POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

# AN APPROACH FOR ENERGY EFFICIENT CO-SCHEDULING OF PARALLEL APPLICATIONS ON MULTI-CORE PLATFORMS

Autore:

**Simone LIBUTTI**

matr. 755283

Relatore:

**Prof. William Fornaciari**

Correlatori:

**Dott. Ing. Giuseppe Massari**

**Ing. Patrick Bellasi**

Anno Accademico 2012-2013

# Any acknowledgments?

*I do not think so.*[1]

*Legnano, July 2013*

[1]But I would like to express my sincere footnote gratitude to Giuseppe. He footnotely deserves it.

# Contents

# List of Tables

# List of Figures

Chapter 1

# Introduction

Today the majority of computing devices exploits multi-core processors. The concept is now a consolidated one; in 2001 the first general purpose processor featuring multiple processing cores on the same CMOS die was released: the POWER4 processor of IBM [12]. It exploited two cores on a single die, which had never been seen among non-embedded processors. Since then, multi-core processors have quickly replaced single-core ones on the market. That is not strange indeed, since exploiting more cores has become one of the few effective ways to achieve good speed-ups, having the rate of clock speed improvements slowed after decades of aggressive rising [4].

## 1.1 From single to multiple cores

Exploiting multiple cores on the same die leads to numerous advantages. For example, circuitry sharing leads to savings in terms of circuit area and allows the companies to create products with lower risk of design error than devising a new wider core design. Moreover avoiding signals to travel off-chip lessen signal degradation, allows the cache-coherence circuitry to operate at higher frequencies (snooping circuitry, for example, can operate much faster), and reduces power consumption. In fact, both avoiding rooting signals off-chip and exploiting several small cores instead of a several-time bigger monolithic core leads to lower energy consumption [15]. This is a very important feature given the recent trends toward mobile and embedded computing. The increasing number of cores, unfortunately, has lead to serious problems. As for CPU clock speed in single core processors, the number or cores in multi-core processors cannot reach arbitrary high values. For instance, exploiting too much cores on the same die leads to communication congestions and elevate power consumption. In recent years, the trend is moving from multi-core to many-core devices. The idea is to exploit a high number of symmetric low-power and performing processing elements (PEs) connected by a network-on-chip (NoC). NoC provides an infrastructure for better modularity, scalability, fault-tolerance, and higher bandwidth compared to traditional infrastructures. From now on, with the term *processor* we will refer to either a multi-core or many-core processor, exploiting more than one PE.

## 1.2 Applications co-scheduling on multi-core processors

The advancements in processor technology mentioned above have opened new and interesting scenarios; one of these is the study of the concept of co-scheduling, introduced in 1982 by J. K. Ousterhout [16], now more actual than ever. Having multiple PEs at disposal (or even more, in the case of many-core processors), the concept of scheduler has evolved to the concept of co-scheduler. While the main goal of a scheduler is to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously), a co-scheduler aims to schedule in parallel different tasks on given sets of resources. While lots of effort had been placed in parallel programming and resource management, few effective solutions had been proposed regarding resource utilization aware co-scheduling heuristics and algorithms. That using a random co-scheduling approach would be in general a sub-optimal solution is obvious; every application has different characteristics in terms of resource utilization, so scheduling two applications on the same PE is not a priori an intelligent choice. It depends, essentially, on which applications we are scheduling. Unfortunately, determine which applications will run on the same PE is not the only issue; a large part of today's operating systems schedulers treat each PE of a processor as a distinct device. This is not necessarily true, because there are interdependencies among the PEs. For instance, it could be thought that two memory-intensive applications should not run on the same PE. But what if we schedule them on different PEs sharing a significant part of the cache hierarchy? It would anyway lead to contention, which would slow down the execution of the applications and induce inefficient use of energy, since cores stalling due to resource contention dissipate power without making any real progress. Memory hierarchies are not always shared among all the PEs; in complex systems such as non uniform memory access devices (NUMAs), each group of processing elements (node) has a private memory, giving the opportunity to avoid the problem mentioned above. Unfortunately, the issue persists: being the number of nodes limited by technology and scalability issues, even when running light workloads we can not schedule each application on a different node. However, we can co-schedule applications on the nodes in a way that minimizes performance losses. A number of additional causes of contention can be found; in a certain way, what a scheduler should achieve is not only a fair allocation of resources, viewed as a set of distinct PEs of the applications. It should aim to fairly distribute applications' resource utilization considering the impact on the functional units of the processor.

### 1.2.1 Resource utilization aware co-scheduling

As mentioned in Sec. 1.2 on page 4, the co-scheduling problem is about scheduling $m$ applications on $n$ PEs, with $m$ greater than $n$. Looking at the modern multi-core processors, each PE is not completely isolated from the others; each of them shares certain resources with a number of other PEs. The main problem in the management of processing resources is certainly related to the cache hierarchy. Running concurrently $n$ applications characterized by the same execution time on $n$ PEs, the theoretical expected speed-up would amount to $n$ in comparison with the case of a serial execution on a single PE. In our case, however, PEs are not independent; sharing a portion of the cache hierarchy, we have levels of cache where a certain number of tasks concurrently read and write data, conditioning the respective performance. Reading data from a shared cache is evidently a problematic activity; let us think about a totally fair sharing of a cache: if a cache is shared among $n$ PEs, each PE "effectively exploits" only $1/n$ of the cache. This *virtual* reduction of the cache size leads - as the reader can certainly guess - to a massive raise of *cache miss* ratio. With the term *cache miss* we refer to the situation when a task tries to read or write a piece of memory in the cache, but the attempt fails. In this cases, to execute the action the main memory has to be accessed, which has a very long latency. What about co-scheduling two applications on the same PE? This scenario is even more complex than the previous one; in fact, in this case the performance degradation induced by shared caches could lead to scenarios where both the applications are stalled waiting for data and the PE wastes cycles, with negative effects on both performance and power consumption. This problem, indeed, comes along the one mentioned above: the resulting performance degradation, in certain cases, could even discourage concurrent execution, because speed-ups would be too low with respect of the number of PEs exploited by the system.

How is possible to minimize performance losses caused by resource sharing? These losses are typically workload dependant, being correlated to the characteristics of the single applications. In the last years, a number of solutions regarding resource-utilization aware co-scheduling had been proposed, which are briefly shown in the next section. These solutions are often based on the concept of preventive *training phase*; this term refers to the phase, at design time, when applications information is collected and analysed. Thus, having at disposal resource utilization information about the applications, a scheduler can perform optimal scheduling choices at runtime.

### 1.2.2 Prior art

In [9], a rather old but interesting work, an approach is proposed that does not require a training phase. Their approach is based on the identification, at runtime, of groups of activities characterized by high mutual interaction. The idea is that applications which frequently access to the same area of data - thus sharing the data - are probably characterized by strong mutual interactions. Here the managed resources amount to a single-core processor but the article shows how, even twenty years ago, the idea of paying attention on which groups of applications were to be co-scheduled together (in this case, co-scheduled in the same slice of time) to reduce performance losses was present in literature. Another interesting work focuses on bus utilization to better co-schedule applications in symmetric multiprocessors systems (SMPs)[1]. This approach, proposed by [5], aims to reduce bottlenecks caused by system bus congestion. They demonstrate that, exploiting thread bandwidth request information collected during design time, the scheduler can avoid bus overuse, thus reducing performance degradations running heavy workloads. The scheduler is validated running two different benchmarks: the former is characterized by heavy bus utilization, the latter by very light bus utilization. The experimental platform is a dedicated 4-processor SMP with Hyperthreaded Intel Xeon processors, clocked at 1.4 GHz. It is equipped with 1 GB of main memory and each processor has 256 KB of L2 cache. The system bus of the machine run at 400MHz. The operating system is Linux, kernel version 2.4.20. Their approach lead

[1]Here we refer to an architecture composed of two or more identical processors connected to a single shared main memory. Communication is based on system bus or crossbar.

to a maximum throughput increment of 26% with respect to the standard Linux scheduler. Bus bandwidth utilization information is collected using performance counters, which is the main approach in these works. One could wonder which counters are best-suited to be exploiting during the schedule phase; in [8] several performance counters are analysed, trying to answer to this question. Their study, carried out on a system exploiting two dual-threaded processors, is centred on a comparison between four schedulers: RFUS scheduler (register file utilization scheduling), RFCS scheduler (register file conflict scheduling), DCCS scheduler (data cache conflict scheduling), IPCS scheduler (IPC-based scheduling), and RIRS scheduler (ready in-flight ratio scheduling). Each of these schedulers exploits different performance counters to characterize the applications and compute the optimal scheduling choice. Running heterogeneous workload mixes, they demonstrate that the maximum performance losses of RIRS amounts to 2%, while it reaches 10%, 13%, 11% and 14% with DCCS, RFUS, RFCS, IPCS respectively. Thus, general information as the number of ready and in-flight instructions resulted much more consistent than first level cache misses and register file utilization.

In [13], the focus has shifted to memory utilization. Here two important concepts are introduced: first of all, running several benchmarks from SPEC CPU 2006 benchmark with a modified Linux scheduler, they show that co-scheduling applications characterized by different memory usage can lead to performance and energy efficiency improvements. Second, they choose energy-delay product (EDP) as evaluation metric for their scheduler. Their extension of Linux 2.6.16 kernel to allow memory-utilization aware scheduling granted a maximum EDP reduction of 10% with respect to the standard Linux scenario, using an IBM xSeries 445 eight-way multiprocessor system (eight Pentium 4 Xeon Gallatin processors with 2.2 GHz each processor). The system consists of two NUMA nodes with four two-way multi-threaded processors on each node. The choice of the EDP metric to evaluate the scheduler is important because [11] shows that EDP is a relatively implementation neutral metric that causes the improvements that contribute the most to both performance and energy efficiency to stand out. In other words, using EDP we have at disposal a single metric which characterize both performance and energy efficiency.

The works mentioned above aim to demonstrate that certain resource usage statistics are well suited to be taken into account during the co-scheduling phase. The major weakness of these works is that they take into account only one type of resource; each work demonstrates that the others are not taking into account resources whose overuse had been proven to be cause of performance degradation and higher power consumption. To respond to this issue, in [14] the concept of task activity vector is introduced. An activity vector describes to what degree a running task utilizes various processor-related resources. The dimension of this vector is equal to the number of resources we want to consider. Each component of the vector denotes the degree of utilization of a corresponding resource. The goal of their scheduler is to co-schedule applications so that each resource in the system is used by applications having very different usage ratios on that resource. Doing so, for example, two applications which are characterized by a high degree of utilization of a certain resource would not be co-scheduled on the same PE. To investigate the effects of resource contention and frequency selection, they choose a 2.4GHz Intel Core2 Quad Q6600 processor. When suitable combinations of tasks can not be found, contention is mitigated by frequency scaling. This allows to achieve up to 21% EDP saving across various applications from SPEC CPU 2006 benchmarks.

In the last years, several approach have been proposed. In our opinion, however, there are still important issues to resolve:

- Real implementations are not always provided. In fact, validation is often performed with the aid of simulators.
- None of the works featuring a real implementation is portable. New scheduling policies are implemented with the aid of modified Linux kernels.
- The only work which takes into account arbitrary amounts of resources to compute possible co-scheduling combinations does not provide a method to evaluate which resources are the best suited to be modelled; additionally, users cannot define which resources are the most important to monitor during co-scheduling phase, nor can find out if any of them needs actually to be valued more than the others.
- Architecture-dependant information is often required for training and evaluation purposes (e.g. processor floorplans).
- A complete and standardized flow spanning from application characteristics collection to energy efficiency/performance improvements evaluation is not provided.

Currently, the literature lacks the design of a scheduling policy taking into account multiple resources which the user can effectively prove to be highly correlated with power consumption and performance. This approach should be characterized by portability and adaptability on any multi-core device. Moreover an entire flow is needed, spanning from application characteristics collection to performance/energy efficiency improvements evaluation to guide the user through the training phase and the execution of workloads on a real system.

## 1.3 Objectives of this thesis work

The goal of this work is to prove that a smart resource utilization aware co-scheduling policy can lead to a wide variety of benefits, such as performance speed-ups, energy efficiency improvements, reduction of thermal hotspots. The policy has been designed and implemented as an extension of one of the policies exploited by the BarbequeRTRM [7], a highly modular, extensible and portable run-time resource manager developed at DEIB - Politecnico di Milano - under the context of the European Project 2PARMA [17] which provide support for management of multiple applications competing on the usage of one (or more) shared MIMD many-core computation devices. We prove that, focusing not only on which are the best tasks to schedule but also on how to co-schedule them, further improvements in performance and energy efficiency can be achieved with respect to the current BarbequeRTRM case. We validate the policy, codename CoWs. (CO-scheduling WorkloadS), on two different platforms: the first one featuring a 2nd Generation Intel Core i7 Quad-core Processor (eight cores with hyper-treading), the second a NUMA device featuring four AMD 10h Family Opteron 8378 Quad-core processors with distinct cache hierarchies. The validation is based on the comparison of energy-delay product of different workload scenarios in Linux, the resources being managed by Linux standard scheduler, BarbequeRTRM framework, BarbequeRTRM framework with CoWs support. The benchmark used for the validation is the PARSEC benchmark v.2.1 from Princeton University [19]. We also design a standard flow which drives the user through application creation and design space exploration, training phase, scheduler optimization with respect to the user workload and EDP evaluation.

## 1.4 Organization of the thesis

**In Chapter 2** the BarbequeRTRM framework will be introduced. A high-level view of the framework will aid the reader to better understand the advantages offered by this resource manager, ad why it is a good starting point to apply our co-scheduling theories. The policy which we will extend, codename YaMS, will be introduced. Its main concepts explanation will be followed by an analysis of its weak spots in terms of resource utilization aware co-scheduling, and the first basic ideas on how an extension to this policy can bring benefits on performance and energy efficiency will be introduced.

**In Chapter 3** the concept of resource utilization aware co-scheduling will be defined, focusing on which application characteristics one should exploit in order to achieve an efficient scheduling. An analysis of architectural and non-architectural events will lead to the selection of the counters which are best-suited to our purposes. Then CoWs scheduler will be designed, pointing out its interaction with the BarbequeRTRM framework. The entire flow from application creation to results evaluation will be defined, along with the role of all the tools exploited.

**In Chapter 4** the implementation will be validated, focusing on each phase of the work and on the interaction among the different tools exploited to collect resource utilization information. This chapter will also guide the reader from the creation of an application to its scheduling on the PEs, passing through the integration with the BarbequeRTRM framework and the resource-usage information collection. Then, the results will be discussed. We will see how the BarbequeRTRM framework can achieve better results in resource management than the standard Linux scheduler, and how CoWs support is able to improve energy efficiency and performance on even higher levels.

**In Chapter 5** We will discuss why CoWs policy is able to achieve such results, and what could be done in the future to further improve its performance.

**In appendix A** The main tools exploited during our work will be introduced, along with the main advantages we gained using them during our work.

**In Chapter 6** an abstract in Italian language is provided.

Chapter 2

# The BarbequeRTRM Framework

Given the recent trend toward many-core devices, resource utilization has become an important research area. As mentioned in Chap. 1 on page 3, this devices could be thought of as general purpose GPUs composed by increasing numbers of symmetric processors. Exploiting single instruction multiple data (SIMD) programming model, this processors are able to execute the same code, concurrently, on huge amounts of data. Having lots of resources at disposal, as mentioned above, one feels the increasing need to get help for resource managing and allocation, provision of QoS and similar services and so on. This is, from a very high level point of view, the goal of a resource manager. The next section will introduce the BarbequeRTRM framework, a resource manager by Politecnico di Milano.

## 2.1 Overview

The BarbequeRTRM is a framework being developed at DEIB - Politecnico di Milano - under the context of the European Project 2PARMA [17] and it has been partially funded by the EC under the FP7-ICT-248716-2PARMA grant. It features an highly modular and extensible run-time resource manager which provide support for an easy integration and management of multiple applications competing on the usage of one (or more) multi-core processors and shared MIMD many-core computation devices. Among its numerous features, it provides:

❏ Support for pluggable policies for both resource scheduling and the management of applications coordination and reconfiguration.

❏ Automatic application instrumentation to support Design-Space-Exploration

(DSE) techniques.

- ❏ Easy porting of the framework on different platforms and integration with specific execution environments such as the Android run-time. This is provided by the framework design itself, consisting in platform abstraction layers, built on top of Linux kernel interfaces (see Fig. 2.1).

- ❏ Easy coding of resource management policies which support an optimized assignment of resources to demanding applications considering;
  - Application properties (e.g. priorities, requirements, operating modes).
  - Resources availability and state (e.g. power and thermal conditions).
  - Tunable run-time optimization goals (e.g. power reduction, energy optimization, performance maximization).

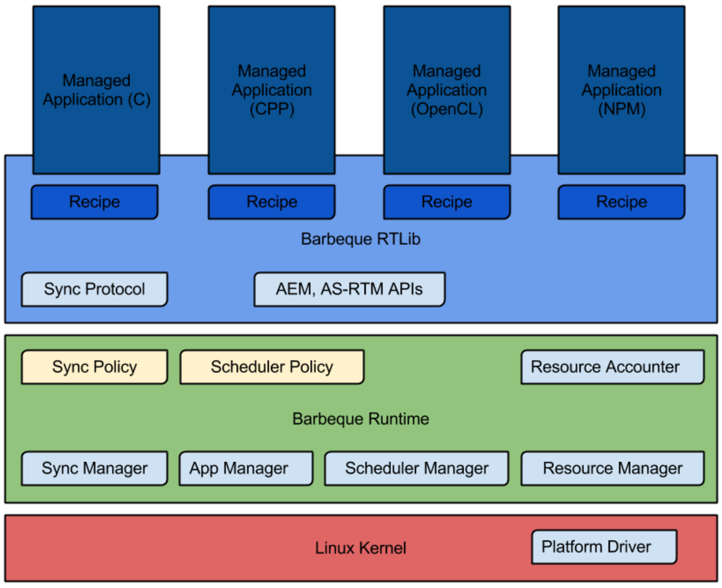


Figure 2.1: BarbequeRTRM framework

Th BarbequeRTRM runtime is the core of the framework: a layer masking the Linux kernel, the runtime is the key for portability of and comprehends the resource manager and the scheduler policies. A suitable library, RTLib, provides to the application the interface towards the resource manager, with support for profiling and Design Space Exploration (DSE). Aim of this framework is to manage resources granting fairness of the resources allocated to the applications, achieving in the meanwhile the desired quality of service and leading to performance improvements and energy efficiency executing critical applications. The intended usage of the framework is to execute critical applications on a managed device. The framework itself is executed on a set of resources called *host device*, along with the *unmanaged applications*, that is, the applications which are not managed by the resource manager. The *managed device*, which is an external device or a system partition isolated from the host one, is exploited to run only the *managed reconfigurable applications*.

## 2.2 Reconfigurable applications

A key concept that the framework aims to exploit is related to the reconfigurable applications. The idea is that an application can be *reconfigured* during runtime, choosing among one of its possible execution configurations to execute. These configurations, which are user defined, are characterized by different resources usage (currently memory and CPU usage), thus providing different levels of performance and quality of service and granting the application the capability to be executed in a wide range of scenarios and possible system loads. To facilitate the development of run-time reconfigurable applications, the RTLib provides an Abstract Execution Model (AEM). The AEM ”embeds” the execution flow of the *managed applications* in a way that let the BarbequeRTRM to manage their life-cycle. The structure of an application is split in one or more Execution Contexts (EXC), which can be defined as ”schedulable tasks”. Having more EXC in the same application can come from the need of schedule parts of the application with different priorities and resource usages. This is made possible associating many execution modes to each EXC. This modes, called application working modes (AWMs), define all the possible configurations of the application in terms of CPU and memory usage. Thus, an application can be run (”cooked”) in many different ways. The AWM choice is based on the current system status, the application needs in terms of QoS, and so on. The application AMWs are placed in a container, called recipe (see Fig. 2.2). The priority levels for the *managed applications* are in *[0, n]*, where a priority level amounting to zero is exploited to characterize critical applications, while a priority level in *[1,n]* denotes a *best-effort application*, that is, an application without strict and critical requirements.

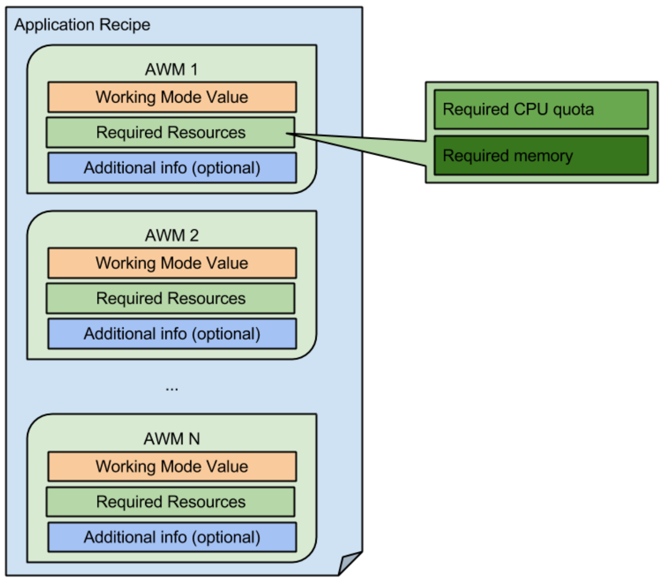


Figure 2.2: Application recipe, containing different AWMs

To integrate a BarbequeRTRM-managed reconfigurable application is quite easy. The application must define its execution contexts (EXCs) as intstances of a C++ class derived from a specific C++ base class provided by the RTLib, i.e., Bbque-EXC. For each EXC a recipe, that is, the XML file describing the AWMs, must be provided. The BbqueEXC class has five private methods to implement, which are described in Tab. 2.1.

Once these functions have been implemented, the EXC are registered and the application is integrated with the framework. Fig. 2.3 shows the application execution flow. After the setup phase, the application is configured and run. It there are no resources to allocate to the application, it is suspended. At the end of the processing, if the application has finished release method is called. Else, monitor method computes QoS and application specific objectives to provide a feedback on the current AWM performance. This feedback will be exploited during the nex workload processing phase to help AWM selection.

