges or sudo are needed). It is possible to specify a different installation path by modifying the MAMMUT_PATH variable in the Makefile.

5.2.2 Using the tool

Mammut is able to manage a set of functionalities each one monitored by a specific logical module of the library. The following code snippet shows how to initialise the library:

```
1 #include <mammut/mammut.hpp>
2 using namespace mammut;
3 int main(int argc, char** argv) {
4  Mammut m;
5 }
```

In the rest of this section we will describe the available modules of the library with some examples of codes that use the library functionalities.

Topology module. The *topology* module allows the topology of the machine to be analysed by discovering the relationship between virtual cores (e.g., in the case of HyperThreaded CPUs on Intel multicores), physical cores and CPUs. Furthermore, it can be used to turn on/off of the cores, and to manage the idle states (C-states). Just to provide and example, the following code snippet shows how to get the list of CPUs, physical cores and virtual cores present on the machine:

```
#include <mammut/mammut.hpp>
#include <cassert>
#include <iostream>
#include <unistd.h>

using namespace mammut;
using namespace mammut::topology;
using namespace std;

int main(int argc, char** argv){
 Mammut m;
 Topology* top = m.getInstanceTopology();
 vector<Cpu*> cpus = top->getCpus();
 vector<PhysicalCores*> phCores = top->getPhysicalCores();
 vector<VirtualCores*> vCores = top->getVirtualCores();
}
```

In a similar way it is possible to gather the identifiers of the virtual cores associated with a physical core, the physical cores associated with a CPU and so forth. Furthermore, you can check which modes are present on your CPUs (e.g. sse, avx, etc...). Further details are available in the online library documentation[5].

This module allows the programmer to turn off/on virtual cores using the library methods hotUnplug() and hotPlug() on a virtual core object obtained through getVirtualCores() on the topology object. These operations require the user to have the superuser privileges. In a similar way, the topology module provides the possibility to analyze and modify the cores idle states. The getIdleLevels() method of a virtual core object can be executed to retrieve the a vector of VirtualCoreIdleLevel objects each one representing a idle state enabled for that core. Useful methods are provided to inspect a idle state and return information like the identifier of the level, its name, a textual description, the base consumed power in this state, whether it is enabled or not, and others (see the online documentation).

[5] http://danieledesensi.github.io/mammut/manual.html

Energy module. The *energy* module provides power/energy-related measurements from a running computation. Energy is monitored if the user is interested in evaluating the actual Joules consumed by a portion of code with a finite execution time. In the case of the instant consumption, usually needed in long-running computations like streaming ones, power consumption is usually the target profiling measurement to collect. In both the cases, the module is able to gather such information on the following list of multi-core architectures: Intel SandyBridge, IvyBridge and Haswell architectures. It is possible to read energy/power consumption of each CPU, of the cores on the CPU, of the DRAM controller and of the integrated GPU card. To use the features of this module, the msr kernel module must be properly loaded. Execute a sudo modprobe msr command on a superuser shell to load this module.

The power of the Mammut library is the simplicity of its interface. The following code snippet shows how to gather the actual energy consumed by a program run on a virtual core. In the code, a handler to the energy module is obtained (line 12), and from it we get an energy counter (line 14). After that, we measure a code section (lines 15-18).

```
1  #include <mammut/mammut.hpp>
2  #include <cassert>
3  #include <iostream>
4  #include <unistd.h>
5
6  using namespace mammut;
7  using namespace mammut::energy;
8  using namespace std;
9
10 int main(int argc, char** argv){
11  Mammut m;
12  Energy* energy = m.getInstanceEnergy();
13  Joules j;
14  Counter* counter = energy->getCounter();
15  counter->reset();
16  // call to a user function Foo
17  Foo(...);
18  j = counter->getJoules();
19  cout << j << " joules consumed by Foo" << endl;
20 }
```