After this simple and abstract introduction of the framework, the reader should have achieved a general idea of its main concepts. In the next section, YaMS scheduler will be introduced. For more information on the framework itself, consult the project website [7].

| | |
|---|---|
| `onSetup()` | `Function to be implemented with initialization and thread creation stuff.` |
| `onConfigure()` | `Called when a AWM has been assigned for the first time, or a change of AWM has been necessary.  Here must be placed the code to setup the execution of the next runs, taking into account the set of resources related to the AWM.` |
| `onRun()` | `This is the entry point of our task.  Here must be implemented the code to execute a computational run.  It is very important each run would last some tens, or at maximum a few hundreds of milliseconds, in order to make the task interruptible with a "reasonable" time granularity.  This would prevent the application from being killed by the RTRM.` |
| `onSuspend()` | `There are no resources for the EXC. Its execution must be stopped.  Here should be coded whatever is needed to leave the application in a safe state.` |
| `onMonitor()` | `After a computational run, the application may check whether the level of QoS is acceptable or not.  In the second case, some action could be taken.` |

Table 2.1: BarbequeRTRM: functions to implement in order to integrate an application

## 2.3 YaMS: a multi-objective scheduler

*YaMS* is a modular multi-objective scheduler designed and implemented as a plug-in module of BarbequeRTRM. The resources available to the system are split into sets called *binding domains* (BDs); once partitioned the system resources into BDs, the goal of this scheduler is to bind the applications to the BDs. For each priority, a list of the ready applications' possible bindings is created. Its size, obviously, amounts to the number of BDs times the number of AWMs related to the ready applications at the current priority (see Eq. (2.1) on page 17). Strictly speaking, for all priorities, for all applications, all AWMs are to be evaluated on all the BDs.

$$sizeof(bindings(priority\ p)) = sizeof(BDs\_list) * sizeof(AWMs(p)) \tag{2.1}$$

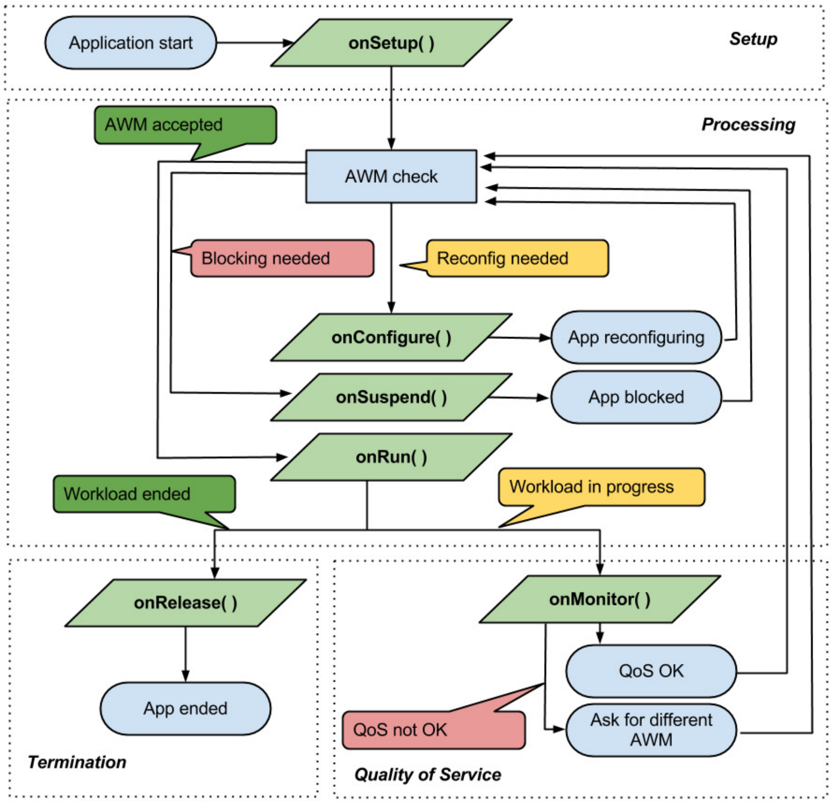


Figure 2.3: Application execution flow

The *binding*, that is, the atomic unit under analysis is the single couple *[AWM - BD]*, meaning a particular AWM if bound on a certain BD.

### 2.3.1 Resource binding domains

Let us further clarify the concept of *resource binding domain*, which from now on will be referred to with the term *binding domain*, or BD: BarbequeRTRM provides logical partitioning of the resources. The partition layout is arbitrarily decided by the user, according to his particular needs; for example, a managed device consisting of 16 CPUs can be divided in 4 BDs, 16 BDs, or even divided asymmetrically into heterogeneous-sized BDs (see Fig. 2.4). In any case the scheduler's job is to decide, for each application ready to run, which BD should it be allocated on.

Inside the list containing all the AWMs of the ready applications at the same priority, the optimization process takes place. The optimization is, strictly speaking, an imposition of relative priorities in the list. Every application has multiple

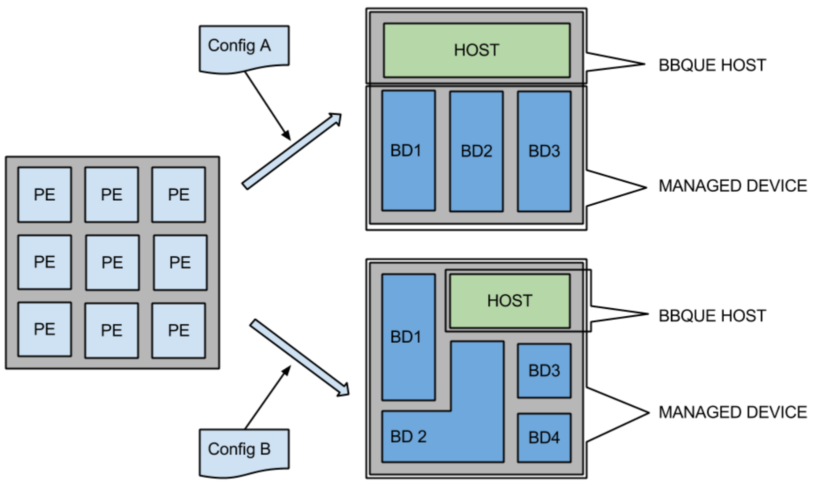


Figure 2.4: Logical partitioning of the resources

AWMs present in the list and, obviously, only one will be chosen. So, AWMs must be ordered by importance. The evaluation is performed taking into account multiple goals. The set of goals, easily extendible, currently features the metrics shown in Tab. 2.2

Each combination [AWM,BD] is evaluated taking into account the above mentioned goals (see Fig. 2.5). Then, the AWMs of the current priority are ordered by the value obtained by the evaluation and scheduled on the resources. Obviously, AWMs related to a scheduled application (that is, an AWM referring to the same application has already been scheduled) are skipped.

### 2.3.2 Scheduling contributions

Optimization performed by YaMS takes into account bound AWMs which, by definition, are AWMs associated with a BD. As mentioned in Chap. 1 on page 3, this is the key concept in co-scheduling: migrating from a pure time model to a time-space model. Evaluating an AMW not per se, but in association with a location in the resource set is a great idea, following the concept previously delineated. The problem is: is it really a resource-aware approach? Let analyse the single contributions of the scheduling policy.

| | |
|---|---|
| AWM value | An user-defined value. It represents the importance of this AWM only among its competitors in the related application recipe (in other words, among the AWMs related to the same instance of the application). |
| Reconfiguration | Represents the time overheads to reconfigure an already running application (if any) from its current AWM to this AWM. |
| Fairness | Represents the degree of fairness of this AWM. For instance, rewarding a heavy resource-demanding AWM when there are currently several applications ready to run would not be a fair choice. |
| Migration | As mentioned above, each AWM is evaluated on each BD. Similar to reconfiguration metric, it represents the time overheads to migrate an already running application (if any) from its current BD to this BD. |
| Congestion | Represents the congestion contribution of an AWM-BD choice. The goal is to balance the load among the different binding domains. |

Table 2.2: YaMS: currently supported metrics
Main features

#### 2.3.2.1 AWM value

*AWM Value* is an application dependant metric. That is not a flaw, since giving a resource aware-flavour to a policy is not about avoiding all references to pure application dependant values. In fact, being able to value an application above another regardless of its placement in the system is often crucial. This value, though, is definitely not exploitable from a pure resource-aware point of view, being it capable to discriminate an AWM importance only among the AWMs contained in the same recipe. To be more clear, the *AWM value* is an indication of how much an AWM is desirable from the user point of view. So, AWMs of two distinct applications or of two instances of the same application cannot be confronted in term of value. In fact, the metric is exploited to characterize performance or QoS (or whatever is the metric chosen by the user) of an AWM with respect to the other AWMs contained in the same recipe.

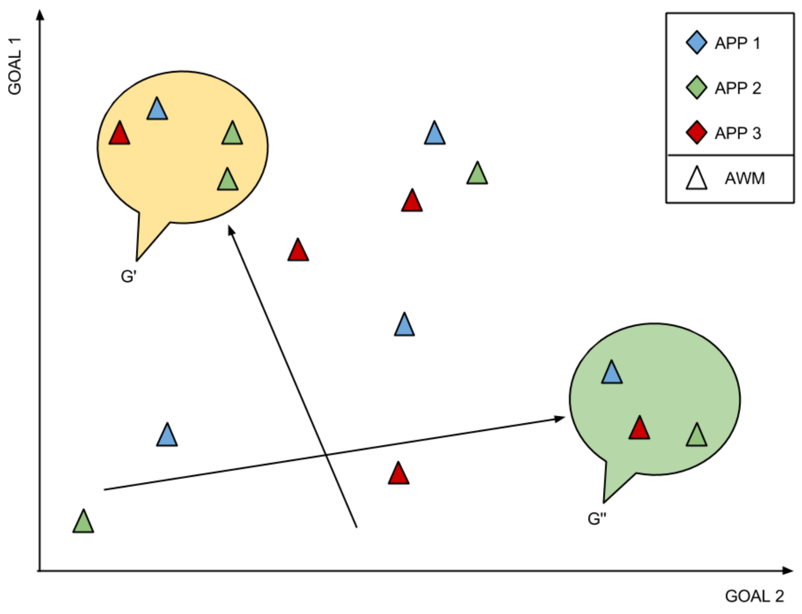


Figure 2.5: YaMS multi-objective AWM evaluation

#### 2.3.2.2 Reconfiguration overhead

This contribution is one of the many hindrances to a resource-aware policy. Often, better configurations than the current running one are rejected due to reconfiguration penalties. In any case, one can not avoid to take this goal into account. Rather, a good balance between theoretical reconfiguration benefits and reconfiguration penalties is to be found, and this is exactly what *YaMS* does. The only problem is the definition of the concept of benefit; in this case, we should think about it in terms of ”union of the other goals”. As we will show in this section, the other goals are not fully able to characterize resource usage of applications. This lead to a sub-optimal usage of this metric: the balance,Main features in this case, is more about reconfiguration penalties versus theoretical speed-up and system load balancing improvement induced by that reconfiguration.

#### 2.3.2.3 Fairness

*Fairness* metric represents the degree of fairness of this AWM. This is an important and required metric, and it relates more to resource management than to resource-aware scheduling. The system has the duty to give a fair amount of resources to application characterized by a certain level of priority. So, this metric is not a significant one from our point of view, but but nevertheless crucial to a good quality scheduling.

#### 2.3.2.4 Migration overheads

This is a metric that, from our point of view, is strongly similar to *reconfiguration*. Application reconfiguring and migrating are means to react to certain environmental changes. The application is reconfigured in its current BD or moved to another BD because something happened, and this led to an environment variation that caused this operations to be taken into account. However, migration metric Main features is somewhat more important from our point of view, since the main goal of a co-scheduler is to compute where (which BD, in this case) an application should be run. So, deciding whether to migrate an application to a more convenient BD or not is a crucial question. This is indeed the most interesting metric regarding resource aware co-scheduling among the ones currently exploited by *YaMS*. Unfortunately, the main goal of a resource-aware scheduler is not deciding whether to migrate an application or not, it is deciding where to migrate the application to. So, this metric is not a meaningful one to characterize resource utilization features of an application.

#### 2.3.2.5 Resource congestion

As mentioned in Chap. 1 on page 3, schedulers tend often to treat each PE of a processor as a distinct device, and this is absolutely the case. This metric is exploited to achieve load balancing, trying not to overload a BD. The problem, here, is that the applications are viewed as atomic objects. Could an application be "heavier" than another under a certain point of view? Running two instances of an application A on the same BD is equal to run an instance of A and an instance of another application B, even if the two applications require the same resources? This is what we want to characterize, and from our point of view this metric is not a great help.

### 2.3.3 The algorithm

From a very general point of view, *YaMS* scheduler's behaviour is summarized as follows:

❏ For the priority level currently under analysis, the list of ready applications is considered.

❏ For each application, the set of AWMs is evaluated, considering at the same time the binding of the resources against every BD defined in the *managed device*. Therefore, the evaluation is performed for each *binding* (e.g. each $[AWM, BD]$ pair), by computing a metric as a result of the combination of the following contributions:

  – AWM base value.

  – Reconfiguration overheads due to an AWM change.

  – Fairness index of the assignment of AWM requested resources taking into account current system load.

  – Overhead in migrating the application among BDs.

  – Congestion impact of the *binding* choice on the binding domain itself.

❏ AWM list is ordered by the metric computed in the previous step.

❏ Applications are scheduled one by one according to the *binding* with the highest metrics, skipping those already scheduled having competitors in earlier positions in the list.

❏ Continue with the ready applications of the next (lower) priority level.

The process explained above, which is shown in Fig. 2.6, has a main limitation: the selection of the best AWM is not separated from the selection of the best AWM placement. This causes the list to explode: in a $x$-BD-sized *managed device*, co-scheduling $y$ applications (for simpleness sake, at the same priority) with $z$ AWM each causes the evaluation of $x * y * z$ bindings. The effectively selected bindings will evidently be, in the best case, $y$ (being $y$ the number of applications to schedule). In the quite frequent case where the first AWM occurrence for each application is selected (earlier AWMs, in reality, may be skipped due to full resources), $x * z$ bindings are useless.

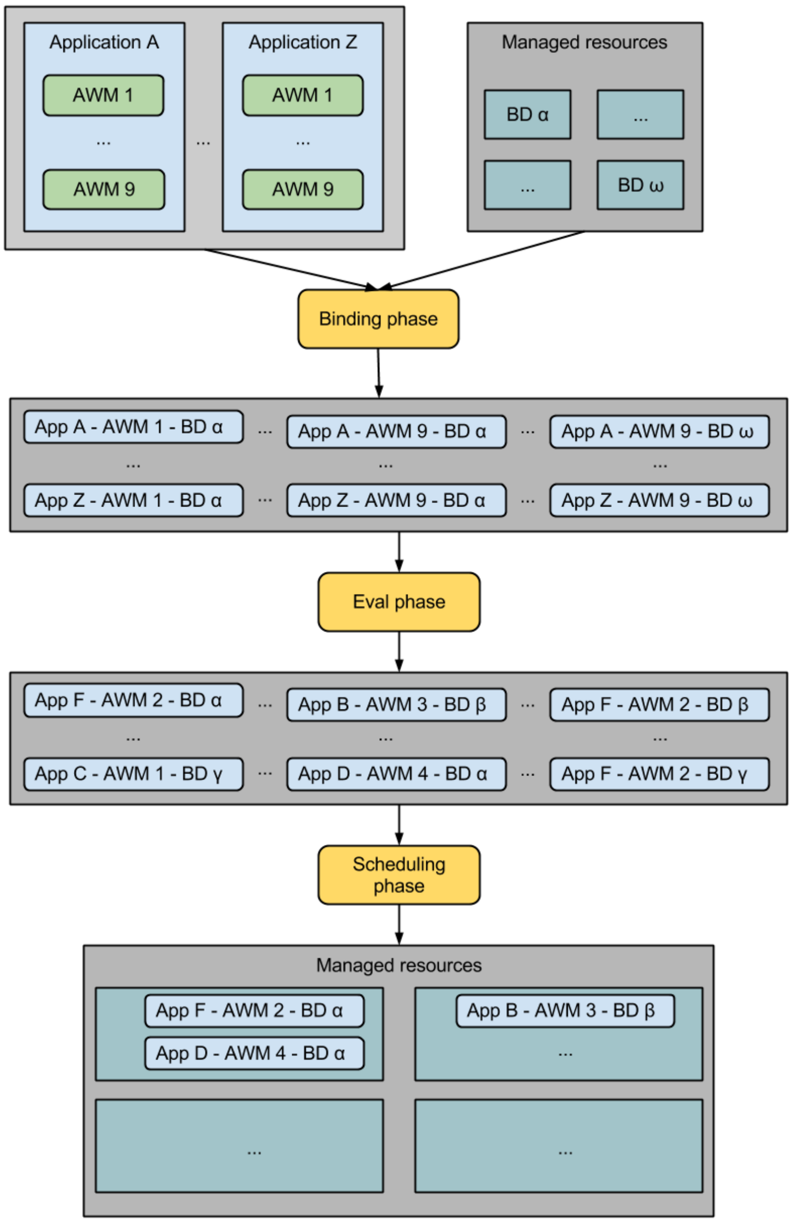


Figure 2.6: YaMS scheduler behaviour

## 2.4 Our proposal

In this chapter we have outlined the strong points of the BarbequeRTRM framework and of *YaMS*, its multi-objective modMain featuresular scheduler. Results show that this resource manager is indeed valid and innovative [6]. Unfortunately, we have seen how resource-aware co-scheduling is only partially taken into account in the scheduling of application. To prove our thesis, that is, the fact that resource utilization aware co-scheduling is needed in these systems to further improve performance and energy efficiency, our goal will be to design and implement an extension of *YaMS* to give a resource-aware flavour able to bring benefits in terms of performance and energy efficiency in the execution of parallel applications on multi-core processors. We refer to this policy with the acronym *CoWs*, which stands for *CO-scheduling WorkloadS*. At this point, however, an actual implementation seems not to be enough. These managed devices are often exploited, especially but not only in the case of many-core devices, to run specific jobs with specific quality, time and many other requirements. While best effort applications can be run with relatively little concern of their requirements, critical applications have typically strict requirements but, most of all, are known. So, we can manage to perform a training phase time during which extracting resource usage and other useful information from the applications to be run. The design and implementation of *CoWs* will come along the description of an entire flow that consists of:

- ❏ Application creation, as described in Sec. 2.2 on page 15
- ❏ Design space exploration:
  - Resource utilization statistics extraction
  - Best configurations computing - AWM creation
- ❏ Scheduler parameters calibration with the desired workloads
- ❏ Evaluation of execution time and energy efficiency

Most of these activities will exploit the BarbequeRTRM framework even to collect statistics. Others will make use of external tools, which be introduced in Appendix A on page 81. Before the introduction to these tools, Chap. 3 on page 27 will present a study on what resource utilization statistics were chosen to represent application characteristics, and why.

Chapter 3

# CoWs: Co-scheduling Workloads

In the first part of this chapter a theoretic study of the effects of co-scheduling choices on performance will be presented. After that, we will focus on determining which statistics are best suited to effectively characterize resource utilization of an application. In the second part of the chapter, *CoWs* scheduler will be designed along with the flow which guides the application developer from design space exploration to EDP evaluation. The design of the policy will be delineated. We will describe how to choose the best binding domain for an application with a set of associated statistics. We will describe how *YaMS* scheduler will be extended to support resource-aware co-scheduling.

## 3.1 Resource-aware co-scheduling

Let us redefine the concept of *binding domain* (BD). It should be considered as a group of processing elements that share one or more levels of the memory cache hierarchy. For example the Intel i7 processor, that is one of the processors we tested the scheduler on, has four cores. Each core executes two threads (hyper-threading) and features private L1 and L2 caches. The L3 cache is shared among all the cores (see Fig. 3.1). A natural choice, here, is to select the single group [double-threaded core, L1, L2] as a binding domain. So, this group of resources consists in a core which can provide up to 200% CPU usage and a double-level cache hierarchy distinct from the other BDs. This lead to a new concept, as the reader may have just guessed: the concept of *relative last level cache* (R-LLC). In fact, choosing BDs such as these, each core sees its L2 cache as last level cache. From now on, when talking about LLC information of a BD, we will refer to its R-LLC. In the case of Intel i7 processor, so, we have a total of four BDs. However, only three of them are really available. The reader should take in mind that the BarbequeRTRM is a resource manager. Generally, the framework is executed on the system processor and manages a set of external resources such as hardware accelerators or general purpose GPUs. In our case, the system is partitioned so that BarbequeRTRM runs on the first two hardware threads, actually managing the remaining three binding domains (see Fig. 3.2). To be more clear the BarbequeRTRM framework, in this configuration, operates so that the first BD is reserved for the execution of the framework itself and the *unmanaged applications*. *Reconfigurable applications*, that is, the ones we need to manage in a very efficient way, are scheduled on the three-BDs *managed device*.

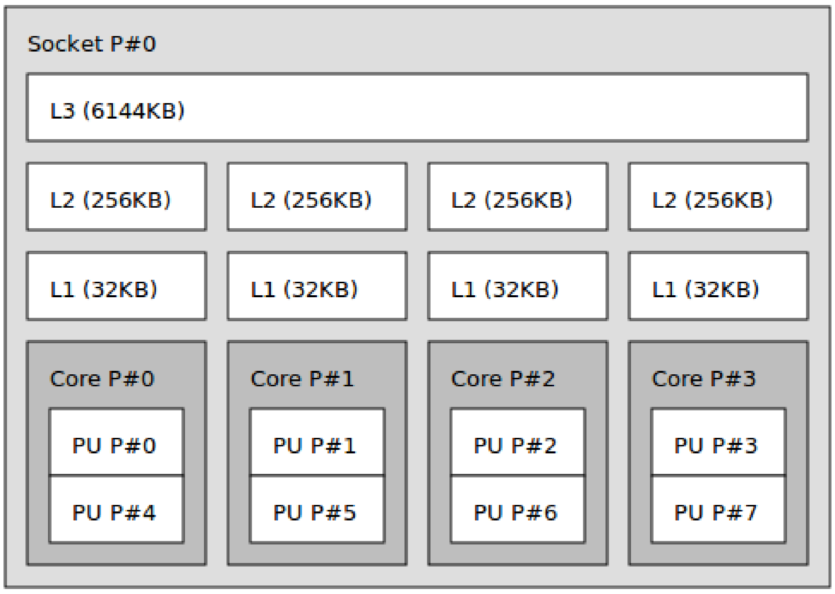


Figure 3.1: Intel i7 processor socket graphical representation

Once partitioned our *managed device* as a union of independent binding domains, two goals to achieve:

- ❑ Minimization of performance losses due to co-scheduling applications on the same BD.
- ❑ Allocation of expected functional units utilization so that the device is exploited in a fair way.

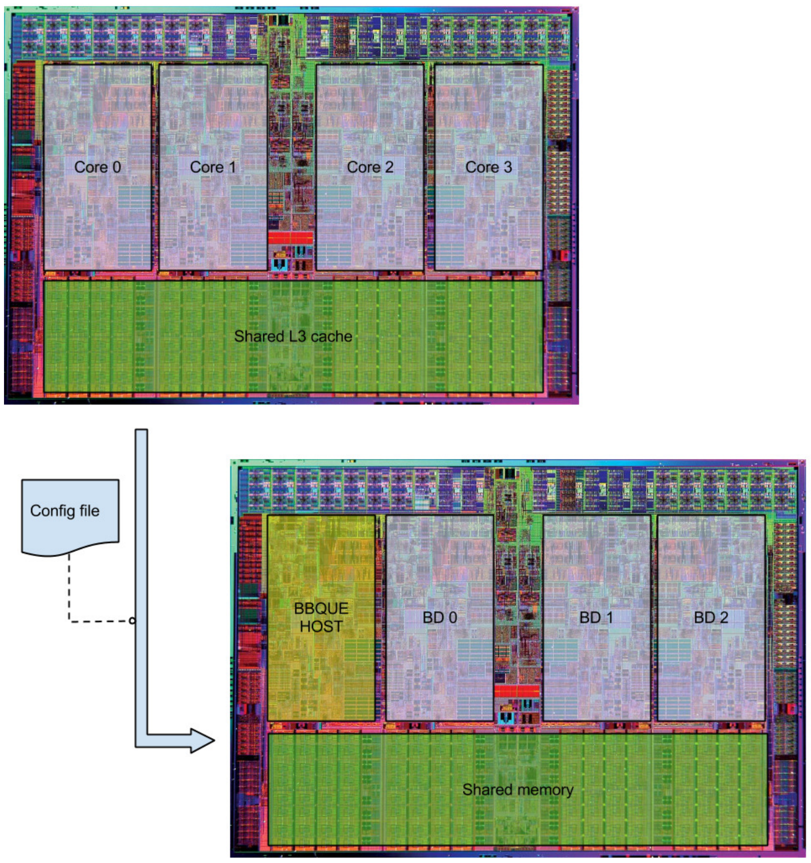


Figure 3.2: Intel i7 processor viewed as combination of a *unmanaged device* and a three-BDs *managed device*

### 3.1.1 Performance counters exploitation

As mentioned in Sec. 1.2.2 on page 6, applications' resource utilization statistics are sampled with the aid of performance counters, which are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. In order to extract information from an application execution we must select a number of performance counters to sample. For reader's sake, a few words about the aim of this phase must be spent. We are not trying to understand what type of application configuration (memory and CPU quota available to the application, parallelism level and so on) guarantees the best performance. By extending *YaMS* scheduler adding resource-utilization computation, it must be understood that we will have at disposal a list of user-defined AWMs, and or goal will be the computation of the best co-scheduling in terms of energy efficiency and minimal performance loss. So, we are not checking whether a certain resource utilization ratio has impacts on the best execution time of an application, but to find out if a certain resource utilization ratio could be related to energy consumption (in order to characterize energy efficiency) and performance degradation of applications co-scheduled on the same BD. The real effort will be put on the former problem. In fact, the latter is an already known and resolved one. As shown in [13] and [14], performance degradation is highly correlated to the number of *last level cache misses* (LLCM) of an application. In fact, LLCM penalties are very long and cause heavy performance losses, especially in the case of multiple memory-bound applications seeing the same cache hierarchy. The reader could wonder if the number of misses on the smallest cache in the hierarchy, the L1 cache, could be as informative as LLCM. This is another already-known question. In fact, [8] demonstrates that L1 cache misses are an inconsistent metric if used to make co-scheduling decisions. So, chosen LLCM metric for performance losses evaluation, we will now concentrate on the energy consumption issue.

The first step to determine the best performance counters to sample during an application execution to characterize power consumption and/or functional units utilization (the most important aspects from our point of view) is to compile a list of suitable - and available - counters. Thus, our primary goal is to compute the correlation between these counters and the power consumption of our applications. Additionally, we ask that these counters cover a large amount of resources. In fact, we want at least a counter for memory utilization, one for the stalls, one for instructions, and one specific counter for floating point operations (see Tab. 3.1).

The test is performed on a 2nd Generation Intel Core i7 Quad-core Processor (eight cores with hyper-treading). According to this processor developer's manual [1] and to [10], a list of suitable performance counters to evaluate was selected (see Tab. 3.2 to Tab. 3.5).

At this point, we chose the most representative performance counters among the ones listed above. The aim, here, is to maximize the coverage of the performance counters. We certainly can not afford to keep track, at runtime, of high numbers of counters; so, we elected to chose a limited set of counters able to characterize as much resources and founts of contention as possible. In the Memory operations group, we selected *last level cache misses*, which we need to characterize the memory boundedness of the applications. For the *Stalls* group, *resources stalls* is evidently the most suited counter to represent wide amounts of stalls (and resource

| Group | Related information |
|---|---|
| Memory utilization | Represents memory bottlenecks, number of transitions from and to memory, memory boundedness of an application |
| Stalls | Performance bottlenecks covering various resources, e.g. OOO buffer, physical registers stalls, RAT stalls ecc. |
| Instructions | Retired/issued instructions and micro-operations, to characterize the real frequency of an application, its length in terms of executed instructions, the percentage of "heavy" instructions |
| Floating point operations | Floating point operations are among the most slow and energy consuming ALU operations. |

Table 3.1: Candidate performance counters groups

| Counter | Description |
|---|---|
| MEM_UOPS_RET | Qualify any retired memory uops. |
| LLCM | Last level cache misses. |
| MEM_LOADS_RET | Counts the number of instructions with an architecturally visible load retired on the architected path. |
| MEM_STORES_RET | Counts the number of instructions with an architecturally visible store retired on the architected path. |

Table 3.2: Candidate memory related counters

| Counter | Description |
|---|---|
| ILS_STALLS | Instruction Length Decoder is stalled. |
| RES_STALLS | It counts the number of Allocator resource related stalls. For example, stalls from register renaming buffer entries, memory buffer entries.<br>In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations. |
| RAT_STALLS | Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe. Cycles when partial register stalls occurred Cycles when flag stalls occurred Cycles floating-point unit (FPU) status word stalls occurred. |
| UOPS_STALLS_DEC | Counts the cycles of decoder stalls. |

Table 3.3: Candidate stalls related counters

| Counter | Description |
|---|---|
| UOPS_ISD | Issued micro-operations. |
| UOPS_RET | Retired micro-operations. |
| INSTR_RET | Retired instructions. |

Table 3.4: Candidate instructions related counters

| Counter | Description |
|---|---|
| FP_EXE_SSE | Counts number of SSE FP packed uops executed. |
| FP_EXE_X87 | Counts number of X87 uops executed. |

Table 3.5: Candidate floating point operations related counters

utilization) sources. In the *Floating operations* group, we chose *fp_exe_x87*, that is, the number of x87 floating point operations executed. In fact, SSE operations counter is architecture-specific (SSE is a SIMD instruction set extension to the x86 architecture, exploited in Intel CPUs). The *Instructions* group does not contain counters that can be immediately chosen, so we will chose them after the correlation computation. *Retired instructions* would be surely preferable, being it architectural-independent, but we need to evaluate the difference with the other *Instructions* group counters in terms of correlation with power consumption. In fact, it is not easy to say a priori if the number of micro-operations is more or less expressive than the number of operations alone. Preferring instructions count statistics over micro-operations statistics could lead to possible losses in terms of accuracy. From the results of the computation we will also discover if the already selected counters are well fit to characterize our application's power consumption.

Performance counters are measured with the aid of Likwid (see Sec. A.3 on page 86) and perf [21], running all the ten PARSEC benchmark applications (see fig Fig. 3.3). This is important, because one can not afford to perform a study centred only on the benchmarks that will be tested during validation phase. During this study, we need to characterize the correlation between counters and power consumption for a wide range of applications, each radically different from the others. Here follow the list of the applications, along with a brief explanation of the type of operations they perform.

- ❏ Blackscholes - Option pricing with Black-Scholes Partial Differential Equation (PDE).
- ❏ Bodytrack - Body tracking of a person.
- ❏ Facesim - Simulates the motions of a human face.
- ❏ Ferret - Content similarity search server.
- ❏ Fluidanimate - Fluid dynamics for animation purposes with Smoothed.
- ❏ Freqmine - Frequent itemset mining.
- ❏ Raytrace - Real-time raytracing.
- ❏ Swaptions - Pricing of a portfolio of swaptions.
- ❏ Vips - Image processing.

- X264 - H.264 video encoding.

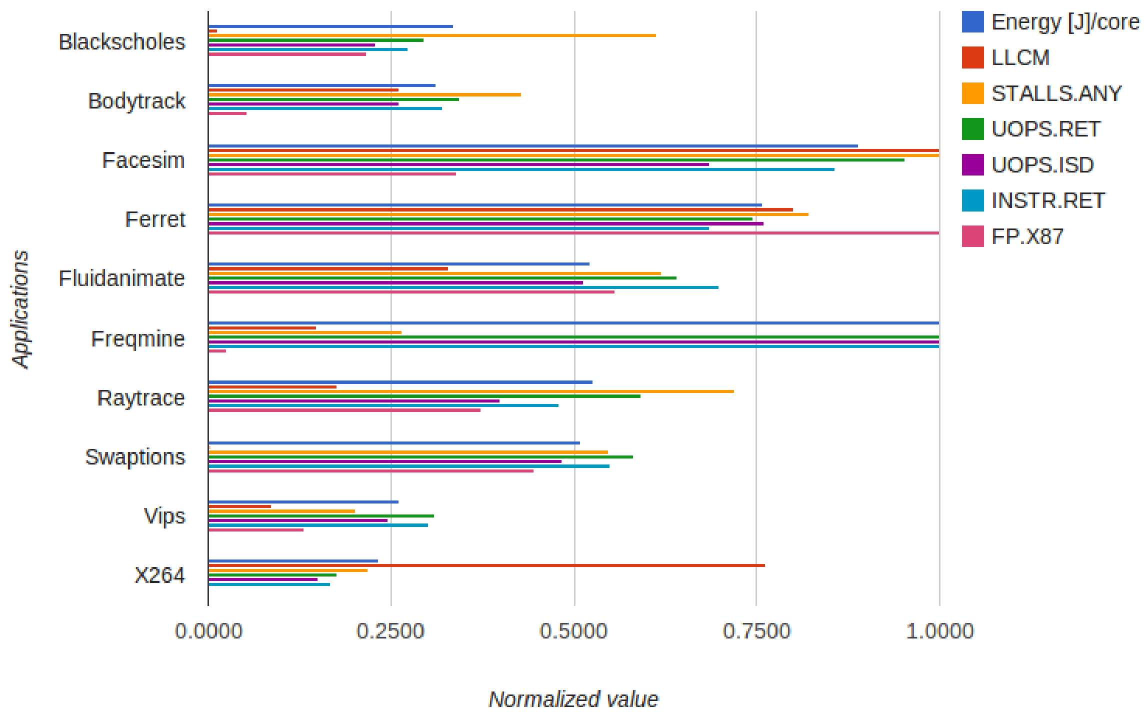


Figure 3.3: Sampled performance counters and power consumption for PARSEC benchmark applications. Values are normalized to 1.

Correlations, computed for each 10-sized vector using Spearman's rank correlation method, are shown in Tab. 3.6. Take in mind that highly correlated metrics are characterized by $\rho$ value tending to 1, while highly uncorrelated one are characterized by $\rho$ value tending to $-1$.

The results show that the selected counters are correlated with power consumption, and so are exploitable to achieve our goals. As expected, the best counters in terms of correlation are the ones from *Instructions* group. We decided to chose *retired instructions* counter for two reasons: first of all, the number of retired instructions can give us a hint on the real frequency at which an application is running, and various useful derived statistics for performance evaluation (e.g. CPI). Second, *retired instructions* - like *fp_exe_x87* - is not architecture dependant. Being it similar to the other instructions counters in terms of correlation, it has to be preferred over the others.

One could wonder why, among the counters we analysed, we did not include

| Counter | $\rho[-1;1]$ |
|---|---|
| llcm | $\sim 0.3$ |
| res_stalls | $\sim 0.6$ |
| instr_ret | $> 0.9$ |
| uops_issued | $> 0.9$ |
| uops_ret | $> 0.9$ |
| fp_exe_x87 | $\sim 0.4$ |

Table 3.6: Correlation between selected counters and power consumption

the ones directly related to communication congestion. The answer is simple: being this aspect widely covered by the counters of *Memory* and *Stalls* groups, we elected not to include it in our study.

Performing this study, we have pointed out how the total number of stalls, retired instructions, x87 floating point operations and last level cache misses are able to:

- ❏ Characterize application profile (i.e. CPU-bound or memory-bound).
- ❏ Cover a great part of performance/energy related resources, having us chosen coverage as a main metric to determine which counters to select.
- ❏ Characterize energy efficiency due to good degrees of correlation with energy consumption.

The selected performance counters will be exploited to represent the *fu_unbalance* of the system functional units and, in the case of last level cache misses, the *bound_mix* of a scheduling.

### 3.1.2 Application profile: CPU-bound versus memory-bound

To minimize performance losses induced by co-scheduling, we must choose the right applications to co-schedule on a BD. Thus, this particular goal is a BD-local one. Performance degradation caused by the execution of two application on the same BD can be formalized - in any case, from our purposes - as stated in equation Eq. (3.1) on page 35 where $a_1$ and $a_2$ are two applications, $max_{runtime}(a_1, a_2)$ is the execution time of the longest application.

$$Degr_{a_1 \wedge a_2} = runtime(a_1 \wedge a_2) - max_{runtime}(a_1, a_2) \tag{3.1}$$

Note that, obviously, if we eliminate the losses the runtime we expect is the runtime of the longest application. Being losses mainly caused by memory contention (and being such contention unavoidable) we need to try masking $a_1$ stalls with $a_2$ execution, and vice-versa. The problem is that we can not effectively know when, during the execution of a generic workload, an application will be characterized by high (or low) amounts of memory stalls; co-scheduling applications with the same stalls ratio wishing that they stall at different times is unthinkable. So, we decided to operate a one-way masking: let $a_1$ be an application characterized by high memory stalls average ratio (and so referred to as *memory-bound application*). To mask its memory stalls, what we want is to schedule an application $a_2$ that, conversely, is characterized by low memory stalls ratio. The application $a_2$ will fully benefit from this configuration, because it will exploit almost all the time while $a_1$ is stalled, being $a_2$ not to often stalled itself. One can not always provide two ready applications, one being heavily memory bound and the other being not memory bound, to be co-scheduled on a BD. However, what we can do is to chose the best couple $a_1, a_2$ so that the memory boundedness difference between them is the maximum available among the ready applications. In this way, BDs will contain, at the same time, the lowest performance-degradation-inducing application couple. Following the above mentioned line of thought and generalizing it with arbitrary amounts of applications, our proposal to evaluate the *bound_mix* - in terms of performance losses mitigation - of the co-scheduling of $n$ applications on the same binding domain is represented in Eq. (3.2) on page 36, where the constant $mem\_bound(a_i)$ represents the memory stalls ratio of application $i$, expressed as number of *R-LLC* misses per cycle.

$$bound\_mix(a_1 \wedge ... \wedge a_{n-1}) = var(mem\_bound(a_1), ..., mem\_bound(a_{n-1})) \tag{3.2}$$

High variances means high differences between the boundedness values of the applications co-scheduled on the same binding domain.

### 3.1.3 Functional units balance

Achieving energy efficiency is also a matter of resource balancing and bottlenecks avoidance. The practice formalized in Par. 3.1.2 on page 35 may certainly be enough to reduce energy consumption, but what about inducing a fair functional units utilization? This has a number of well-known effects; for example, spreading functional units utilization evenly through the entire system leads to energy consumption and wearing effects balancing among the cores, reduces temperature hotspots with positive effects to mean time to failure [18], and helps avoiding other minor causes of stalls further reducing - in our opinion - performance losses in the case of co-scheduled workloads. In this case, the goal evidently refers to a system-wide view of the allocated usage of the functional units; our aim should be to spread all utilization ratios evenly among the BDs. Thus, the ideal case would be the one where every BD of the *managed device* "receives" the same retired instructions, floating operations, stalls ratios. As above, this is generally not possible. However, we can try to schedule our applications so that the usage ratios allocated on each BD are reasonably even with the ones allocated on the other BDs. The *functional units unbalance* of a co-scheduling choice regarding a workload to be allocated on a device composed of $n$ BDs is definable as shown in Eq. (3.3) on page 37, where $s$ is the current scheduling choice, $fu$ is the functional unit under analysis, $avg(i, BD_j)$ is the average resource ratio for unit $i$ allocated on BD $j$.

$$fu\_unbalance(s, fu) = var(avg(fu, BD_1), ..., avg(fu, BD_{n-1})) \tag{3.3}$$

Here, conversely to the previous case, the goal is to achieve low variances. Utilization ratios have to be always as close to the global average value as possible.

## 3.2 Extending YaMS with CoWs support

Our proposal, explained in detail in the next section, is the partition of the policy in two parts: in the former, a computation to select the best AWMs evaluated in themselves is performed. In the latter, the best binding for every item of the ordered list is computed. In this manner, for example, if an AWM needs to be skipped because the related application has already been scheduled, no time is spent to evaluate its bindings. Moreover, from the first step we receive a list of the best performing AWMs. It is our duty, in the second phase, to chose how to co-schedule these applications, according to the chosen *binding*, in order to achieve minimal performance loss and energy consumption. Thus, the goodness of an AWM is no more related to his placement; an AWM, under this perspective, is to be preferred for its low execution time, its fair CPU demand and so on; where to place it, conversely, is only a matter of resource usage optimization. The importance of the AWM in itself is shown also in the scheduling application phase which will be performed not at the end of the list analysis, but whenever a single AWM is analysed: if an application cannot be scheduled on the best computed BD, maybe because that BD has not any more sufficient available resources to serve the AWM, the second best BD is selected and so on; the AMW had been evaluated to be better than the others, and so must and will be scheduled even if not in its ideal BD.

Please note that, to calculate *bound_mix* and *fu_unbalance* for a certain AWM on the possible BDs, system information regarding resource allocation means and variances both system-wide and per BD is needed. In fact, to compute the effect of the scheduling of an AWM on a binding domain, we must know which AWMs had already been chosen, bound and scheduled on the system. In this case, querying continuously system information from the resource manager is not a wise choice. A cleverer method is to keep a simple and lightweight internal system representation to keep track of the current co-scheduling statistical information.

The entire process, which is shown in Fig. 3.4, is is summarized as follows:

❏ For each priority level, a list of the pairs [Application, AWM] related to the currently ready applications is created. Here the unit under evaluation is couple [Application, AWM], not the *binding*. The choice to decouple AWM and BD evaluation comes from the need to move toward an approach that sees the spatial aspect of the co-scheduling problem as a co-scheduling optimization one. In other words, when analysing an application we need to select the best AWM in itself, postponing the problem of where to schedule the related application on to reduce performance degradations induced by the other running applications.

❏ AWM evaluation starts. Every pair [Application, AWM] is evaluated through a multi-objective computation taking into account:

  – AWM base value.

  – Reconfiguration overheads due to an AWM change.

  - Fairness index of the assignment of AWM requested resources taking into account current system load.

  Here, *congestion* and *migration* contributions from *YaMS* have not been taken into account. The reason is that *congestion* contribution will be represented during the BD evaluation phase by means of functional unit balance, while *migration* contribution, which is evidently *binding* dependant, will be exploited during BD evaluation.

- ❏ Beginning by the head of the ordered [Application, AWM] list, for each element which has not to be skipped, a list of ordered BDs - from the best to the worst candidate - is created. At this point, scheduling of the current application (configured as imposed by the current AWM) on the best BD is attempted. If failed, the next BD is selected and so on. BDs are ordered taking into account:

  - *bound_mix*, as defined above, with respect of the AWMs already scheduled on this BD.
  - Retired instructions, stalls and floating operation *fu_unbalance*, as defined above, with respect of the pairs [Application, AWM] already scheduled on the *managed device*.
  - Overhead in migrating the application among BDs.

Another interesting feature added by the *CoWs* extension is that the importance of the resource-aware goals is tunable at runtime. This allows more flexibility, more control given to the system user (that is, the ability to correct how the scheduler chooses the BDs during runtime, without modifying the scheduler code) and, above all, easy integration with design space exploration tools to chose the right weights to apply to these goals by means of automatic training on desired workloads.

These are the main features of our proposal. Now we will follow the design process step by step explaining our choices.

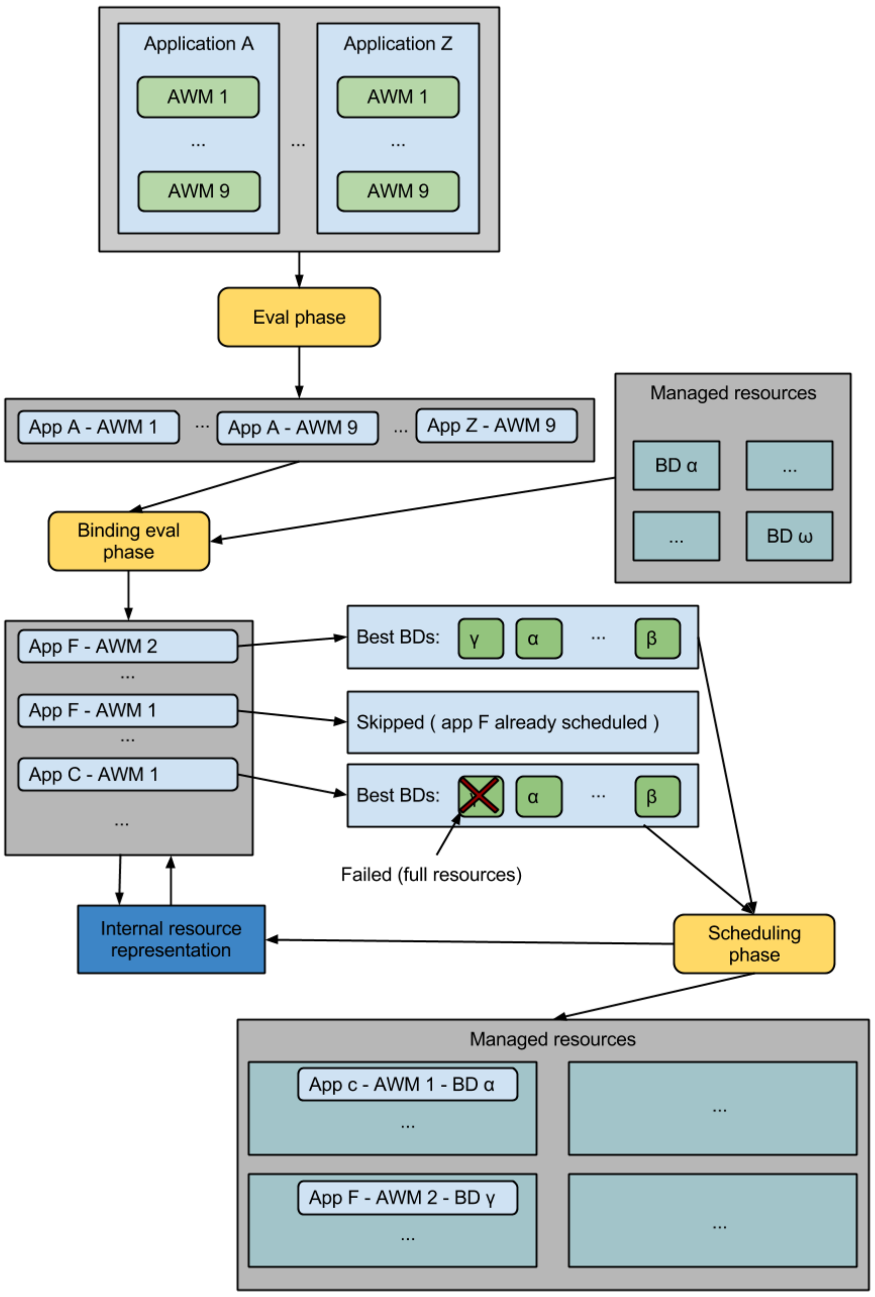


Figure 3.4: CoWs scheduler behaviour

### 3.2.1 Resource binding algorithm

The BarbequeRTRM framework and the concept of BDs, AWMs, *YaMS* multi-objective bindings evaluation may be a little confusing to the reader, being maybe the first time he faces this framework. In this section, a step by step explanation of *CoWs* centred on simpleness and clarity is provided. Some concepts will be reintroduced, to make sure the reader has understood them and can effectively get to the end of *CoWS* design description.

In BarbequeRTRM, scheduling is primarily triggered by events, like an application starting, ending or demanding more resources. The motive is obvious: a scheduler execution is needed only when the system changes, acquiring more or less free resources, or needing to adjust QoS for any application. When this trigger happens, there is a short time while all the applications prepare to be safely interrupted. As mentioned in Chap. 2 on page 13, every application comes with the *onRun* method, which has to contain the implementation organized, possibly, in light looped code. Usually, every time the application finishes this method, it is ready (if a trigger has happened) to be interrupted for the schedule phase. Now, as already mentioned, every application has a certain number of execution configurations. Every configuration is characterized by different theoretical performances, quality of service, required resources and so on. As already discussed, these are what we call AWMs (application working modes), and are collected in the *application recipe*, that is, an XML file describing the AWMs. Every application has its recipe, which also specifies the priority of the application.