Mammut supports different types of energy counters: *Plug counters* that count the energy consumption of the entire machine; *CPU counters* count the energy consumption of the CPUs and of individual components of the CPU. To check the types of available counters, it is possible to call the getCountersTypes() on the energy object, which returns a vector of available counter types. By default the getCounter() call used in the previous example returns the most fine-grained counter among

those available. However, it is possible to ask for a specific counter type through the getCounter(COUNTER_PLUG/COUNTER_CPUS) call. The following code snippet depicts an example of using CPU counters:

```
...
CounterCpus* c = (CounterCpus*) energy->getCounter(
    (cont.)COUNTER_CPUS);
Foo(...);
cout << "CPUs Joules: " << c->getJoulesCpuAll();
cout << "Cores Joules: " << c->getJoulesCoresAll();
if(counterCpus->hasJoulesDram())
 cout << "Dram Joules: " << c->getJoulesDramAll();
if(counterCpus->hasJoulesGraphic())
 cout << "GPU Joules: " << c->getJoulesGraphicAll();
...
```

Frequency module. The *frequency* module provides the possibility to check the current frequency "governor" and to modify it. On current Intel microarchitectures, the frequency can be changed at the CPU level (and not for individual cores). For this reason, Mammut has the concept of *frequency domain*. The following code snippet shows how to retrieve the frequency domains available on the monitored machine and the virtual cores associated to each domain:

```
...
int main(int argc, char** argv){
Mammut m;
CpuFreq* frequency = m.getInstanceCpuFreq();
vector<Domain*> domains = frequency->getDomains();
...
```

A *governor* is a specific algorithm that regulates the frequency management. A set of useful methods of the domain object are listed below:

- getAvailableGovernors() gets the list of available governors;
- getCurrentGovernor() gets the currently enabled governor;
- setGovernor(mode) sets the current governor to *mode*/

In the case of the *userspace* governor, the kernel allows the user to set a different CPU frequency. In Mammut a switch to a frequency F can be activated by using the setFrequencyUserspace(F). The list of the available frequencies can be get by using getAvailableFrequencies() call.

TaskManager module. This module manages the threads/processes running on the machine. A handler is associated with each process/thread. To get a process handler the programmer should use the getProcessHandler(pid) call on the module object. The handler must be explicitly released by executing a releaseProcessHandler (cont.)() call. Similarly, a thread handler can be obtained through a getThreadHandler (cont.)(tid) call on the ProcessHandler object. The following code shows how to obtain the module and how to use it to get a handler to the thread with identifier 9999 in the process with process identifier 9900:

```
#include <mammut/mammut.hpp>
#include <cassert>
#include <iostream>
#include <unistd.h>

using namespace mammut;
using namespace mammut::task;
using namespace std;

int main(int argc, char** argv){
 Mammut m;
 TasksManager* pm = m.getInstanceTask();
 ProcessHandler* process = pm->getProcessHandler(9900);
 ThreadHandler* thread = process->getThreadHandler(9999);
 ...
}
```

Typical operations on this module consists in configuring the process/thread pinning on the virtual cores. The following code snippet show some examples in which we control the pinning for a specific thread:

```
...
// allows the thread to run on any core on CPU 0
Cpu* cpu = topology->getCpu(0);
if(cpu)
 thread->move(cpu);
 // allows the thread to run on the physical core with id 2
 PhysicalCore *pCore = topology->getPhysicalCore(2);
 if(pCore) thread->move(pCore);
 ...
}
```

Mammut is further capable of controlling the priority of threads/processes. This is possible by using the getPriority () and setPriority () calls on a thread or process handler. The values are platform dependent.

Finally, the library allows the programmer to check which is the percentage of time spent by a process/thread on the CPU. This value can be obtained by calling the getCoreUsage() method on a thread/process handler. Further details and a more complete explanations of the API provided by the library can be found in the reference web site.