How to decide which AWM is to be chosen from the recipe and what BD is to be chosen to bind the application with? *YaMS*, as explained in previous sections, selects all the ready applications at a given level of priority and creates a list of *bindings*. The evaluation of the *bindings* is a multi-objective one taking into account AWM general characteristics, current system status, possible effects of the bindings (time that would be spent to reconfigure and migrate is the binding is selected, and so on). After the evaluation phase, the list has already been reordered from the best *binding* to the worst. The list of AWMs is read from the start, and every *binding* is analysed; if the AWM refers to an already chosen application - that is, a binding referring to the same application has already been found in the list and had not been skipped - the couple is skipped. If the BD is full, the couple is also skipped. Else, scheduling happens: the application is configured as indicated in the AWM, and migrated on the BD.

In the case of *CoWs* support, *YaMS* behaves in a quite different way: no binding is required because, as already mentioned, from our point of view each AWM is to be evaluated in itself. The evaluation of the AWMs is a multi-objective one taking into account only AWM general characteristics and current system status. After the AWM evaluation, the BD evaluation is to be performed. The list, that during the previous phase has been reordered from the best AWM to the worst, is read from the start, and for each element its possible *bindings* are analysed; *bindings* analysis consists in a multi-objective evaluation performed to create a list of BDs ordered from the best to the worst, where the application could run. The metrics taken into account, this time, are:

❏ $R\_LLCM/cycle$ (needed for *bound_mix* computation)

❏ $RETIRED\_INSTR/cycle$ (needed for *fu_unbalance* computation)

❏ $STALLS\_ANY/cycle$ (needed for *fu_unbalance* computation)

❏ $FP\_OPERATIONS/cycle$ (needed for *fu_unbalance* computation)

❏ $MIGRATION\_PENALY$ (to this BD).

Application resource utilization statistics are now a collection of data that enrich the pair [Application, AWM]. In other words, in the XML application recipe every AWM description contains a section storing any number of additional resource utilization information. In our case, every AWM description contains last level cache misses, retired instructions, stalls, floating point operations per cycle of the AWM. The reader could wonder if is there a significant difference in terms of functional units utilization and execution profile (CPU or memory-bound) from an AWM to another one referring to the same application; the answer, indeed, is *definitely yes*. In Sec. 4.2.2 on page 62, these differences will be explained in detail, with real applications statistics.

How to evaluate a binding? The reader may have understood, from the earlier brief explanation, that our approach is in a cerFlowtain way very different from the one used in *YaMS*; here, to evaluate a binding, we need to exactly know what bindings had already been selected and scheduled. In fact, during a binding evaluation:

❏ How choosing this binding would affect the current average functional unit utilization is computed. This is needed to compute *fu_unbalance* indices of stalls, retired instructions and floating operations.

❏ An analysis of the BDs is performed, to calculate the current degree of *bound_mix* of the combination of applications scheduled on each BD. This information is exploited to understand if the binding under analysis would improve or worsen the *bound_mix* of the related BD.

So, while in *YaMS* scheduling requests are performed after the evaluation process, here we have a scheduling request after each AWM bindings ordering; once created the list of best BDs, the scheduling is performed using the first element of this list. If impossible, the second is picked and so on. After a successful schedule system information, contained in an internal structure initialized at the beginning of schedule phase, is updated. It will be exploited during the binding evaluation of the next AWM in list.

The metric exploited to order the BDs is introduced in Eq. (3.7) on page 43, where $AWM_i$ and $BD_j$ are the current binding to evaluate, $s$ is the system information, $F$ the functional unit related metrics (retired instructions, stalls, floating operations), $[\alpha, \beta, \gamma]$ the objectives values weights summing to 1, $[B, F, M]$ are the variations of *bound_mix*, *fu_unbalance* and *migration penalty* as defined in Eq. 3.4 to 3.6.

$$B = bound\_mix(AWM_i \wedge awms(BD_j, s)) - bound\_mix(awms(BD_j, s)) \tag{3.4}$$

$B$ is the difference between the *bound_mix* achieved by adding this AWM to the BD, and the *bound_mix* of the mix of applications currently scheduled in it.

$$F = \sum_{f \in F} fu\_unbalance(s, f) \tag{3.5}$$

$F$ is the total *fu_unbalance* for the processor resources representative metrics we chose to exploit during scheduling phase. How retired instruction, stalls, floating operation average system ratio would be affected by the scheduling of this binding? The higher is this value, the worse is the choice.

$$M = migration\_penalties(BD_j) \tag{3.6}$$

$M$ is the *migration penalty* related to previous system configuration. In other words, if before the scheduling phase an application was already running and it was on a certain BD, re-scheduling it on the same BD would avoid to spend time migrating the application.

$$cows\_metrics(AWM_i, BD_j, s) = \alpha * B - \beta * F - \gamma * M \tag{3.7}$$

The system information structure contains the following elements:

- average *bound_mix* and quadratic *bound_mix*, per BD.
- average *fu_unbalance* for the elements of $F$, per BD and system-wide.
- system load, that is, the number of applications scheduled on each BD.
- $[\alpha, \beta, \gamma]$ values. This element contains the weights, that can be accessed and modified at runtime from a fifo queue for design space exploration and testing purpose.

If the AWM refers to an already scheduled application, it is skipped even before evaluating its bindings. Else, the above process takes place and the best bindings list is created, then schedule is immediately attempted: if the schedule results possible, the application is configured as indicated in the AWM, and migrated on the BD. Otherwise the next BDs will be tried following the order imposed by the list and, upon a successful schedule, system information will be updated. Only then the next pair [Application, AWM] will be analysed.

Note that all possible bindings can be rejected; if the AWM has heavy resource requirements and the system is already quite loaded, a schedule could be not possible on any BD. That is not a problem, since further in the list there will be other AWMs related to the same application, probably requesting fewer resources (in fact, AWMs further in the list are the less performing ones).

## 3.3 Methodology

In this section, we introduce a standard and semi-automatic flow to create applications integrated with BarbequeRTRM which can exploit *CoWs* support. First of all, training phase methodology is presented. The BarbequeRTRM integration has been briefly explained in Sec. 2.2 on page 15. Once having at disposal a few integrated applications, a *training phase* is needed. Statistic collection takes place with the aid of MOST tool (see Sec. A.2 on page 83), which gives us a mean to perform automatic tests on our applications, identifying the best configurations of inputs that minimize an arbitrary set of objectives. Energy consumption information, conversely, is extracted exploiting monitoring tools. In the case of Intel processor family, we made use of *Likwid* tool (see Sec. A.3 on page 86), while in the case of the

AMD NUMA platform *IPMItool*, an utility for managing and configuring devices that permits system information monitoring, is exploited (see [20]).

These tools, along with the RTLib provided by the BarbequeRTRM, form a powerful infrastructure to sample and collect resource usage, performance and energy profile of any application, following the flow shown in Fig. 3.5.

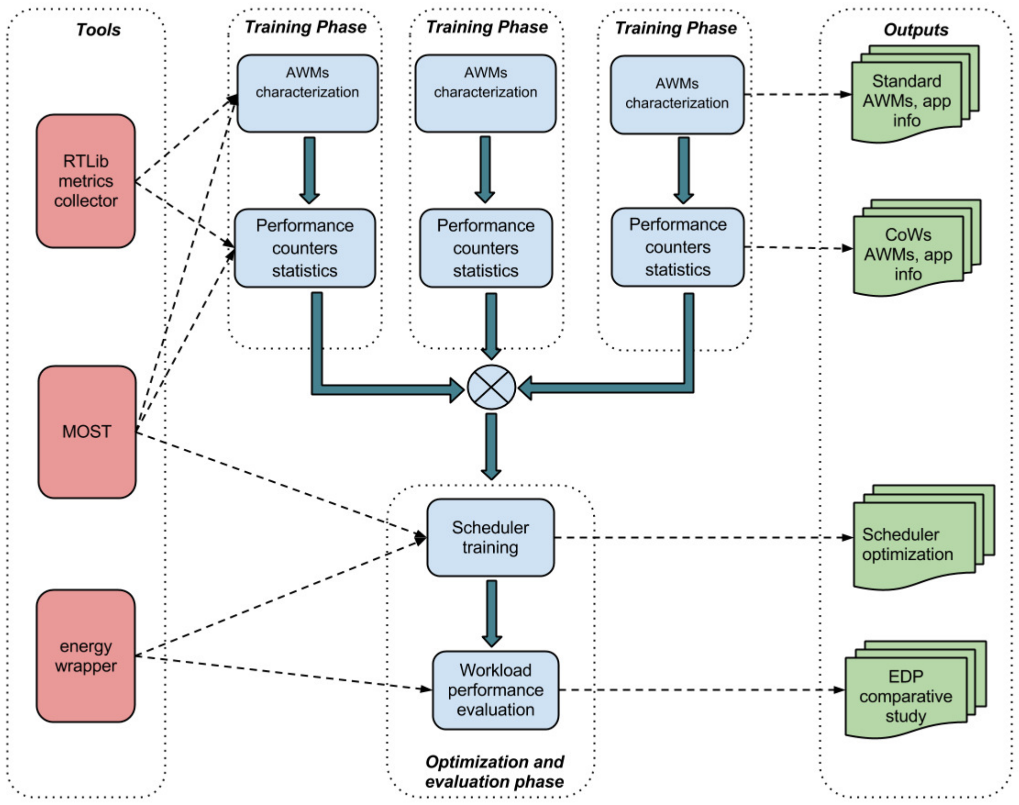


Figure 3.5: Application development, integration and evaluation flow.

The RTLib metrics collector is exploited to collect information about the applications during MOST exploration. This information will aid the applications developer to better understand his application characteristics, and so to optimize the code and choose the best suited AWMs to insert in the recipe. The role of MOST, who is needed for almost all the phases of the flow, is to provide automatic exploration of the possible configurations, and the creation of databases containing the collected information. Exploiting MOST optimization algorithms, the developer have no longer the need to manually analyse the whole space of configurations. Clustering and filtering reduce the possible points so that only the best configurations remain in the database. Once the training phase is performed for the desired applications, the developer is able to identify a number of possible - or, in any case, interesting - workloads, composed of the applications. These workloads will serve to train the scheduler and tune it with a combination of weights such that the average EDP among the selected workloads is minimized. To compute the EDP metric, a script that collects energy consumption information during the execution of the workload is needed. Hence the creation of the energy wrapper described in Par. 3.3.3 on page 49.

### 3.3.1 Application working modes (AWMs) characterization

In the first phase we use, as inputs, a range of possible CPU and memory quotas along with any desired user-defined inputs of the application. Fig. 3.6 shows, from an abstract point of view, the interaction between the tools; MOST, which needs to receive user-created files containing the description of the application (inputs and outputs and their domain) and of the tests to be performed (design of experiments, goals, optimizations) runs on the *unmanaged device*, while the application runs on the *managed device*, launched and monitored by the MOST wrapper. The results are stored, at the end of the execution and after a series of optimization steps defined by the developer, in a database (or, if desired, in a report complete with graphs and other useful statistics) with the statistical information about all the tested configurations.

The developer, analysing the results, is able to understand several important characteristics of his application. Most of all, he will be able to tell how the inputs influence the performance of the application according to the given configuration explored. For example, the results could show that performance does not benefit from configurations requesting more a certain number of threads. In other cases, one would even discover that an application performance does not scale properly by increasing the amount of assigned resources, so creating AWMs with resource request greater than that would be useless.

The reader must take in mind that this first test has not the aim to find out what configurations are the most performing in general. It serves to understand, at each level of desired performance and QoS, which are the locally more performing ones. So, wanting to create an AWM which requests a minimal CPU quota and very low memory - a typical configuration to be selected when the system is too heavy loaded - we can find out which value of the other inputs is the best to im-

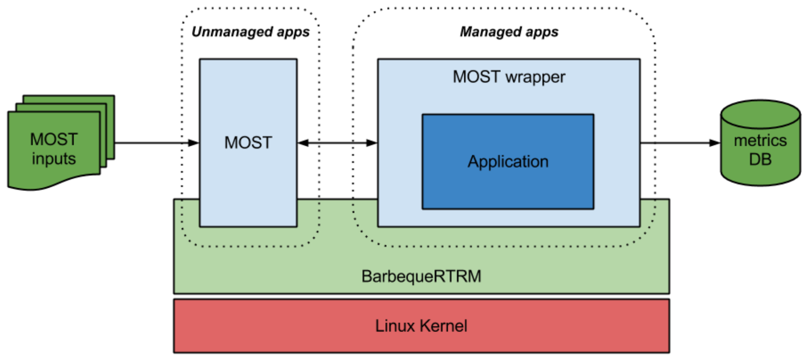


Figure 3.6: MOST - BarbequeRTRM integration schema.

pose to this configuration to maximize performance, energy efficiency or whatever the developer needs to maximize. These tests also aid the developer to discover the upper bound of the resources to allocate to an application at runtime or of the ideal parallelism level to exploit during the execution. Once mastered the usage of MOST, with a little inventiveness almost any type of test can be done; this is one of the characteristics that make of MOST a very powerful tool.

At the end of these tests, what goes in the recipes is a small set of AWMs representing resource requests and qualities of service as much varying as possible, so that the resource manager can find the most suitable AWM for each possible load of the *managed device*.

For each test, we collect information regarding a single AWM. So, to test an application which we plan to provide with $n$ AWMs, we have to run the test $n$ times. In the case of huge sets of inputs, a lot of tests would be performed; an application with $n$ AWMs, number of threads in $[1; m]$ and, for example, another custom input in $[1; p]$ would need $n$ tests, each test running the application $m * p$ times. An huge number of results is not a problem during database evaluation, because MOST gives the possibility to operate clustering on the points and to remove dominated points with the aid of Pareto curves. In fact, the output consist usually of few points, leading to fast and easy analysis by the developer, who makes the final choice over the configurations to insert in the recipe. Regarding test duration, conversely, MOST give us the possibility to indicate which type of exploration we want to exploit during the tests, in order to reduce significantly the exploration time. For more information, please refer to Sec. A.3 on page 86.

### 3.3.2 Performance counters statistics

The goal of these tests is to collect information that will later be exploited by *CoWs*. The general schema of the test is the same of the previous phase, shown in Fig. 3.6. For each AWM we want to insert into the recipe, we run a MOST test to collect resource usage information, with the aid of the performance counters selected in Sec. 3.1.1 on page 29. The inputs will be the desired inputs for the application, for example the quality of a video encoding, or parallelism level. The outputs, as briefly mentioned above, are:

- ❏ last level cache misses.
- ❏ resource stalls.
- ❏ retired instructions.
- ❏ x87 floating point operations.
- ❏ number of cycles, needed to normalize the above mentioned statistics.
- ❏ variance of the above mentioned statistics, to aid the developer during comparison phase.
- ❏ execution time

In this case, few minimization objectives are mandatory; we focus on finding out what configurations are the fastest, but trying also to reduce any type of stall, to achieve energy efficiency improvements. Here are the goals we selected during our own tests:

- ❏ $stalls/cycle$
- ❏ $R - LLCM/cycle$
- ❏ $CPI$
- ❏ $executiontime$

Conversely to the previous phase, here we do not need any optimization algorithm to be exploited by MOST. This is a test to be run one time per AWM, simply to collect the desired information. Done this, the obtained values are to be inserted into the application recipe, as will be shown in detail in the next chapter.

Note that, as we did for our own tests, to speed up this phase of the flow the developer can merge this test into the one performed in the previous phase; in fact, if there are no other inputs to evaluate other than the ones studied in the previous phase - that is, if we already decided how the application will be executed and our only goal is to collect its resource utilization information - we can add the metrics to collect to the ones to be collected in the first phase. Once chosen an AWM to insert in the recipe, all related information is already available in the MOST database.

### 3.3.3 Scheduler training

*CoWs* comes with predefined standard values for the weights used to differentiate *bound_mix*, *fu_unbalance*, and *migration* penalties importance during the multi-objective optimization performed to evaluate bindings. Optionally, to further improve performance, another type of test can be performed: the scheduler training test. This test has the aim, once the developer has individuated a certain number of representative or critical workloads, to find out which combination of the weights exploited during the BD multi-objective computation performed by *CoWs* guarantees the best results in terms of *energy-delay product* during the execution of the selected workloads. This test is very simple even if, from a theoretical point of view, it is considerably more complicated than the previous ones. Its schema is shown in Fig. 3.7; MOST runs, as always, on the *unmanaged device* while the energy consumption sampling script - which will be referred to as energy wrapper - is executed on the *managed device* to be able to see the correct partition. This time, the application - which, in reality, is a script capable to invoke whole workloads comprehending the applications which had been analysed in the previous phases - is started by the energy wrapper, which is invoked by MOST wrapper. Doing this, the following flow of data is generated:

- ❏ MOST communicates the current design point to his wrapper.
- ❏ MOST wrapper:
  - Communicates with the RTLib to set the weights as commanded by the design point.
  - Communicates with energy wrapper to execute the application.
- ❏ The energy wrapper executes the application, so that:

- RTLib samples and computes any desired useful statistics, yielding them as an output for MOST wrapper.
- The energy wrapper samples and computes energy statistics, yielding them as an output for MOST wrapper.

❏ MOST wrapper saves all the received results in a database point, as done during previous tests, for later optimization.

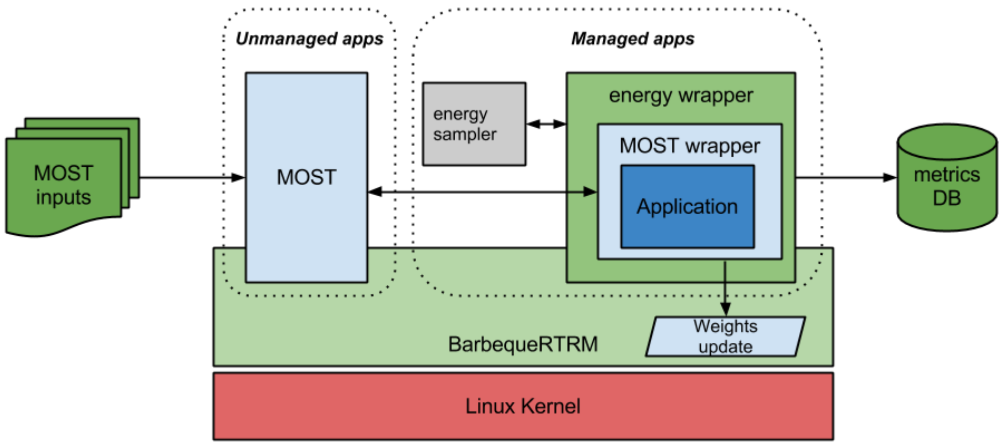


Figure 3.7: Interaction between the tools during weights training.

The energy wrapper is a simple script that starts the execution of the workload, sampling in the meanwhile energy consumption and execution time. When the workload finishes to execute, it yields such information to the MOST wrapper. The algorithm can be viewed, in a simplified version, as follows:

```
begin;
   start_time = start_timer();
   launch_workload_script();
   while(workload_active) sampling.sample_energy();
   end_time = stop_timer();

   exec_time = end_time - start_time;
   energy_consumption = sampling.average_energy();

   return [exec_time, energy_consumption];
end;
```

The output of the scheduler workload training will be a database containing the vectors $[\alpha, \beta, \gamma]$ inducing best performances and energy efficiency.

### 3.3.4 Workload results evaluation

This test is the simplest among the ones described in this chapter; the developer needs only to exploit the energy wrapper to run:

- a basic version of the application or workload.
- the BarbequeRTRM-integrated version of the application or workload.
- the BarbequeRTRM-integrated version of the application or workload, exploiting *CoWs* support.

Analysing the outputs, the developer will be able to compare execution time, energy consumed per core, and EDP of the three different types of execution.

This flow has been followed using two applications from PARSEC benchmark suite - *bodytrack* and *ferret* - on an Intel quad-core processor and on a NUMA AMD processor. In the next chapter, after a complete description of the experimental setup, results and outputs will be shown for every distinct phase regarding the tests on the Intel processor, thus showing a complete use case scenario. Then, the performance results showing EDP comparisons between the three cases (BarbequeRTRM with and without *CoWs* support, and the no-BarbequeRTRM case) will be presented and commented for both the platforms.

Chapter 4

# Experimental Results

In this chapter the experimental setup will be described, focusing on the two platforms exploited to validate our thesis. The results will then be introduced and commented.

## 4.1 Analysed multi-core based systems

In Chap. 1 on page 3 we talked about two different types of devices: multi-core and many-core processors. The BarbequeRTRM resource manager addresses both these types of devices; in fact, it keeps a part of the resources to execute itself along with the *unmanaged applications* and exploits the remaining resources to execute *managed applications*, and this can be done in both multi and many-core devices. The best scenario, in any case, would be the one where a host device manage an external accelerator, such as a generic GPU. For the validation we chose two very different devices. The first, a quad-core Intel processor, has not a huge amount of resources to exploit. Here a resource utilization aware approach becomes interesting to understand how a workload can be managed in a ″constrained″ device, where the applications have an high probability to be subject to performance degradations due to resource contention. The second device under testing is a NUMA platform, which consists of four nodes each exploiting a quad-core AMD processor. This device is interesting to analyse because:

❏ There are lots of resources to manage, with respect of the Intel case. This causes the managing of resources to become a very important aspect during the execution of our workloads, and should show the real BarbequeRTRM contribution in terms of performances and energy efficiency with respect to the Linux case. *CoWs* results will be compared to the BarbequeRTRM results in an environment that greatly benefits from BarbequeRTRM support.

❏ Keeping each node as a BD, we can manage to exploit three 4-PEs BDs, with potential higher speed-ups with respect of the Intel case, where we had three single core (double-threaded) BDs.

❏ The nodes share only the main memory: every node has its independent cache hierarchy, and this should permit more contention avoidance than in the Intel case.

Before describing the two systems we exploited during the validation, another clarification is needed. As mentioned in Sec. 1.2.2 on page 6, in [14] frequency scaling was exploited to reduce contention when suitable combination of applications were not found. In our case, we let Linux manage the frequencies. We selected the *on demand* kernel governor, that scales the CPU frequencies according to the CPU usage. In this way, we aim to achieve good performances, having in the meanwhile the assurance that energy consumption will benefit from frequency scaling when possible. Here follows a detailed description of the two devices, along with the needed information about operative system and resource partitioning.

### 4.1.1 Single multi-core processor

The multi-core case exploits a system featuring an Intel Core i7-2670QM processor running up to 2.20GHz (see Tab. 4.1). Each core has independent L1 and L2 caches, while the L3 is shared between all the cores. Hyper-threading permits each core to run two threads. So, from our point of view, the set of resources consists in 8 PEs. This device does not offer many resource partitioning choices; in fact, as mentioned in Par. 3.1 on page 27, we need BDs that share the smallest possible part of the cache hierarchy. Being the L3 a resource shared among all the cores, we already know that our BDs can not be totally independent. We also know that the two PEs featured by a core have to belong to the same BD; in fact, those PEs share L1 and L2 caches. Being the single core - due to the problem just introduced - the minimal possible BD, we chose to partition the processor in three BDs, so that the device is seen as a single-BD *unmanaged device* and a three-BDs *managed device* (see Fig. 4.1). The system under analysis features 8GB of RAM and exploits Linux kernel 3.5. The memory has been fairly divided among the BDs, so each BD can exploit a 2GB memory partition.

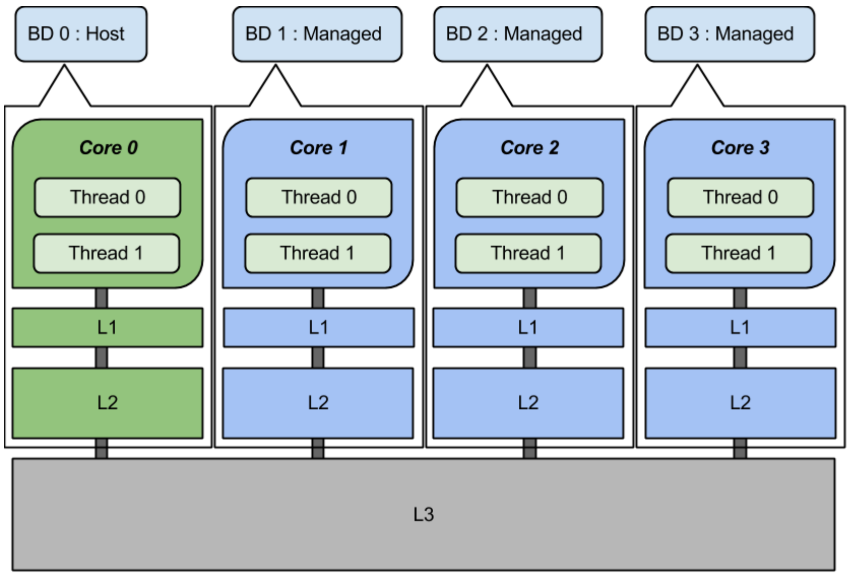


Figure 4.1: Intel i7 resource partition

### 4.1.2 Multiple multi-core processors

In this case, we exploited a NUMA device consisting in four quad-core nodes. each node features a AMD Opteron 8378 Quad-core processor running up to 2.80GHz (see Tab. 4.2). In the single node, each core has independent L1 and L2 caches, while the L3 is shared between all the cores. The set of resources, having as mentioned above four nodes at disposal, consists in 16 PEs. This device offers many resource partitioning choices; however, as mentioned in Par. 3.1 on page 27, we

| | |
|---|---|
| Microarchitecture | Sandy Bridge |
| Processor Core | Sandy Bridge |
| Data width | 64 bit |
| Number of cores | 4 |
| Number of threads | 8 |
| Floating Point Unit | Integrated |
| Level 1 cache size | 4 x 32 KB instruction caches<br>4 x 32 KB data caches |
| Level 2 cache size | 4 x 256 KB |
| Level 3 cache size | 6 MB |

Table 4.1: Intel Core i7-2670QM CPU information

need BDs that share the least possible part of the cache hierarchy. Being the L3 a resource shared among all the cores of a node, the choice had been driven toward the partitioning of the system into four BDs, each containing a node. The single node doesn't share any part of the cache hierarchy with the other nodes, and this means that choosing the single node as a BD, as shown in Fig. 4.2, we have at disposal 4-PEs BDs (granting good performances) with the least possible memory contention issues between each other (granting less contention-related performance losses). The system under analysis exploits Linux kernel 3.9. The memory has been fairly divided among the BDs, so each BD can exploit a 6GB memory partition.

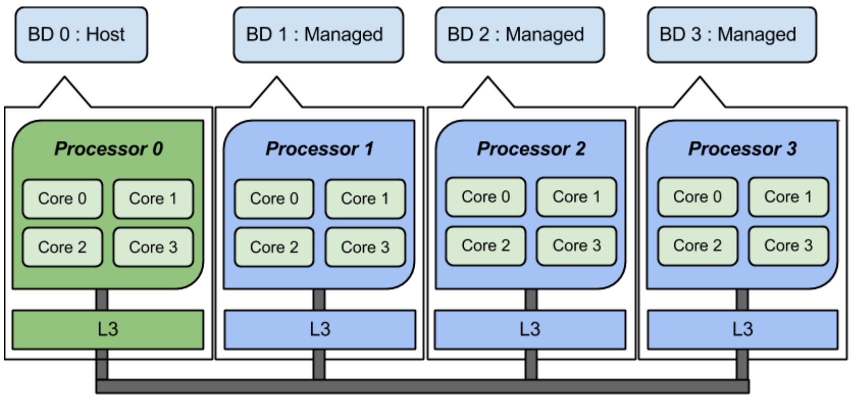


Figure 4.2: AMD NUMA resource partition

## 4.2 Methodology exploitation

In this section, the entire methodology flow described in Chap. 4 on page 53 is followed. The applications under analysis are, as already mentioned, *bodytrack* and *ferret* from PARSEC benchmark suite. *Bodytrack* performs the body track of a person. This computer vision application is an Intel RMS workload which tracks a human body with multiple cameras through an image sequence. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces. *Ferret* implements a context similarity search server. This application is based on the Ferret toolkit which is used for content-based similarity search. It was developed by Princeton University. The reason for the inclusion in the benchmark suite is that it represents emerging next-generation search engines for non-text document data

| | |
|---|---|
| Microarchitecture | K10 |
| Processor Core | Shanghai |
| Data width | 64 bit |
| Number of cores | 4 |
| Number of threads | 4 |
| Floating Point Unit | Integrated |
| Level 1 cache size | 4 x 64 KB 2-way associative instruction caches<br>4 x 64 KB 2-way associative data caches |
| Level 2 cache size | 4 x 512 KB 16-way associative caches |
| Level 3 cache size | 6 MB 48-way associative shared cache |

Table 4.2: AMD Opteron 8378 CPU information

types. In the benchmark, they have configured the Ferret toolkit for image similarity search. *Ferret* parallelism is performed exploiting the pipeline model, conversely to the *bodytrack* case.

In the next section, the final results will be shown for both the processors we elected to use for the evaluation. The use case presented here, conversely, refers only to the Intel processor. This processor is an Intel Core i7-2670QM exploiting 4 CPUs at 2.20GHz (maximum), hyper threading, 6144 KB Intel smart cache. The OS is Ubuntu Linux 12.04 ”Precise Pangolin”, kernel version 3.5.

### 4.2.1 Application working modes characterization

In the preliminary application study, the goal is to understand which AWMs to insert in the recipes, and what is the best value of any other execution parameter to maximize performance. For the purpose of this study, the recurrent parameter of all tests is the parallelism level exploited by the application. The need to smartly evaluate the number of threads is obvious: trading more memory and communication/synchronization overhead with increased parallelism is a common choice, but finding out the correct balance is not always easy. The second parameter is the amount of resources BarbequeRTRM will assign to the application. This is not an implicit parameter; in fact, we do not give it as an input to MOST. It is an input, specific for every set of tests, which has to be given to the BarbequeRTRM framework itself. This means that, as already mentioned, for every AWM of a design space exploration will be run, each design point characterized by a different number of threads.

Tests had been performed using MOST. Being the design space (the number of possible input combinations) quite small, we elected to perform a full search design of experiments, with the aid of Pareto curves optimization but without clustering the results. In Fig. 4.3 to Fig. 4.3 the results for *bodytrack* application are shown, for an exploration with number of threads ranging from 1 to 5. As can be immediately seen, there is not so much difference between the results. Even if MOST has removed some points during optimization phase, the user can immediately notice that, probably, the removed points suffered due to little fluctuations caused by the limited accuracy of the sampling. The reason of these uniform results, as anticipated in the previous chapter, is the fact that in this case the BDs feature only a two-threaded core, making this processor not well suited for massive parallelism.

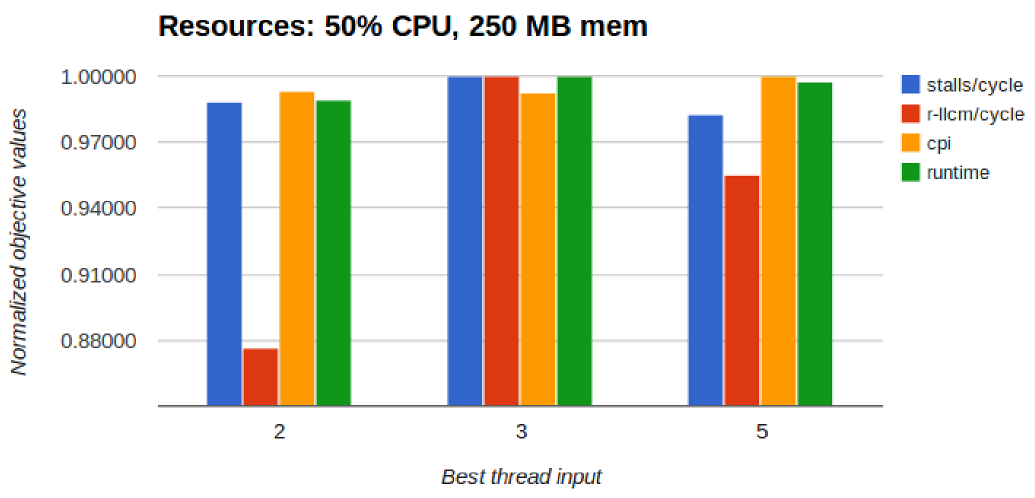


Figure 4.3: *Bodytrack*: DSE for 50% CPU usage case

Results from *ferret* (see Fig. 4.7 to Fig. 4.10) are analogous to the ones just mentioned. Take in mind that the parallelism exploited in *ferret* is obtained due to *pipeline parallelism*. So, each loop iteration is split into stages and threads operate on different stages from different iterations concurrently. It is understandable that, in cores exploiting only two threads, this technique can not reach its full potential. Results show that the best number of threads to assign to each stage is, unsurprisingly, 1.

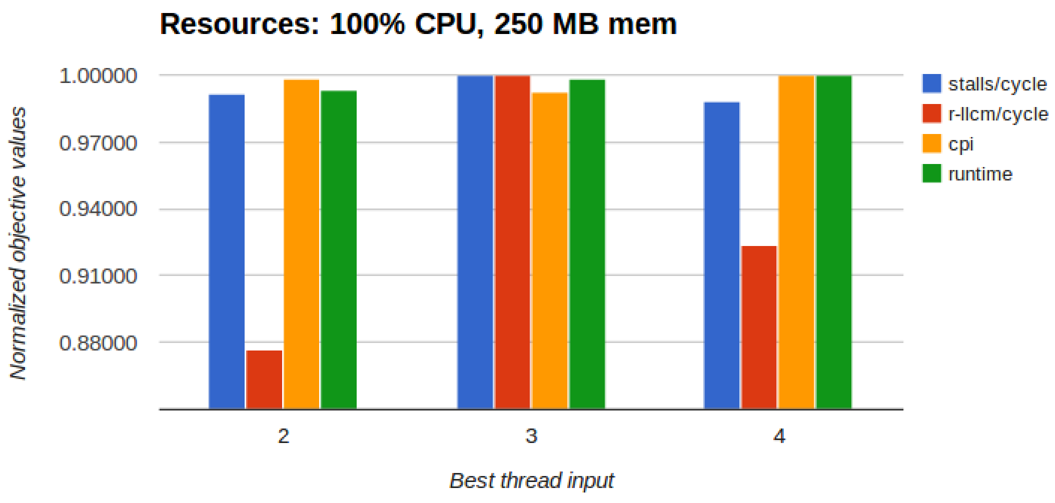


Figure 4.4: *Bodytrack*: DSE for 100% CPU usage case

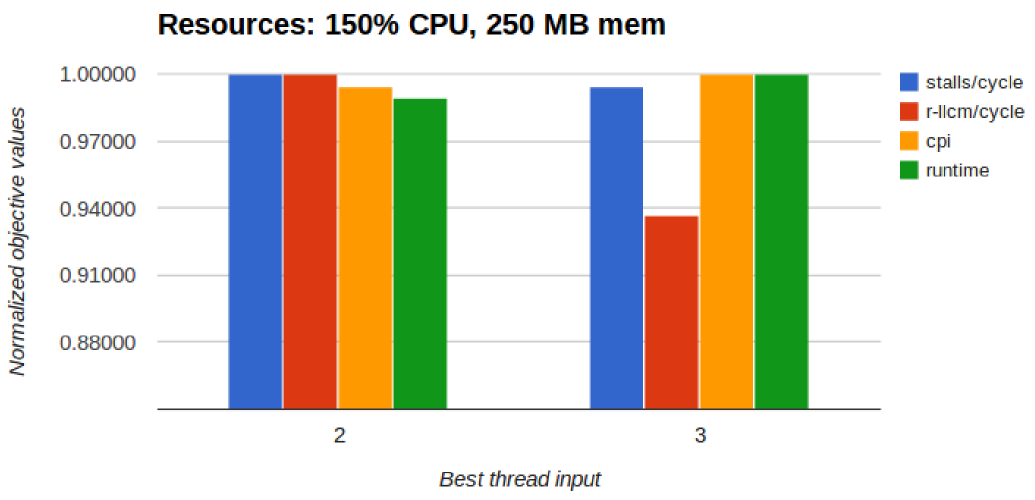


Figure 4.5: *Bodytrack*: DSE for 150% CPU usage case

Chosen the number of threads, we analysed the impacts of CPU quota on performance. This could be done also for memory utilization, but in this case we already know that the applications do not suffer from memory limitations; in fact, in this configuration up to 2000 MB of memory can be reserved to each BD, so memory is evidently not a problem. CPU quota, conversely, is a good parameter to analyse. In fact, what we want to is to discover how, increasing the CPU quota granted to the execution of the application, the performance scales. This information permits us

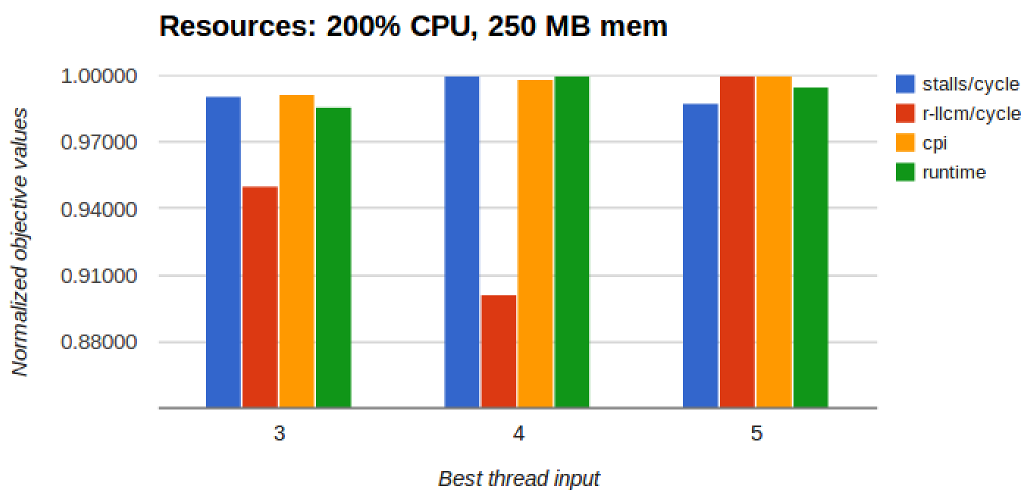


Figure 4.6: *Bodytrack*: DSE for 200% CPU usage case

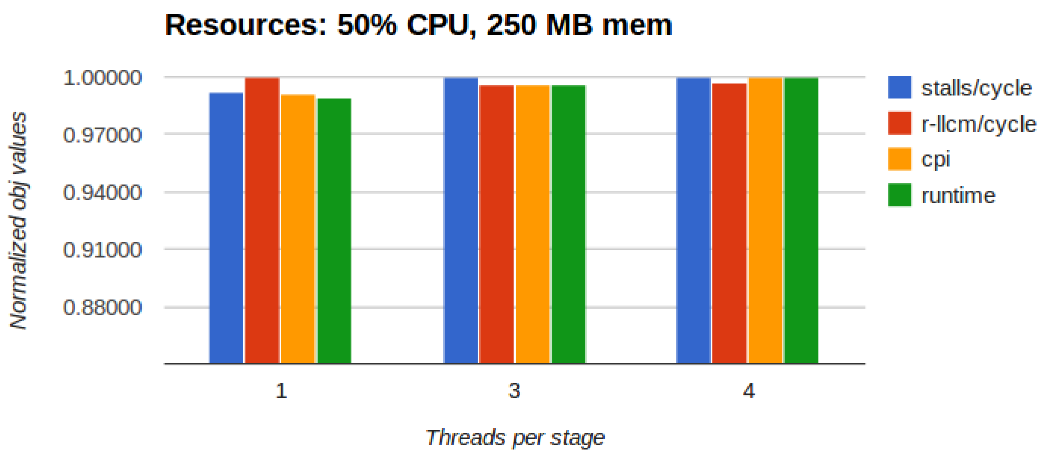


Figure 4.7: *Ferret*: DSE for 50% CPU usage case

to set an upper bound to the maximum CPU quota available to the application, and to calculate the *value* metric, the one exploited during AWM evaluation to represent the importance of each AWM among the others of the same recipe.

The results, that can be seen in Fig. 4.11 are very interesting and show that:

❑ Both the applications are not so memory-bound that they cannot effectively exploit a full BD; the full BD AWM, meaning 200% CPU quota in this case, is always the most performing AWM.

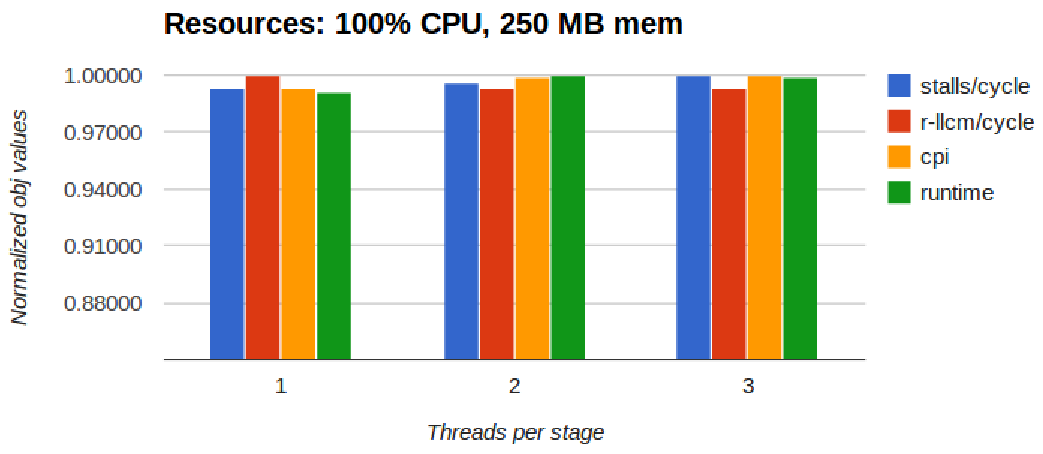


Figure 4.8: *Ferret*: DSE for 100% CPU usage case

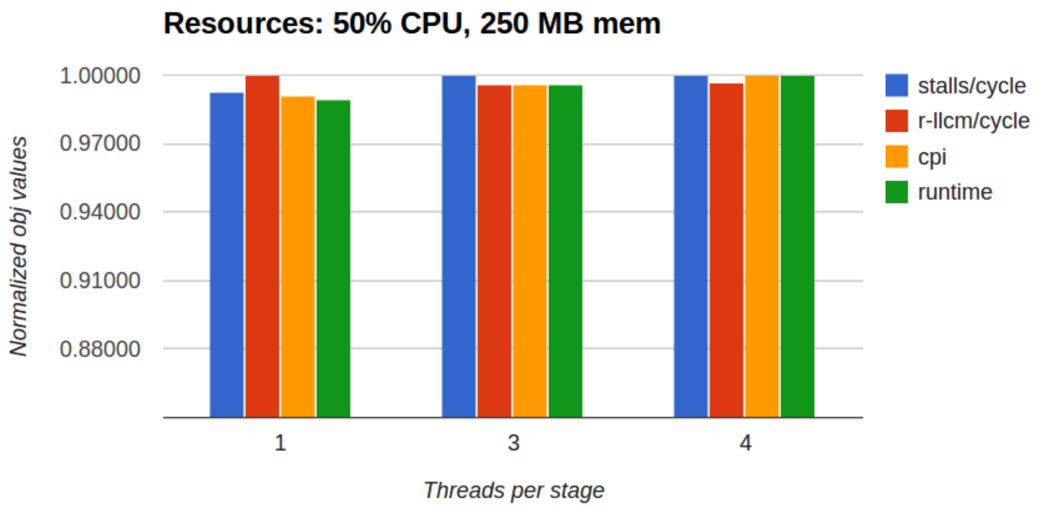


Figure 4.9: *Ferret*: DSE for 150% CPU usage case

- In both the applications, AWM with 150% CPU quota are not better than the ones requesting 100% CPU quota. *Bodytrack* application even executes faster if requiring 100% CPU quota instead of 150%.
- The two applications are characterized by a different degree of *boundedness*. In fact, *ferret* results to be more affected by CPU quota increments, with a speed-up of $\sim$ 3.45 between 50% and 200% CPU quota case, versus the $\sim$ 2.63 speed-up value achieved by *bodytrack*.

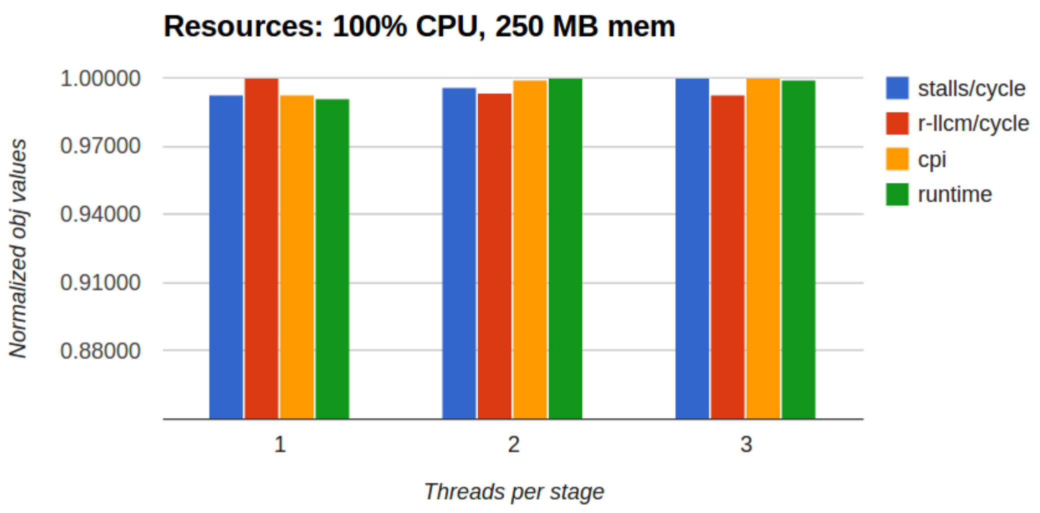


Figure 4.10: Ferret: DSE for 200% CPU usage case

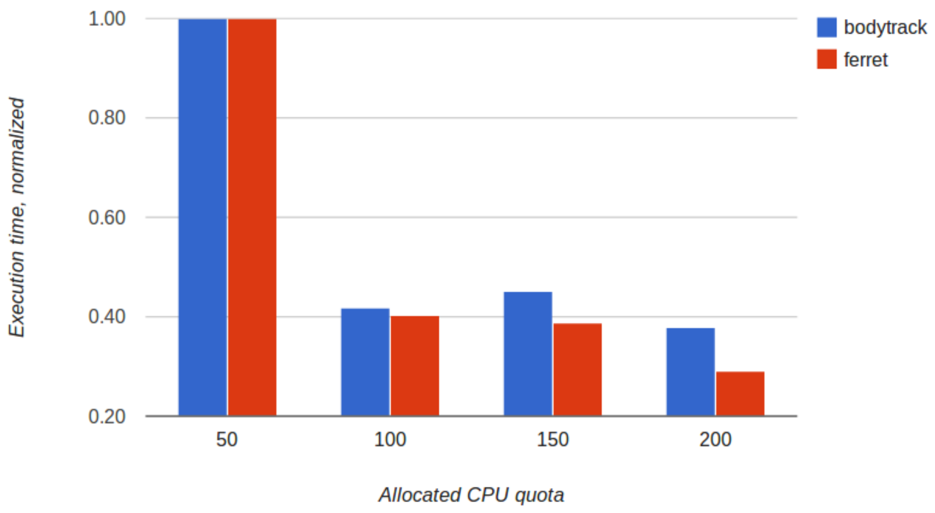


Figure 4.11: Influence of CPU quotas on *bodytrack* and *ferret* execution time

### 4.2.2 Performance counters statistics

Once created the recipes, resource utilization information has to be collected as defined in Par. 3.3.2 on page 48. The information collection can be performed with the sole aid of RTLib, or exploiting MOST to further study the application and make the final choice regarding the AWM to chose or else - as we did - exploiting the information already collected during the previous phase. In fact, during the preventive study we extracted all the needed information from this test. So, once chosen a running configuration and a set of inputs, we already had all the needed information stored in the database. This could happen since our input consisted only in the number of threads, an input whose value had already been decided studying the outputs of the first phase.