5.3 The PMLIB Library

PMLIB is a framework for power-performance analysis of parallel scientific codes [2]. The framework collects samples of a large number of power sampling devices, including external commercial products, such as APC 8653 PDU and WattsUp? Pro .Net, internal DC wattmeters, like a commercial data acquisition system (DAS) from National Instruments (NI) and, alternatively, specific designs that use a microcontroller to sample power data. Calls to the PMLIB application programming interface (API) from the application instruct the tracing server to start/stop collecting the data captured by the wattmeters, dump the samples in a given format into a disk file (power trace), query different properties of the wattmeters, etc. Upon completion of the application's execution, the power trace can be inspected, optionally hand-in-hand with a performance trace, using some visualisation tool. The PMLIB package can be found at: https :// github.com/mdolz/PMLib.

5.3.1 Setting up the PMLIB server daemon

This section describes the steps for running the PMLIB daemon on a server machine. Once the package has been extracted, the server directory needs to be copied on the server, where the physical wattmeter devices are connected. Next, the configuration file needs to be set in order to specify the server IP and port where the clients will have to connect to. It is also necessary to specify the computers that will be measured and the wattmeters available on the server, along with their connections. An example of configuration file is provided in the same package. Finally, the server instance can be started in the following way:

```
$ pm_server --start
Starting devices...
Starting device DCMeter1...        [  OK  ]
Starting device WattsUp2...        [  OK  ]
Starting server...                 [  OK  ]
Server listening at (IP '192.168.1.3', Port 6526)

$ pm_server --stop
Stopping devices...
Stopping device DCMeter1...        [  OK  ]
Stopping device WattsUp2...        [  OK  ]
Stopping server...                 [  OK  ]
```

5.3.2 Obtaining power measures from wattmeters

Once the PMLIB daemon is running on a server machine, the client part, as C/C++ library, has to be be used to instrument user applications. Listing 5.1 shows an example for measuring the sleep routine for 10 seconds. Specifically, this example is intended to measure the power and energy consumption of the target machine in the idle state using the wattmeter DCMeter1 and sampling lines from 1 to 3.

Listing 5.1: Example of usage to collect samples from the wattmeters

```
#include <iostream>
#include "pmlib.h"

int main (int argc, char *argv[]) {
 server_t server;
 counter_t counter;
 line_t lines;
 int i, frequency= 0, aggregate= 1;

 pm_set_server("192.168.1.3", 6526, &server);
 pm_set_lines("1-3", &lines);
 pm_create_counter("DCMeter1", lines, !aggregate, frequency
    (cont.), server, &counter);
 pm_start_counter(&counter);
 sleep(10);
 pm_stop_counter(&counter);
 pm_get_counter_data(&counter);
 pm_print_data_text("out.txt", counter, lines, -1);
 pm_finalize_counter(&counter);
 return 0;
}
\end{figure}
```

Once the application is compiled and run, the out.text file is generated with the power trace. As can be seen, the first columns contains the Set_id, the second indicates the time instant where the power measured were gathered. Finally, the last remaining columns stand for the power lines requested, while the last one sums the powers of such lines.

```
$ g++ example2.cpp -lpm -I${INCLUDE_DIR} -o example2
$ ./example2
$ cat out.txt
Set_id Time     Line 1     Line 2     Line 3     Sum
0   0.000000   29.766075  0.000000   17.344013  47.110088
0   0.035717   28.359804  0.000000   16.640877  45.000679
0   0.071434   29.062941  0.000000   17.812769  46.875710
...
```

On the other hand, the PMLIB client side also comes with the pmlib_info utility. This tools allows users to see the current status of the wattmeters and counters and to read, at real time, the powers gathered by these devices. Some examples of its usage are summarised as follows:

```
$ pm_info --help
Usage: pm_info -s|--server SERVER:PORT
       pm_info -l|--lines
       pm_info -c|--counters
       pm_info -r|--read DEVNAME [-f|--freq FREQ]

Options:
  -h, --help            show this help message and exit
  -s SERVER:PORT, --server=SERVER:PORT
  -l, --lines
  -c, --counters
  -r DEVNAME, --read=DEVNAME
  -f FREQ, --freq=FREQ

$ pm_info -s 192.168.1.3:6526
Dev: DCMeter1 - Freq: 30 Hz - Lines: 12
 - Computer: w3
Dev: WattsUp2 - Freq: 1 Hz - Lines: 1
 - Computer: w1

$ pm_info -s joule.act:6526 -r WattsUp2 -f 1
14:53:53 - 66.10 = 66.10
14:53:54 - 66.00 = 66.00
14:53:55 - 65.90 = 65.90
...
```

5.3.3 A module to detect power-related states

The PMLIB library and its framework can also obtain traces of the C- and P-states of each core. In order to obtain information on the C-states, the daemon integrated into the power framework reads the corresponding MSRs (Model Specific Registers) of the system, for each CPU X and state Y, with a user-configured frequency. Note that the state-recording daemon necessarily has to run on the target application and, thus, it introduces a certain overhead (in terms of execution time as well as power consumption) which, depending on the software that is being monitored, can become non negligible. To avoid this effect, the user is advised to experimentally adjust the sampling frequency of this daemon with care.

5.4 Forthcoming Features

Previous sections outlined the status of the tools and of the data of this deliverable, which is the first one in a set of deliverables that will illustrate the evolution of these tools up to the project end. We wish to point out two particular aspects not covered in the current version of the tools that we are in the process of integrating in the forthcoming releases of the tools:

- Memory usage counters are currently not included in the set of hardware counters managed by the libraries (e.g. Mammut). As of the project DoW, we will eventually use also these measures to optimise the project use cases and applications. Despite no facility has been included at the moment in Mammut library, reading a cache miss counter works in the same way as reading the power counter, except that a different "counter address" and unit conversion has to be used. We are currently working to include also the counters of interest to characterise the memory accesses in Mammut, and they will be available in the next WP4 software release.

- The tool described in Section 5.1.2 is currently available only for the FastFlow backend. The project however, will eventually produce patterns running on top of different backends. As agreed in one of the project meetings, the project tools will eventually enable to run patterned applications on top of FastFlow, plain C++ (with TBB) and possibly OpenMP, not all the patterns being suitable to be implemented on top of all these backends, nor the patterns implemented on top of backend i being necessarily interoperable with those implemented on top of backend j. However, to ensure the possibility to develop use case applications which are "backend neutral", we are designing a common pattern interface that will eventually be exposed to the patterned application programmers by all the pattern backend implementations. We are currently evaluating two different interfaces, an object oriented interface and a "functional" interface heavily relying on the new features of the new C++ standards. One of the two interfaces will be chosen in a few months and the next WP4 (and project) software release will provide a performance, energy and memory monitoring interface compatible with the chosen pattern interface.

Chapter 6

Conclusions

This deliverable described the work done in WP4 for the basic dynamic adaptation infrastructure. The set of tools described here, which will be further extended to add new capabilities and address a wider range of applications and computing environments, allows the applications to change their structure dynamically as a response to changes in their behaviour and/or computing environment (e.g. change in the load of processors). Additionally, it allows deriving of some pattern parameters (e.g. number of threads to allocate for each group of components of the patterns) that are very hard to set manually, but are necessary for the efficient execution of patterned applications, thus adding a further level of abstraction over hardware to the pattern model considered in **RePhrase**. All this allows us to address a very important issue for the software engineering of parallel applications, which is retaining good performance and/or energy consumption of application under dynamically-changing hardware environment, adding adaptation capabilities to the applications developed using the **RePhrase** technology.

We described static mapping tools (Chapter 2) that provide both alternative versions of simple parallel patterns (in this case, the *map* pattern) that do implicit mapping and replication of the data required by the pattern to the memory modes of Non-Uniformed Memory Access (NUMA) hardware and four different heuristics to decide on initial instantiation of pattern components. In this way, we "complete" the parallel application that was prepared using tools developed in WP2 and WP3 by deriving values for the missing extra-functional pattern parameters (such as the number of workers in each *map* and *farm* pattern), thus enabling its execution and dynamic adaptation by other tools developed in this work package. The next step in our dynamic adaptation mechanism is compilation of the final application code, which also allows for different versions of the same function to be compiled into intermediate representation. This is supported by the just-in-time (JIT) compiler described in Chapter 3. JIT compiler both prepares the code for dynamic adaptation and also inserts the code to actually perform adaptation (i.e. switch between the prepared versions of the function as a response to the external and internal changes) into the application. Adaptation is performed by compiling the intermediate repre-