Now the recipes are filled and ready to be exploited by the resource manager. Before describing the compiled recipes, two interesting aspects of this study have to be shown; first of all, we chose to insert 150%-CPU-utilization AWMs even if we demonstrated, in the first phase, that they are not performing with respect of less resource-requesting AWMs. As described in previous chapters, the scheduler - having the recipes at its disposal and knowing that these AWMs are not much more performing than the ones characterized by less resource utilization - will not ever chose them, confirming its multi-objective nature that make it able to evaluate both performances and quota fairness of the AWMs. Gaining the slightest of speed-ups or even no speed-ups at all at the cost of 50% more CPU usage is evidently not a wise choice, and choosing a 150% CPU quota AWM leaves the BD with only 50% of free resources for other applications. As shown during the AWM study, AWM exploiting this minimal quota should be chosen only if really necessary, being notably less performing than the others. The second interesting aspect is that, as mentioned in the previous chapter, having at disposal two full recipes with 4 AMWs each we can evaluate how the CPU quota utilization of an application can influence its resource utilization statistics. Is it really useful to characterize each AWM with resource utilization statistics, or is it purely application dependant information? As shown in Fig. 4.12, resource utilizations statistics can grow even fourfold with increasing CPU quota utilization. This is not a strange fact, because increasing CPU quota means that more resources are exploited. Exploiting more resources and executing an application in less time than the normal, the density of stalls, cache misses and the other statistics increases dramatically. This confirms the need of collecting statistics about not the single application, but all the possible running configurations of an application.

Let us give a closer look to what a recipe look like; Fig. 4.13 shows *ferret* recipe, focusing on a single AWM. The xml file features a container named *awms* containing the description of our AMWs. Each AWM description contains two sections: the former is the required resources descriptor, the latter is the resource utilization statistics container. For more eye clarity, the statistics had been multiplied by 1000.

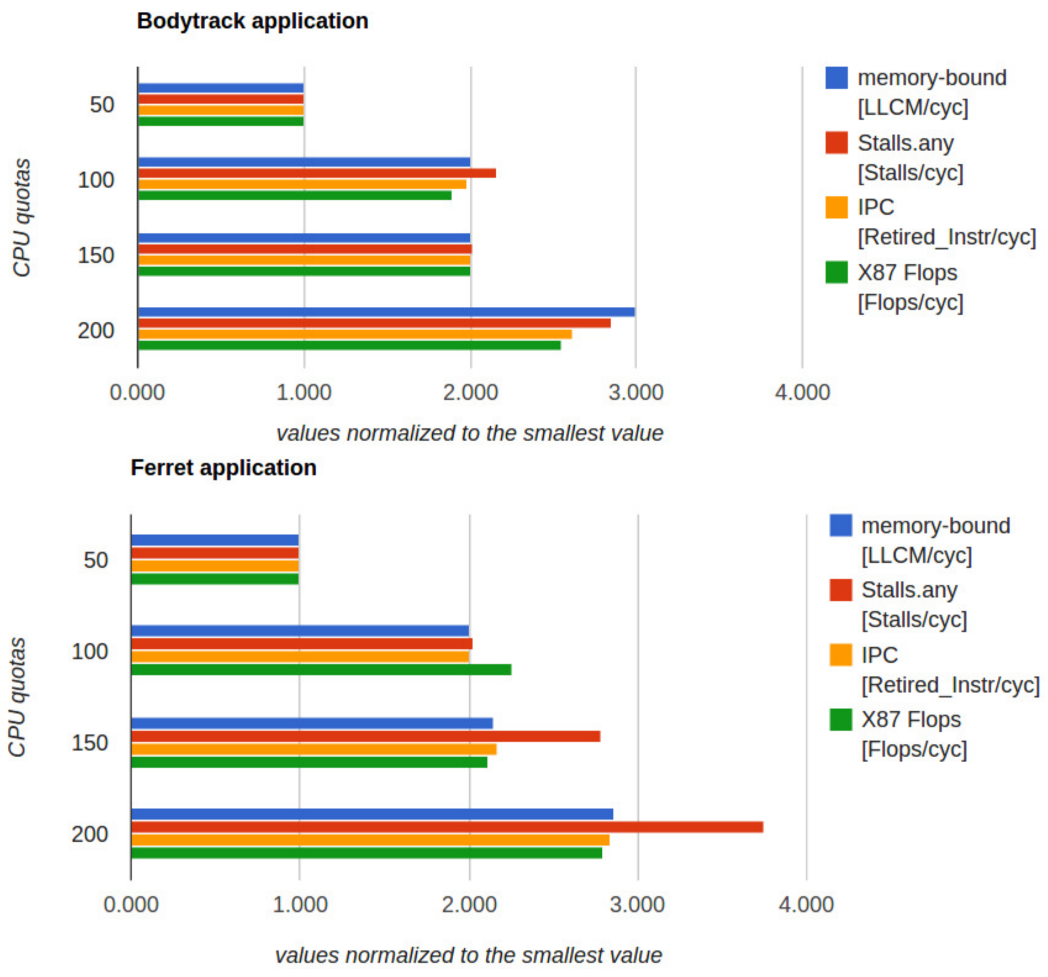
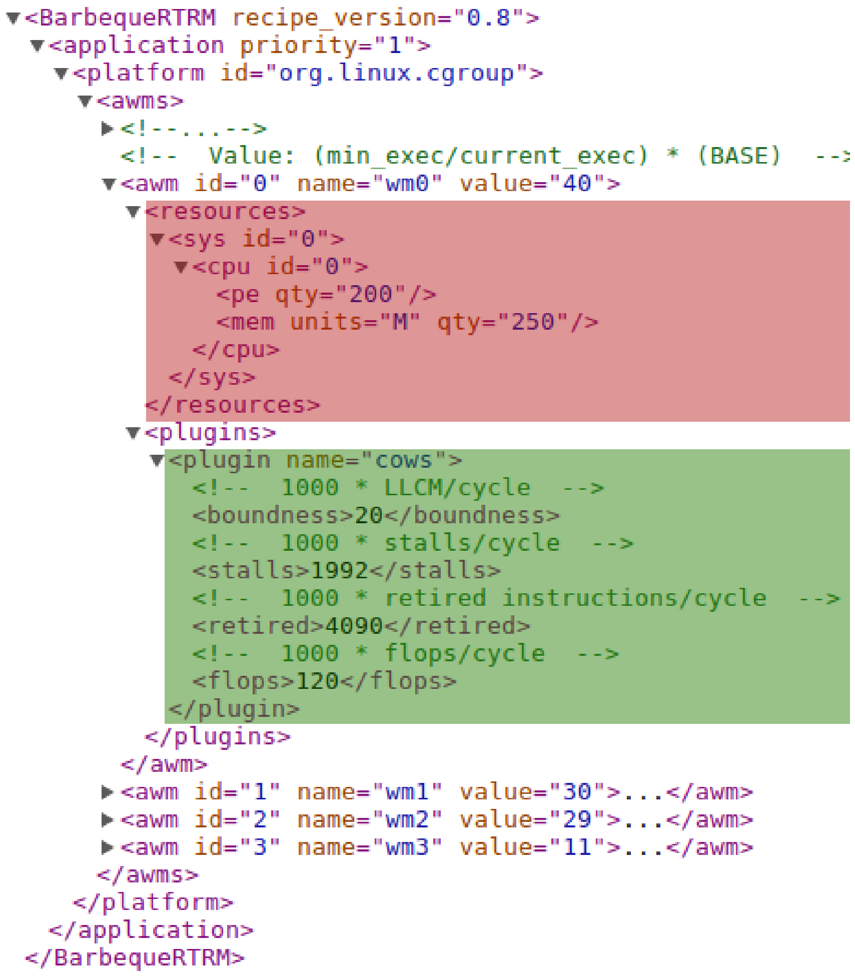


Figure 4.12: Normalized resource utilization statistics versus CPU quota

### 4.2.3 Scheduler training

Once ready to start executing our applications, a final process of optimization can be performed: the scheduler training. Aim of this phase is to chose the best weights to be used to characterize the importance of *bound_mix*, *fu_unbalance* and *migration penalty* during the multi-objective evaluation of the bindings. Running several workloads exploiting BarbequeRTRM and its RTLib to manage the resources and schedule the applications, MOST to automatically run the tests and compute the desired objectives, and Likwid to calculate energy consumption and runtime - which will be exploited to calculate the EDP of the executions - we are able to evaluate the different weights combinations. The interaction between the tools is the one already presented during the design of the tests in Fig. 3.7. The weights represent



```
▼<BarbequeRTRM recipe_version="0.8">
  ▼<application priority="1">
    ▼<platform id="org.linux.cgroup">
      ▼<awms>
        ▶<!--...-->
         <!--  Value: (min_exec/current_exec) * (BASE)  --
        ▼<awm id="0" name="wm0" value="40">
          ▼<resources>
            ▼<sys id="0">
              ▼<cpu id="0">
                 <pe qty="200"/>
                 <mem units="M" qty="250"/>
               </cpu>
             </sys>
           </resources>
          ▼<plugins>
            ▼<plugin name="cows">
               <!--  1000 * LLCM/cycle  -->
               <boundness>20</boundness>
               <!--  1000 * stalls/cycle  -->
               <stalls>1992</stalls>
               <!--  1000 * retired instructions/cycle  -->
               <retired>4090</retired>
               <!--  1000 * flops/cycle  -->
               <flops>120</flops>
             </plugin>
           </plugins>
         </awm>
        ▶<awm id="1" name="wm1" value="30">...</awm>
        ▶<awm id="2" name="wm2" value="29">...</awm>
        ▶<awm id="3" name="wm3" value="11">...</awm>
       </awms>
     </platform>
   </application>
 </BarbequeRTRM>
```

Figure 4.13: Ferret recipe, showing the 200% CPU quota AWM

the degree of importance of a term with respect of the others so, as already said, the three weights should sum up to 1. So, we evaluated the different combinations of weights, with each of them being an integer ranging from 1 to 8, setting constraints in MOST design space to impose that the sum of the three weights would amount to 10. This is another interesting feature of MOST tool: setting constraints helps the tool to reduce considerably the number of design points.

This test, in the same manner of the final evaluation tests, uses EDP as a metric to represent both energy efficiency and runtime improvements of the configurations. We configured a set of three scenarios to evaluate the weights:

- LIGHT LOAD: two instances of *ferret*, two instances of *bodytrack*.
- MEDIUM LOAD: three instances of *ferret*, three instances of *bodytrack*.

- HEAVY LOAD: four instances of *ferret*, six instances of *bodytrack*.

The test has been performed in two steps; the first one was performed with MOST applying clustering and Pareto optimization to remove dominated points. The results shown that the best combinations, in all the cases, were characterized by the template $[bound\_mix, fu\_unbalance, migration] = [X, 1, Y]$, so demonstrating that our considerations about memory stalls avoiding was a good one, and that *bound_mix* and *migration penalties* are to be preferred over *fu_unbalance*. The second step, having understood that, has been to perform a test in which *fu_unbalance* weight was set to 1, and only the other weights changed. Having reduced a lot the design space (having lost an input and being compliant to the constraints, eight combinations of weights were possible instead of thirty-six), a concrete and manual analysis of the results was possible. The results, shown in Fig. 4.14 and Fig. 4.15, are very interesting.

Configurations where *bound_mix* received the maximum possible weight had the best EDP rating in all the tested workloads. So, we chose the combination $[B, F, M] = [8, 1, 1]$ to calibrate the multi-objective evaluation. Note that this is a strong indication that our considerations on cache hierarchy sharing were valid: from the results we can evince that *bound_mix*, that is, the degree of latency masking between running application that share a part of the cache hierarchy, is really the most important metric to consider during a co-scheduling choice. If an AWM is best suited - in terms of *bound_mix* - to run on a certain BD, the best choice is to migrate the application to that BD without worrying about the relative migration penalties.

## 4.3 Run-time scenarios

The final test is needed to evaluate the scheduler. We can run our workload scenarios, where the allocations of resources from the *managed device* partition is controlled by: in three different environments:

- The Linux scheduler and memory management system.
- The BarbequeRTRM exploiting *YaMS* scheduler.
- The BarbequeRTRM exploiting *YaMS* scheduler with *CoWs* support.

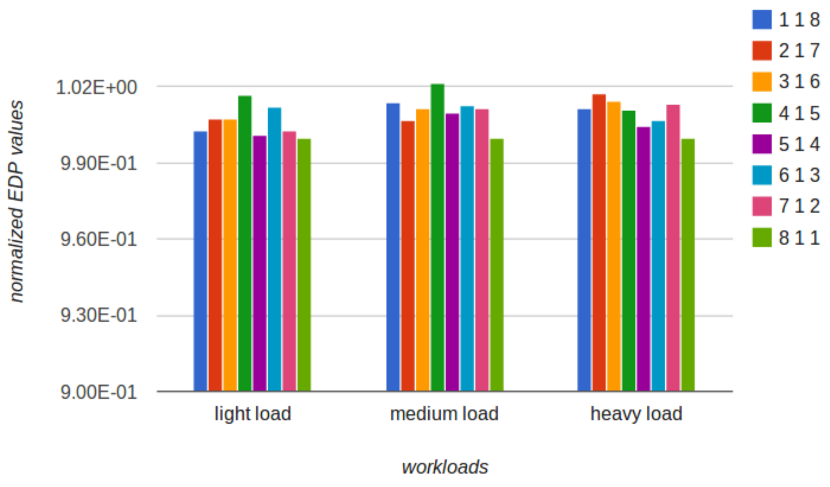


Figure 4.14: EDP results for the three workloads

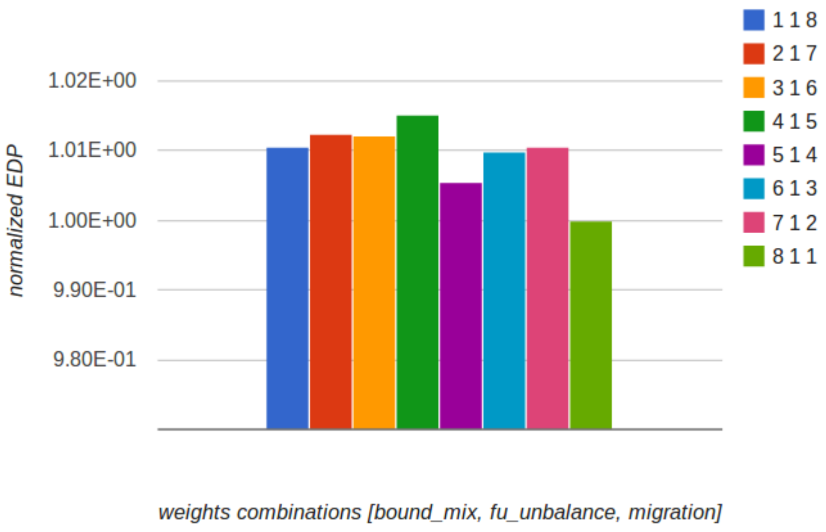


Figure 4.15: Average EDP results for the weights test

For these tests, we added a fourth workload scenario composed of six *ferret* and four *bodytrack* applications, meant to cause a congestion on the system so that, in all the three cases, the scheduling had to work under stress and with a very low degree of freedom.

### 4.3.1 Single multi-core processor

The results demonstrate that a managed device managed by BarbequeRTRM with *CoWs* support achieves significant improvements not only in EDP, but also in both execution time and energy consumption in *all* the tests. As shown in Fig. 4.16, when treating with light loads *CoWs* leads to EDP improvements with respect to *YaMS* alone greater than 30%, speeding up execution where even BarbequeRTRM alone can not rival with the Linux manager. Tests running medium, heavy and congestion workloads shown that both BarbequeRTRM cases improve significantly execution time and energy efficiency with regard to the Linux case, with *CoWs* support always leading to improvements with respect with YaMS scheduler alone. EDP, in fact, is improved compared to *YaMS* case by 27% with the medium load, 9% and 3% with heavy and congestion loads.

The congestion scenario is probably the most significant for the purpose of our study. Note that running the congestion load, a level of congestion is reached such that the system is completely full. In other words, using or not using *CoWs* is irrelevant during much of the execution, because a situation is soon reached when only 6 instances of *ferret* will be executing (average execution time of *ferret* is five time longer the execution time of *bodytrack*). In this case, there is only one optimal way to schedule the applications, with or without *CoWs* support: two instances of *ferret* on each binding domain. The fact that *CoWs* succeeds in achieving best performances leading to the above mentioned 3% improvement is a very important indicator of the validity of our thesis: with or without *CoWs* support, we have that in the first part of the workload execution all the applications are executing, while in the second part only *ferret* instances are executing, two instances per BD. This means that in the first part of the workload execution, the only one where co-scheduling choices of the two versions of *YaMs* differ, *CoWs* co-schedule the applications so that performance degradation due to contention is minimized.

Now, one could wonder what are the effects on system temperature. BarbequeRTRM allow us, as proved, to run applications notably reducing execution time. Do these speed-ups have negative effects on temperature? Or conversely, do achieved energy efficiency improvement and resource utilization balancing succeed in avoiding hotspots and/or reducing the average device temperature?

Fig. 4.17 show how BarbequeRTRM succeeds, even with the congestion load, to decrease the maximum and average temperature of the managed device with respect to the Linux case. In three out of four tests, *YaMS* case is slightly better

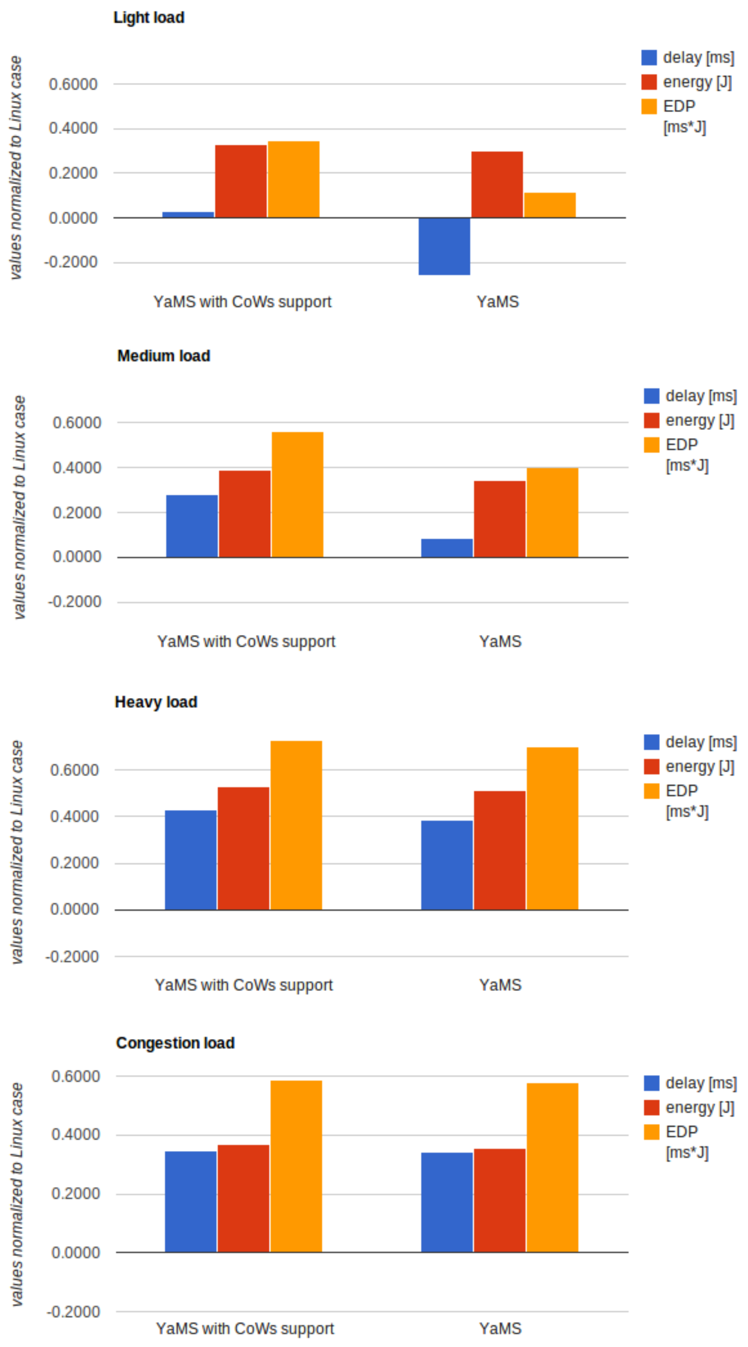
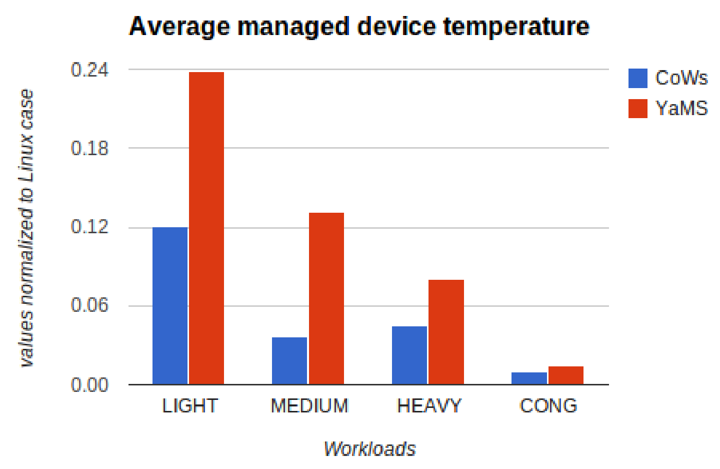


Figure 4.16: Intel case: Gains with respect to Linux case, with and without *CoWs* support

than *CoWs* case; however both of them are better than the Linux case, and *CoWs* improvements in both execution time and energy consumption is enough to nevertheless guarantee the goodness of this resource-aware policy.