sentation of the desired version of a function into the binary code and plugging that code into the call of the function before it is executed next time. After compilation, the application is executed and, during the execution, both dynamic recompilation is performed (as described above) and also application's parallel threads are rescheduled to the available cores, as a response to the changing load of the system. The latter is supported by the PaRLSched dynamic scheduling library, that uses reinforcement learning to collect performance counters and build a model to estimate the best mapping of threads to cores. PaRLSched is described in Chapter 4, where we have also demonstrated benefits of using the library, compared to when scheduling is left to the operating system. Finally, in order to perform dynamic adaptation by compiler and scheduling tools, we need to be able to dynamically obtain information about the execution of the application, including estimation of performance/energy consumption (and other relevant metrics) for threads/processors and reasons for possible degradation of performance/energy consumption during the application execution. For this purpose, we have added dynamic performance monitoring capabilities to the FastFlow pattern library that is used in **RePhrase** that can detect performance bottlenecks in *map*, *farm* and *pipeline* patterns. This is described in Chapter 5, together with two additional libraries, Mammoth and PMLib, that can be used to monitor generic threaded applications. We will integrate these libraries into our performance monitoring infrastructure in the future to enable monitoring of applications developed using OpenMP and Intel TBB models.

In the remainder of the project, we will extend each of the developed tools with additional capabilities and adapt them to a wider range of patterns and architectures, including heterogeneous CPU/GPU systems. In particular, we will add more specialised variants of patterns and more sophisticated thread mapping heuristics to the static mapping infrastructure, extend the dynamic scheduling library to deal with dynamic memory allocation, add the use of memory usage counters to the performance monitoring infrastructure and develop more sophisticated methods for deciding when dynamic recompilation is needed. 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] Sergio Barrachina, Maria Barreda, Sandra Catalán, Manuel F. Dolz, Germán Fabregat, Rafael Mayo, and E. S. Quintana-Ort: An integrated framework for power-performance analysis of parallel scientific workloads. pages 114–119, 2013.

[3] Enrico Bini, Giorgio C. Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Årzén, Romero Vanessa, and Claudio Scordino. Resource management on multicore systems: The ACTORS approach. *IEEE Micro*, 31(3):72–81, 2011.

[4] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *International Journal Parallel Programming*, 38:418–439, 2010.

[5] G. C. Chasparis, M. Maggio, E. Bini, and K.-E. Årzén. Design and implementation of distributed resource management for time-sensitive applications. *Automatica*, 64:44–53, 2016.

[6] 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.

[7] 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.

[8] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. `autopin` - automated optimization of thread-to-core pinning on multicore systems. In Per Stenstrom, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Computer Science*, pages 219–235. Springer Berlin Heidelberg, 2011.

[9] 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.

[10] S.L. Olivier, A.K. Porterfield, and K.B. Wheeler. Scheduling task parallelism on multi-socket multicore systems. In *ROSS'11*, pages 49–56, Tuscon, Arizona, USA, 2011.

[11] Riky Subrata, Albert Y. Zomaya, and Björn Landfeldt. A cooperative game framework for QoS guided job allocation schemes in grids. *IEEE Transactions on Computers*, 57(10):1413–1422, October 2008.

[12] H. Tembine, E. Altman, R. ElAzouri, and Y. Hayel. Correlated evolutionary stable strategies in random medium access control. In *International Conference on Game Theory for Networks*, pages 212–221, 2009.

[13] S. Thibault, R. Namyst, and P.A. Wacrenier. Building portable thread schedulers for hierarchical multi-processors: the BubbleSched Framework. In *Euro-Par. ACM*, Rennes, France, 2007.

[14] Guiyi Wei, Athanasios V. Vasilakos, Yao Zheng, and Naixue Xiong. A game-theoretic method of fair resource allocation for cloud computing services. *The Journal of Supercomputing*, 54(2):252–269, November 2010.