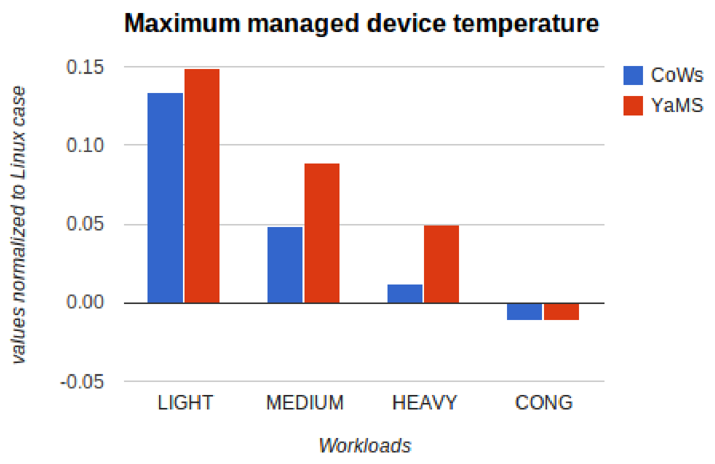


Figure 4.17: Intel case: Managed device temperature with and without *CoWs* support, with respect to Linux case

### 4.3.2 Multiple multi-core processors

The flow was followed also for the AMD NUMA device. Again, we found out that *bound_mix* metric is essential during co-scheduling choice to minimize contention-related performance degradation. Another interesting discovery has been the fact that, in this case, all the best weight configurations featured a very low migration value. As shown in Fig. 4.18, this time a configuration with *fu_unbalance* value greater than 1 was present among the best configurations. However, it lead to the worst EDP result among the best three configurations in all the workloads. Even this time, the configuration $[B, F, M] = [8, 1, 1]$ was chosen as the best possible combination. Having noted that *bound_mix* metric tends to be, as in the Intel case, a dominant metric, we decided to grant to all the metrics the possibility to assume a null value. The fact that combinations with *fu_unbalance* or *migration* amounting to zero are not present among the best ones is a strong indicator that all these metrics, even if exploited using low weights, are nevertheless useful to choose the best co-scheduling.

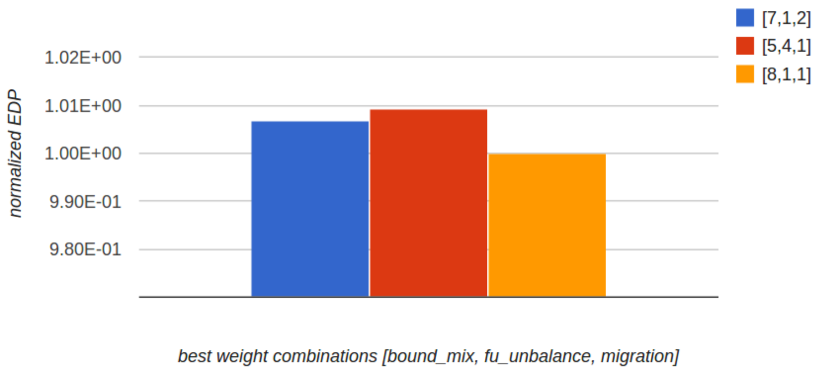


Figure 4.18: AMD NUMA: average EDP results for the weights test

The results demonstrate, as in the Intel case, that a managed device managed by BarbequeRTRM with *CoWs* support achieves significant improvements not only in EDP, but also in both execution time and energy consumption in almost all the scenarios. As shown in Fig. 4.19, having more resources at this disposal and so operating in a system where resource managing is essential, the BarbequeRTRM succeeds in achieving improvements in both performance and energy efficiency with respect to the Linux case in *all* the scenarios while, as shown in Sec. 4.3.1 on page 68, in the Intel case BarbequeRTRM without *CoWs* support was not very effective in the case of light loads. *CoWs*, with the exception of the congestion case where EDP is the same of *YaMS* alone, succeeds in achieving further improvements: when treating with light loads *CoWs* leads to EDP improvements with respect to *YaMS* alone amounting to 30%. Tests running heavy and congestion workloads shown that both BarbequeRTRM cases improve significantly execution time and energy efficiency with regard to the Linux case, with *CoWs* support always leading to further improvements with respect with YaMS scheduler alone. EDP, in fact, is improved compared to *YaMS* case by 30% and 3% with medium and high loads. This time, conversely to the Intel case, we can see that *CoWs* support leads to great improvements only with light and medium loads while in the case of heavy loads, where in any case the behaviour of the two schedulers tents to be the same, little improvements with respect to *YaMS* alone are achieved. In the case of congestion load, the EDP is the same: in fact, BarbequeRTRM with *CoWs* support achieves a slight improvement in performances (2% circa), but has slightly higher values regarding consumed energy (again, 2% circa).

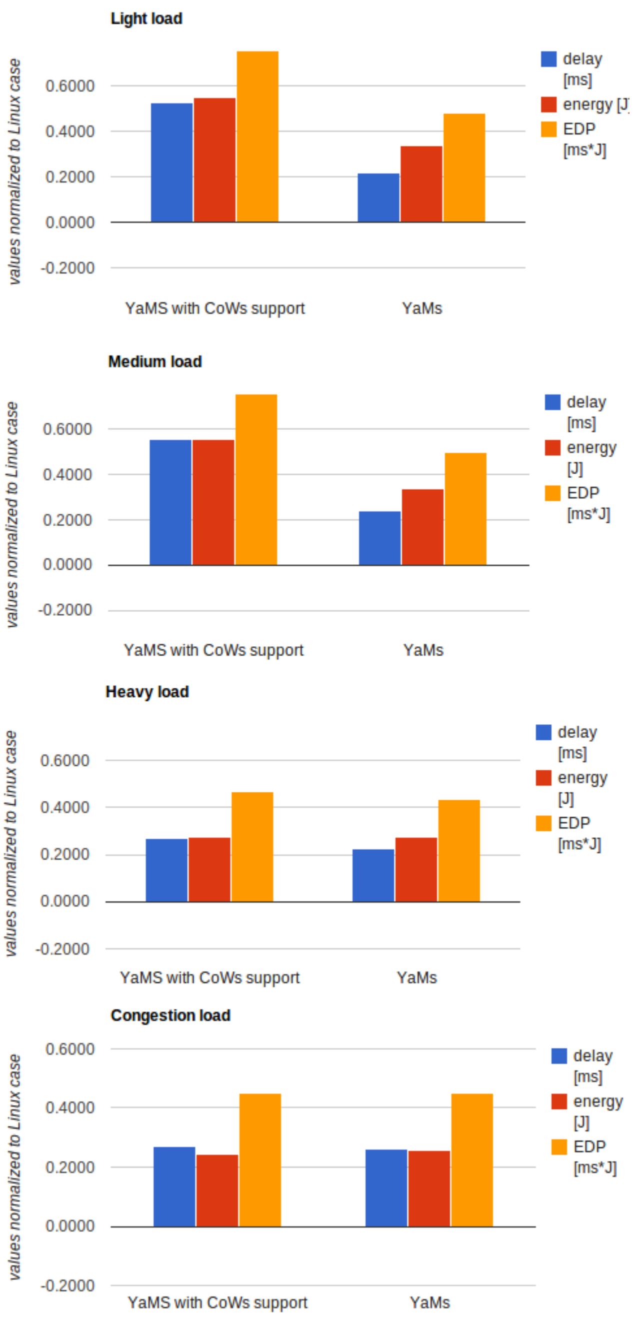


Figure 4.19: AMD NUMA case: Gains with respect to Linux case, with and without *CoWs* support

Chapter 5

# Conclusions

In this chapter the conclusions are drawn, pointing out what we did and how it was done. Every phase of the work will be outlined and described, and possible future developments are proposed.

## 5.1 Achieved results

The main contribution of this work is to prove that a smart resource-aware co-scheduling policy can lead to a wide variety of benefits, such as performance speed-ups, energy efficiency improvements, reduction of thermal hotspots. To do this, a policy had been designed and implemented as an extension of the multi-objective modular scheduler *YaMS*, from the BarbequeRTRM resource manager. This was done because the BarbequeRTRM resource manager is portable, features a ”for free” profiling support for the applications and is characterized by energy efficiency and performance improvements with respect to the standard Linux manager. We proved that, focusing not only on which are the best tasks to schedule but also on where to schedule them, further improvements in performance and energy efficiency can be achieved with respect to the current results. We validated the policy, codename *CoWs* (CO-scheduling Workloads), on two different platforms: the first one featuring a 2nd Generation Intel Core i7 Quad-core Processor (eight cores with hyper-treading), the second a NUMA device featuring four AMD 10h Family Opteron 8378 Quad-core processors with distinct cache hierarchies. The validation had been based on the comparison of energy-delay product of different workload scenarios in Linux, the resources being managed by Linux standard manager, BarbequeRTRM framework, BarbequeRTRM framework with *CoWS* support.

| Contribution | BarbequeRTRM (CoWs support) | BarbequeRTRM (YaMS support) | State of art |
|---|---|---|---|
| Scheduler portability | yes | yes | no |
| Support to any type of multi-core platform | yes | yes | no |
| Explicit performance counters analysis | yes | no | no |
| Multiple resources taken into account | yes | no | yes |
| Tunable weight of each metric by means of scheduler training | yes | no | no |
| Energy efficiency emphasis | yes | yes | yes |
| Integrated statistics collection support | yes | yes | no |
| Design, integration and validation flow | yes | no | no |
| Design space exploration support | yes | yes | no |
| QoS monitoring support | yes | yes | no |

Table 5.1: Our contributions

In addition to the above mentioned contribution, a flow spanning from application analysis and integration with *CoWs* to EDP evaluation of the workload has been proposed, to guide the users through development and scheduling of his applications. Our contribution is summarized in Tab. 5.1.

### 5.1.1 Co-scheduling workloads in a resource-aware perspective

Our approach had been based on the idea that characterizing resource utilization of applications and scheduling them in ways that minimize memory contention and spread other resource utilization ratio over the system could lead to notable improvements in performance and energy efficiency. This procedure was greatly aided by the idea of partitioning the managed resources in resource groups that:

- Comprehend at least one core, that is, can execute applications.

- ❏ Have distinct cache hierarchies. Cache levels shared between multiple groups are not to be considered as resources, but as a part of the infrastructure linking the resources.
- ❏ Do not share any other resource, thus resulting virtually isolated from the other groups.

This kind of partitioning had been greatly aided by BarbequeRTRM resource manager, which provides easy ways to divide the resources in arbitrary ways, just writing the desired resource partitioning in a configuration file.

We introduced the concept of *bound_mix* of a co-scheduling choice, that is, the degree of efficiency of the applications comprehended in the chosen resource-group local co-scheduling to mask each other their memory misses latencies. Along with this concept, we introduced the one of *unbalance*, that is, the degree of unfairness regarding system-wide resource utilization allocation of a co-scheduling choice. Resource utilization statistics had been computed by means of performance counters sampling. We supported the counters choice with a rigorous study centred on the maximization of system resource coverage and energy consumption correlation.

We proved that, co-scheduling applications on the resource groups trying to maximize *bound_mix* and minimize *fu_unbalance* and *migration penalties* to different groups, a scheduling choice can be computed such that execution time and energy consumption of either light, medium, and heavy load workloads are considerably reduced. To aid the user in the case of generically known workloads, which is quite frequent in the case of sets of managed resources exploited to accelerate the execution of critical applications, we provide means to train the scheduler so that the average EDP improvement is maximized among the most critical workloads, or in any case a number of expected workloads. This training phase has the goal to tune *bound_mix*, *fu_unbalance* and *migration* importance during the multi-objective co-scheduling computation.

We selected *energy delay product* (EDP) as the main evaluation metric of BarbequeRTRM applications performance, instead of execution time alone. This is very important because, given the upcoming importance of energy consumption in current devices, a metric representing both performance and energy efficiency is necessary.

### 5.1.2 *CoWs* integration application flow

We provided a flow which, exploiting the BarbequeRTRM resource manager whose scheduler had been extended with *CoWs* support and several tools created to aid the user in design space exploration and performance analysis, partners the user trough application development, resource usage statistics sampling, scheduler training based on expected workloads, workloads evaluation.

Developing an application exploiting *CoWs* support is not different from developing a BarbequeRTRM integrated application. In fact, all the applications already modified to exploit BarbequeRTRM can benefit from *CoWs* extension. This is very important because, provided that resource-utilization information collection is performed, an application integrated with BarbequeRTRM gains resource aware scheduling support *for free*. Moreover, before collecting resource usage information the user can test his applications exploiting MOST-BarbequeRTRM integration to perform optimizations, chose the best inputs and configurations to execute the application and gain a great aid - currently not provided by BarbequeRTRM - to create application recipes, understanding which working modes are unavailing and which, conversely, have to be valued more than the others.

Scheduler training phase has proved to be an important mean to further improve performance and energy efficiency, aiding to chose the weights configuration that minimize mean energy consumption and execution time while running the desired workloads. During the experiments we also shown that, in certain cases, weights combinations can be found such that the objectives are minimized in all the possible workloads, providing a degree of optimization very difficultly reachable without automatic testing tools.

## 5.2 Future developments

Our proposal threw a new light on how resource managers for multi-core and, most of all, many-core devices exploited to execute critical applications can compute how to co-scheduling applications. This is a first step towards a better characterization of applications in a resource-aware point of view, open to optimizations and extensions.

### 5.2.1 Migration and reconfiguration vs optimal co-scheduling

One of the possible developments is the investigation on the trade-off between migration and reconfigurations penalties and performance improvements - viewed as both execution time and energy efficiency - due to effective co-scheduling options. This is very interesting, especially in the case of many-core devices exploited to execute a critical set of known applications. In this case, given the relative low number of scheduling calls, these penalties are minimal with respect of applications runtime, during which resource-aware co-scheduling cause considerable benefits. In fact, taking as an example the BarbequeRTRM case, scheduling is needed only when specific events happen, and not at close and defined times. So, we can run a huge number of applications having only a few scheduler calls.

CoWs scheduling policy has been proved to be faster and more energy efficient than *YaMS* policy alone. One could wonder if he could afford to *encourage* migration and reconfiguration of applications when the evaluation shows that an application should be executed in ways that can lead to better resource utilization. Could the time gains achieved by *CoWs* succeed in masking both their correlated penalties? This is not a totally unmotivated idea; in Sec. 4.2.3 on page 64 and Par. 4.3.2 on page 70 we shown that the importance of *bound_mix* overshadows the one of migration during the binding evaluation. Mind that this is not a trivial study but, just to answer to our question, we tried setting to zero the reconfiguration weight exploited by *YaMS* and the migration one exploited by *CoWS*. We performed the tests on the Intel processor; we already knew from the previous tests that the optimal *migration* weight for our workloads is $M = 1$, but setting these metrics to zero gave us an idea of how a co-scheduling policy totally neglecting the above mentioned aspects would perform. The results in Fig. 5.1 show that this could be a valid idea, electable to be investigated further in the future. EDP improvements amount to 59% and 52% with light and medium loads respectively, 5% and 7% with heavy and congestion loads with respect of *CoWs* scheduler, hinting that, probably, even *reconfiguration* metric from *YaMS* should be evaluated with more attention. Obviously, migration and reconfiguration metrics cannot be discarded a priori; it all depends on the type of applications composing the workload and on the device where it is executed. A serious study of this aspect is certainly needed in future works.

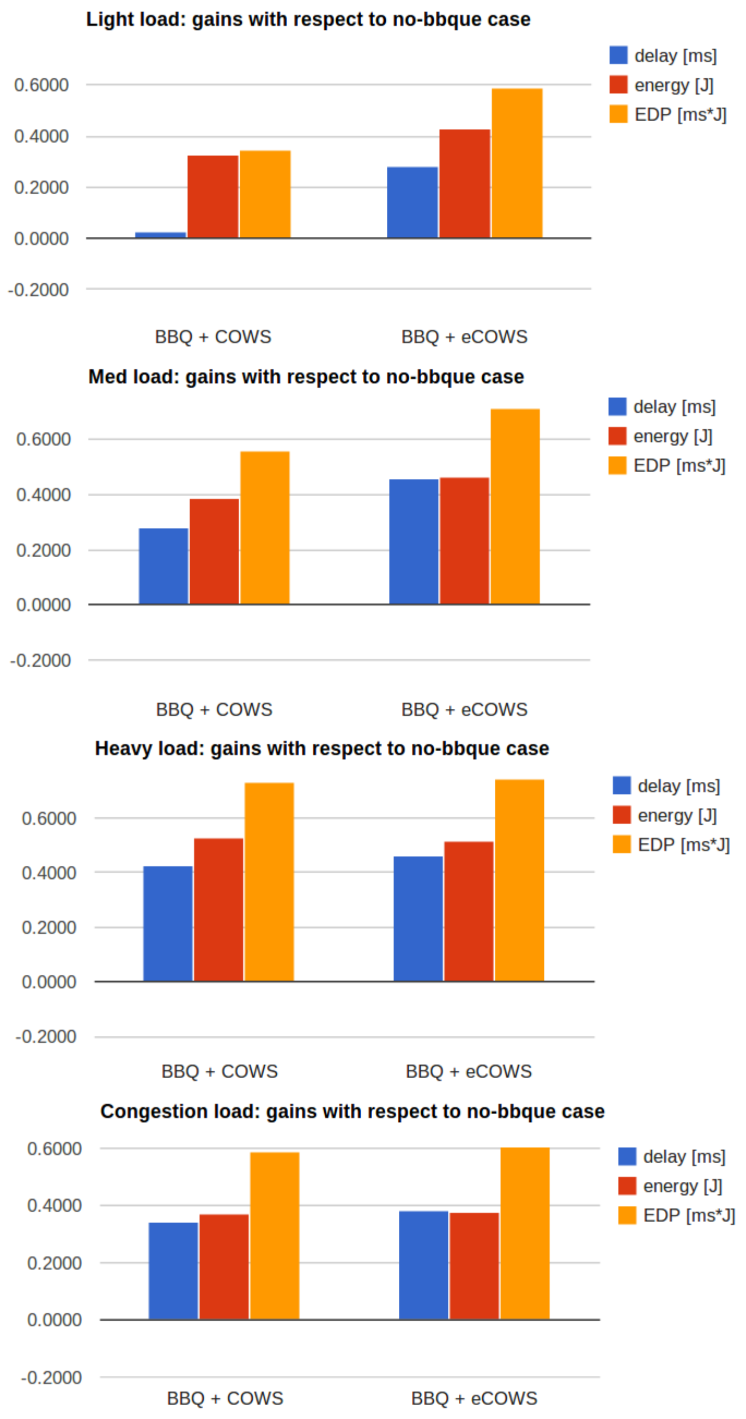


Figure 5.1: Results of *CoWs* support with or without migration and reconfiguration encouraging, with respect to the Linux case

### 5.2.2 From training to learning

The training phase of the applications, that is, the test during which an application is executed to extract resource-utilization information, has to be performed before fully exploiting the system. Unfortunately, this can't be changed because, as mentioned, the application has to be analysed when executing alone on the system. The third phase of the flow described in Par. 3.3.3 on page 49 - the scheduler training - could be, conversely, be transparent to the user. A very interesting idea is the one to exploit a learning approach instead of a static training. During execution time, the system could learn and compute how different sets of applications are to be executed to minimize execution time and energy consumption, changing dynamically the weights exploited during the multi-objective computation of the bindings.

What can be done during the training phase of the application, on the other hand, is a further optimization of the first phase of the flow, that is, the preliminary study of the application. We already noted that BarbequeRTRM doesn't provide means to aid the user during the recipe creation; what could be developed is a service, provided by the framework, to automatize the execution of an application with increasing resource utilization which yields to the user the best possible AWMs, based on the user-provided goals, already containing resource utilization information. This could definitely lead the user to a higher level of experience because, if integrated with the learning feature mentioned above, this could provide the user *for free* recipe creation and *CoWs* support exploitation. What the user would have to do is to integrate the application with BarbequeRTRM as without *CoWs* usage.

Appendix A

# Tools

This appendix will introduce the tools which had been exploited during information extraction and analysis. After a brief explanation of the BarbequeRTRM capabilities in terms of performance counters extraction and aggregation, two tools will be introduced: the first tool is *MOST* (Multi-Objective System Tuner), a tool for architectural design space exploration developed at DEI - Politecnico di Milano. It aided us to automatize the tests, collect and analyse information in order to find the best configurations for our application (ideal number of threads, best resource-efficiency/performance compromises, most energy-saving configurations and so on). The second is *Likwid*, a collection of easy to use but yet powerful performance tools for the GNU Linux operating system. It provides a wrapper able to extract numerous statistics from an application execution (energy and power divided per core or per core-group, IPC an many others) on Intel architectures.

## A.1 Performance counters sampling with RTLib

The RTLib is a component which define the interface and the services exported by the Barbeque Run-Time library. As breefly shown in Chap. 2 on page 13, this library needs to be linked by application that needs to interact with the Barbeque Run-Time Resource Manager (BarbequeRTRM). Additionally. it masks the platform specific communication channel between controlled applications and the run-time manager. The RTLib offers also a set of utility services which can be used alone or combined to develop easily more advanced run-time management strategies. Among them, the one we are going to use, is the metrics collector. During the execution of an application, this component collects various metrics and returns them to the user. Among them hardware, software and cache related events are present. Unfortunately, the common usage of this component expects a metrics group to be selected. Done this, all the metrics in the group are sampled. Sampling too much metrics can lead to accuracy issues. To be more clear, sampling too much counters impose some registers to operate multiplexing and sample multiple counters at the same time, so reducing sampling accuracy. Here comes in aid a helpful feature of the collector: the sampling by means of raw hardware event specification. This feature, which exploits perf tool, consists in specifying the raw event code as specified by the processor developer manual. This lead to two advantages: first of all, even when an event is not available in a symbolic form it can be encoded and sampled. Second, if an user needs only a little set of counter, he can specify them one by one, avoiding the performance losses mentioned above. Having implemented and testing the scheduler and the design flow components on an Intel 2nd generation i7 processor and on a device consisting of AMD Opteron 10h family processors, to find out which event codes where needed to extract the counters selected in the previous chapter we referred to [1] and [3] . Here follows a sampling example for the execution of *bodytrack* application from PARSEC benchmark, executed on 260 frames.

```
[BOSPShell] \> bosp-parsec21-bodytrack
PARSEC Benchmark Suite Version 2.1
[...]
Processing frame 0
[...]
Processing frame 259

.:: MOST statistics for AWM [ps21_btrack:01]:
@ps21_btrack:01:perf:cycles_cnt=260@
@ps21_btrack:01:perf:cycles_min_ms=67.421@
@ps21_btrack:01:perf:cycles_max_ms=594.123@
@ps21_btrack:01:perf:cycles_avg_ms=154.101@
@ps21_btrack:01:perf:cycles_std_ms=69.442@
@ps21_btrack:01:perf:task-clock=74.990661@
@ps21_btrack:01:perf:cpu_utiliz=0.487@
[...]
@ps21_btrack:01:perf:ipc=2.12@
@ps21_btrack:01:perf:stall_cycles_per_inst=372651735@
@ps21_btrack:01:perf:instructions_pct=7.91@
@ps21_btrack:01:perf:instructions_pcu=84.56@
@ps21_btrack:01:perf:branches=45334701@
```

```
@ps21_btrack:01:perf:branches_pct=7.59@
@ps21_btrack:01:perf:branches_pcu=83.47@
@ps21_btrack:01:perf:branch-misses=78498@
@ps21_btrack:01:perf:branch-misses_pct=14.02@
@ps21_btrack:01:perf:branch-misses_pcu=83.21@
@ps21_btrack:01:memory:cache=260333568@
@ps21_btrack:01:memory:rss=73728@
@ps21_btrack:01:memory:mapped_file=0@
@ps21_btrack:01:memory:pgpgin=521103@
@ps21_btrack:01:memory:pgpgout=457527@
@ps21_btrack:01:memory:pgfault=306739@
@ps21_btrack:01:memory:pgmajfault=22@
[...]
```

## A.2 MOST: a tool for design space exploration

MOST (Multi-Objective System Optimizer) is mainly a tool for architectural design space exploration. It lets the designer explore a design space of configurations for a particular system, provided that a simulator of that system is available. However, it can be used in general even for software design space exploration, testing and validation. In fact, every piece of code able to receive inputs and save a log containing outputs can be viewed as a simulator. MOST can be extended by introducing new optimization algorithms such as Monte Carlo optimization, sensitivity based optimization and, as an example, Taguchi design of experiments. All of this by using an appropriate API. The aim of this framework is to drive the designer towards near-optimal solutions to the architectural exploration problem, with the given multiple constraints. The final product of the framework is a Pareto curve of configurations within the design evaluation space of the given architecture. The proposed DSE framework is flexible and modular in terms of: target architecture, system-level models and simulator, optimization algorithms and system-level metrics.

The framework architecture, showed in fig Fig. A.1 , is fairly complex. However, from our point of view, the framework behaviour can be simplified and described as follows (see Fig. A.2 ): a wrapper provides inputs and extracts outputs from the simulator. The input range and the outputs which will be extracted are to be indicated in an user-defined xml file. Input range can be also modified defining constraints, and various methods of exploration are available. For example, one can decide to test every possible input in the range, that is, performing a full exploration (very long if the input range is huge), or to perform a random exploration choosing an arbitrary number of input values to be tested, or to use algorithms specialized in choosing which input values are to be tested and which are to be skipped. The final product is defined in a script or inserted during the process on MOST shell: a database of points - one for every execution of the simulator - is created. This database can be optimized, for example performing clustering, defining minimization objectives and exploiting Pareto optimization to remove dominated points, and so on. This lead the designer to find out which input combination (for architectural exploration, this corresponds to an architecture definition) is the best in order to achieve the defined goals.

As the reader may have probably already guessed, this tool will take the main role during the training phase of the resource-aware co-scheduling work. Treating BarbequeRTRM as a simulator, an application can be executed multiple times with different running configurations (for example, CPU quota at disposal and number of threads), trying to minimize performance and other desired objectives. The desired outputs, obviously, are the collected metrics for every execution. For more details, see Chap. 4 on page 53 .

Regarding the *design space exploration*, that is, the way all possible combinations of inputs are explored, MOST features a number of exploration methods. A full search is not always possible, because executing the simulator taking into all the possible combinations can require huge amounts of time. So, the user can choose to explore only a part of the possible designs. Here follow the exploration methods featured by MOST.

❏ Full search: MOST generates all the designs of the design space.

❏ Neighbourhood: MOST generates all the designs in the neighbourhood of a design point.

❏ Random: MOST generates a set of random design points.

❏ Replica: MOST generates a set of design points that replicate those existing in a database.

❏ Scrambled: MOST generates Full Factorial-like designs of experiments for masks and permutations.

❏ Full factorial: MOST generates Full Factorial design of experiments.

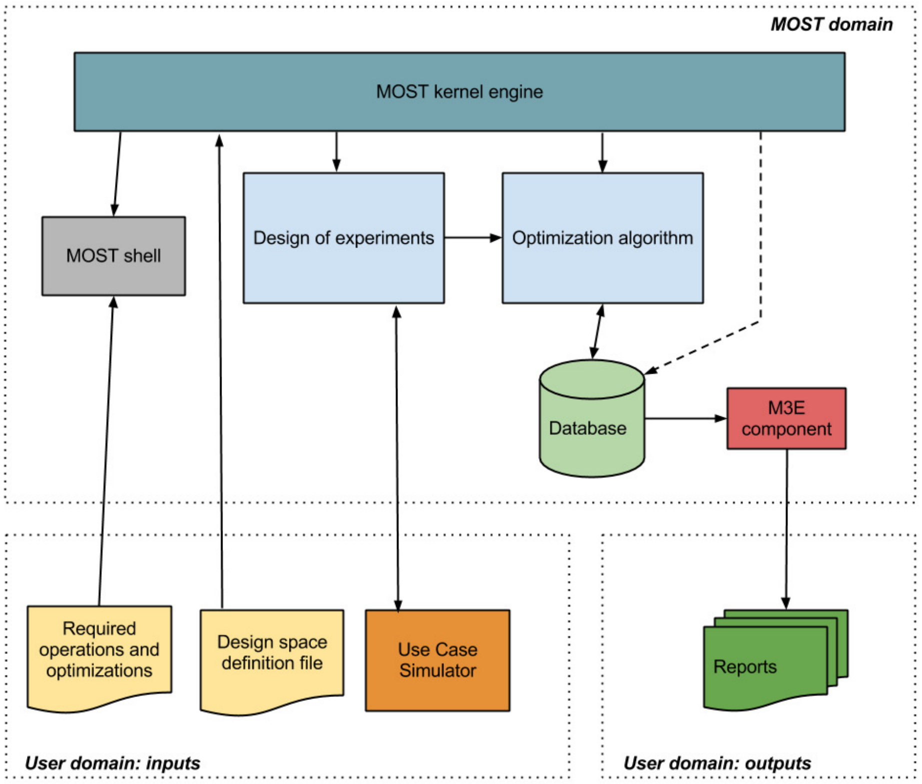


Figure A.1: MOST architecture

- ❏ Full factorial extended: similar to the plain Full factorial design but with a different twist on the generation of opposite values for permutations and masks.
- ❏ Central composite: MOST generates design by using the Central Composite Design technique.
- ❏ Box Behnken: MOST generates design by using the Box Behnken technique.

Our tests exploited the full search design of experiments, because using only parallelism level as an input, it took no huge amounts of time to run exhaustive tests. For detailed information regarding the design of experiments, refer to MOST manual [2].

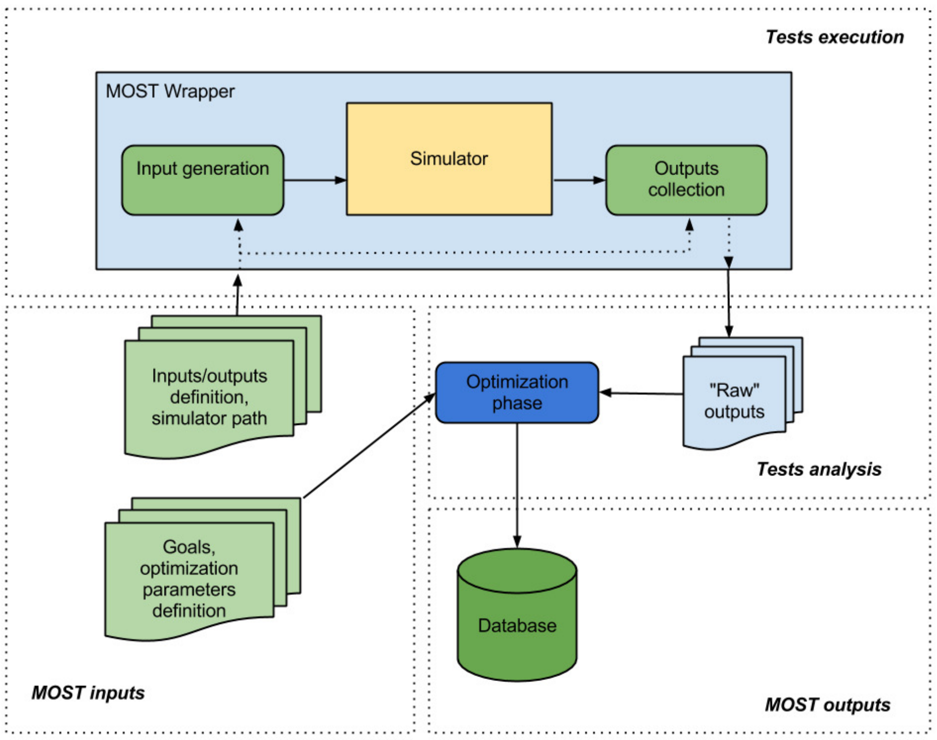


Figure A.2: MOST simplified behaviour

## A.3 LIKWID: Lightweight performance tools

Likwid stands for *Like I knew what I am doing*. The project gives us easy to use command line tools for Linux to support programmers in developing high performance multi threaded programs. No kernel patching is required, any vanilla linux 2.6 or newer kernel works. Additionally, the tools are lightweight, easy to use, simple to build, and there is no need to touch application code code. In the same manner as with the RTLib we can find, among the tools, an useful one to extract information from application execution.

For completeness sake, here is the list of the tools featured by Likwid:

- likwid-topology: Show the thread and cache topology.
- likwid-perfctr: Measure hardware performance counters on Intel and AMD processors.
- likwid-features: Show and Toggle hardware prefetch control bits on Intel Core 2 processors.

- likwid-pin: Pin your threaded application without touching your code (supports pthreads, Intel OpenMP and gcc OpenMP).
- likwid-bench: Benchmarking framework allowing rapid prototyping of threaded assembly kernels.
- likwid-mpirun: Script enabling simple and flexible pinning of MPI and MPI/threaded hybrid applications.
- likwid-perfscope: Frontend for likwid-perfctr timeline mode. Allows live plotting of performance metrics.
- likwid-powermeter: Tool for accessing RAPL counters and query Turbo mode steps on Intel processor.
- likwid-memsweeper: Tool to cleanup ccNUMA memory domains.

The tool we are going to exploit in our work is *likwid-perfctr*. This tool features a wrapper which can extract and compute information about:

- Retired instructions
- Unhalted core and reference cycles
- Runtime
- CPI
- Muops and MFLops/s
- Branch prediction miss rate/ratio
- Load to store ratio
- Power and Energy consumption
- L2 cache bandwidth in MBytes/s
- L2 cache miss rate/ratio
- L3 cache bandwidth in MBytes/s
- Main memory bandwidth in MBytes/s
- TLB miss rate/ratio

The goal of our work is to show that running several applications at the same time choosing their binding in a resource-aware perspective can lead to enhanced performance (reduces the loss of performance due to simultaneous applications run) and energy efficiency (less hotspots, less stalls, less bottlenecks). This tool give us access on all the information we need. This information can be sampled on custom partitions of the resources, and the statistics are provided with total and per-core values.

Here follows, as an example, the sampled information about cores 0 and 3, during the execution of BarbequeRTRM makefile.

```
[BOSPShell BOSP] \> sudo likwid-perfctr -g ENERGY -c 0,3 make
-------------------------------------------------------------
-------------------------------------------------------------
CPU type: Intel Core SandyBridge processor
CPU clock: 2.20 GHz
Measuring group ENERGY
-------------------------------------------------------------
make
==== Checking BOSP build configuration ====
BOSP configured for a Generic-Linux platform
SUCCESS: all required tools are available.

==== Checking building system dependencies ===
Checking for git.......... /usr/bin/git
Checking for cmake........ /usr/bin/cmake
Checking for configure.... /usr/bin/autoconf
Checking for autoreconf... /usr/bin/autoreconf
Checking for libtoolize... /usr/bin/libtoolize
Checking for make......... /usr/bin/make
Checking for doxygen...... /usr/bin/doxygen
Checking for gcc.......... 4.6

==== Setup build directory [/home/stryke/BOSP/out] ====
[...]
==== Installing PARSEC Benchmark Suite (v2.1) ====
Target folder: /home/stryke/BOSP/out/usr/bin/parsec
+-----------------------+-------------+-------------+
|         Event         |   core 0    |   core 3    |
+-----------------------+-------------+-------------+
|   INSTR_RETIRED_ANY   | 1.3631e+11  | 1.21141e+11 |
| CPU_CLK_UNHALTED_CORE | 1.81368e+11 | 1.64499e+11 |
| CPU_CLK_UNHALTED_REF  | 1.46496e+11 | 1.33399e+11 |
|    PWR_PKG_ENERGY     |    4303     |      0      |
+-----------------------+-------------+-------------+
+----------------------------+-------------+...+-------------+
|           Event            |     Sum     |...|     Avg     |
+----------------------------+-------------+...+-------------+
|   INSTR_RETIRED_ANY STAT   | 2.57451e+11 |...| 1.28726e+11 |
| CPU_CLK_UNHALTED_CORE STAT | 3.45868e+11 |...| 1.72934e+11 |
| CPU_CLK_UNHALTED_REF STAT  | 2.79895e+11 |...| 1.39948e+11 |
|    PWR_PKG_ENERGY STAT     |    4303     |...|   2151.5    |
+----------------------------+-------------+...+-------------+
+----------------------+---------+---------+
|        Metric        | core 0  | core 3  |
+----------------------+---------+---------+
| Runtime (RDTSC) [s]  | 171.895 | 171.895 |
| Runtime unhalted [s] | 82.6277 | 74.9427 |
|     Clock [MHz]      | 2717.51 | 2706.74 |
|         CPI          | 1.33055 | 1.35792 |
|      Energy [J]      |  4303   |    0    |
|      Power [W]       | 25.0327 |    0    |
+----------------------+---------+---------+
+---------------------------+---------+...+---------+
|          Metric           |   Sum   |...|   Avg   |
+---------------------------+---------+...+---------+
| Runtime (RDTSC) [s] STAT  | 343.79  |...| 171.895 |
| Runtime unhalted [s] STAT | 157.57  |...| 78.7852 |
|     Clock [MHz] STAT      | 5424.25 |...| 2712.12 |
|         CPI STAT          | 2.68847 |...| 1.34424 |
|      Energy [J] STAT      |  4303   |...| 2151.5  |
|      Power [W] STAT       | 25.0327 |...| 12.5163 |
+---------------------------+---------+...+---------+
```

Chapter 6

# Estratto in lingua italiana

Oggi la maggior parte dei sistemi digitali in commercio fa uso di processori multi-core. Il concetto è oramai consolidato; basti pensare che più di dieci anni fa, nel 2001, è stato messo in commercio il primo processore general purpose con più di un core sulla stessa basetta di silicio: il processore POWER4 della IBM [12]. Tale processore sfruttava due core, il che non si era mai visto se non in sistemi embedded. Da allora, i processori multi-core hanno velocemente sostituito quelli a singolo core sul mercato. Il fatto non è particolarmente sorprendente, dato che sfruttare core multipli è diventato uno dei pochi metodi efficaci per raggiungere buoni speed-up, dato che la riduzione del tempo di clock ha sofferto una notevole frenata dopo decenni di aggressivi miglioramenti [4].

## 6.1 Da core singolo a multi-core

Sfruttare più core sulla stessa basetta porta a numerosi vantaggi. Ad esempio, la condivisione di parte delle risorse e della circuiteria porta ad un buon risparmio in termini di area e permette alle aziende di creare prodotti con minor rischio di errori di progettazione rispetto a quelli dovuti alla creazione di core indipendenti. Inoltre, evitare ai segnali di viaggiare fuori dal chip diminuisce la degradazione dei segnali, consente alla circuiteria di cache-coherency di operare a frequenze più alte (la circuiteria di snooping, ad esempio, può funzionare molto più velocemente), e riduce i consumi di potenza. Infatti, sia evitando ai segnali di viaggiare off-chip sia sfruttando una serie di piccoli core invece di un core grande e monolitico porta ad un consumo energetico notevolmente inferiore [15], e questa è una caratteristica molto importante visto il recente trend verso telefonia mobile ed embedded computing.

Il crescente numero di core, purtroppo, ha portato a gravi problemi. Come per il tempo di clock delle CPU a processori a core singolo, il numero di core nei processori multi-core non può raggiungere valori arbitrariamente elevati. Ad esempio, sfruttando troppi core sulla stessa basetta porta a congestioni di comunicazione ed elevati consumi di energia. Negli ultimi anni, la tendenza si sta muovendo da multi-core a many-core. L'idea è quella di sfruttare un numero elevato di elementi di elaborazione performanti ed a bassa potenza (PEs) connessi da una network-on-chip (NoC). Quest'ultima fornisce un'infrastruttura che garantisce modularità, scalabilità, tolleranza ai guasti, e maggiore larghezza di banda rispetto alle infrastrutture tradizionali. Da ora in poi, con il termine *processore* si farà riferimento ad un processore multi-core o many-core, che quindi sfrutti più di un PE.

## 6.2 Co-scheduling di applicazioni su processori multi-core

Gli avanzamenti tecnologici nell'ambito dei processori hanno aperto nuovi ed interessanti scenari; uno di questi è lo studio del concetto di co-scheduling, introdotto per la prima volta nel 1982 da J. K. Ousterhout [16], e ora più attuale che mai. Avendo a disposizione due, quattro, dodici PE (o anche di più, nel caso di processori many-core), il concetto di scheduling è evoluto nel concetto di co-scheduling. Mentre l'obiettivo principale di uno scheduler è quello di effettuare multitasking (eseguire più processi in un breve lasso di tempo) e multiplexing (trasmettere più flussi di dati contemporaneamente), un co-scheduler mira a schedulare in parallelo differenti tasks su un dato set di risorse. Mentre molti sforzi sono stati compiuti nella ricerca in ambito di programmazione parallela e gestione delle risorse, poche soluzioni originali sono state introdotte per quanto riguarda euristiche e algoritmi di co-scheduling. Che co-schedulare le applicazioni in modo aleatorio sulle risorse non sia la soluzione ottimale è sicuramente ovvio; ogni applicazione ha caratteristiche differenti specie per quanto riguarda lo stress a cui sottoporrà le risorse del sistema e il degrado di performance che causerà alle altre applicazioni in esecuzione su un dato PE. Quindi, co-schedulare due applicazioni sullo stesso PE non può essere considerato una buona scelta a priori. Tutto dipende, banalmente, da quali applicazioni voglio eseguire e dal tipo di sistema che sto utilizzando. Sfortunatamente, ad ogni modo, determinare quali applicazioni debbano o non debbano essere eseguite sullo stesso PE non è l'unico problema che ci si para davanti; la maggior parte degli scheduler dei sistemi operativi odierni vede ogni PE come un processore singolo e isolato dal resto del sistema. Questo non è necessariamente vero, dato che ci sono molte interdipendenze tra i PE. Ad esempio, si potrebbe pensare che due applicazioni caratterizzate da un notevole utilizzo di dati non debbano venire eseguite dullo stesso PE, dato che ambedue saranno spesso in stallo aspettando i dati e creeranno contention sulle cache del PE. Ma se eseguissi tali applicazioni su due PE differenti, che però condividono una buona parte della gerarchia di cache? Questo porterebbe comunque a contention e causerebbe rallentamenti nell'esecuzione delle applicazioni e un utilizzo non efficiente dell'energia, dato che i core in stallo a causa della contention sprecano cicli di clock senza mandare avanti il loro lavoro. Si noti che, in alcuni sistemi, le gerarchie di cache non sono condivise tra tutti i core. Pensiamo alle architetture NUMA (accesso a memoria non uniforme), dove i core sono raggruppati in nodi e ogni nodo ha la sua gerachia di cache: qui, effettivamente, eseguire applicazioni in modo tale che quasi non vi sia contention è possibile. Tuttavia, il problema persiste; essendo il numero di nodi limitato da tecnologia e problemi di scalabilità, anche nel caso di carichi di lavoro piuttosto leggeri non è possibile eseguire un'applicazione per ogni nodo. Quello che possiamo fare, piuttosto, è sincerarci che le nostre scelte di co-scheduling causino degradi di performance il meno pesanti possibile. La contention sulla memoria è sicuramente la causa fondamentale dei degradi. Tuttavia, ne potremmo trovare molte altre minori; in un certo senso, ciò che un co-scheduler dovrebbe fare non è cercare una allocazione *fair* delle applicazioni sui PE, ma distribuire in modo *fair* l'utilizzo di risorse e i possibili degradi di performance sull'intero sistema.

### 6.2.1 Co-scheduling basato sull'utilizzo di risorse

Come già menzionato, il problema del co-scheduling riguarda la scelta di come eseguire $m$ applicazioni su $n$ PE, con $m$ maggiore di $n$. Dal nostro punto di vista, ogni PE non è totalmente isolato; ognuno di essi condivide un certo set di risorse con alcuni degli altri. Certamente la prima risorsa da analizzare è la gerarchia di cache. Lo scenario più banale potrebbe essere quello in cui $n$ applicazioni, caratterizzate dallo stesso tempo di esecuzione, eseguono in modo parallelo su $n$ PE. In questo caso, si potrebbe pensare che il possibile speed-up, rispetto al caso seriale, sia $n$. Essendo però i PE non indipendenti, specie per quanto riguarda le gerarchie di cache, abbiamo livelli di cache dove più applicazioni leggono e scrivono dati in modo concorrente. Utilizzare una cache condivisa è una notevole fonte di degrado di performance. Pensiamo al caso in cui tutte le applicazioni sfruttino la cache in modo *fair*: ipotizzando che ogni PE non possa leggere e scrivere su aree di memoria utilizzate dagli altri PE, se tale cache è condivisa tra $n$ PE ogni PE "vede" solo $1/n$ della cache. Questa riduzione virtuale delle dimensioni della cache porta - come il lettore certamente già immagina - a un innalzamento massivo del cache miss ratio, ovvero della percentuale di accessi in cache falliti sul totale delle richieste. E per quanto riguarda il co-scheduling di più applicazioni sul medesimo PE? Qui la situazione è ancora più critica; in fatti, potremmo arrivare a scanari dove tutte le applicazioni vanno in stallo aspettando dati, a causa dei degradi appena menzionati. In questo caso, il PE starebbe sprencando cicli, non essendo in grado di eseguire alcuna applicazione. Il problema è complesso e, sopratutto, grave: può infatti portare a situazioni dove l'esecuzione parallela andrebbe addirittura evitata, dato che lo speed-up ottenuto è così basso da non essere vantaggioso rispetto alle spese sostenute per acquisire un dispositivo ad elevato numero di core.

Come possiamo ridurre i degradi di performance dovuti alla condivisione delle risorse? Tipicamente, questi degradi dipendono dal carico di lavoro da eseguire, ovvero dalle caratteristiche delle singole applicazioni che verranno eseguite sul sistema. Negli ultimi anni, un cospicuo numero di soluzioni è stato proposto. Quasi tutte queste soluzioni hanno bisogno di una *fase di apprendimento*, ovvero di una fase in cui le applicazioni vengono eseguite una ad una sul sistema, per monitorare il loro comportamento e il loro effettivo utilizzo di risorse a prescindere da eventuali altre applicazioni in esecuzione sul sistema. Le informazioni raccolte vengono poi utilizzate dalle politiche di scheduling per scegliere, a run-time, il co-scheduling ottimo delle applicazioni che sta eseguendo.

### 6.2.2 Co-scheduler basati sull'utilizzo di risorse negli ultimi anni

In [9], un articolo datato ma nondimeno interessante, è proposto un approccio che non richiede fase di apprendimento. Il concetto è l'identificazione, a run-time, di gruppi di attività caratterizzate da un alto grado di mutua interazione. L'idea è quella che applicazioni che accedono spesso alle stesse aree di memoria - condividendo quindi tali dati - sono probabilmente caratterizzate da un'interazione forte. Si noti che il set di risorse, qui, ammonta a un singolo core. Tuttavia l'articolo mostra come, anche vent'anni fa, l'idea di prestare attenzione a quali gruppi di applicazioni dovessero venir eseguite sul medesimo PE era presente nella letteratura.

Un altro lavoro interessante si concentra sull'utilizzo del bus in sistemi a multiprocessori simmetrici (SMP) [1]. Questo approccio, proposto in [5], mira a ridurre i colli di bottiglia dovuti alla congestione del bus di sistema. Dimostrano che, sfruttando la richiesta di banda dei thread raccolta durante la fase di apprendimento per oprerare le propre scelte, lo scheduler è in grado di evitare sovra e sotto-utilizzi del bus, riducendo quindi i degradi di performance durante l'esecuzione di carichi pesanti. Lo scheduler è validato eseguendo due diversi benchmarks: il primo è caratterizzato da un intenso utilizzo del bus, il secondo da un utilizzo leggero. La piattaforma sperimentale è un SMP composto da quatto processori Intel Xeon, con hyperthreading e clock a 1.4 GHz. È dotato di 1 GB di memoria principale, e ogni processore dispone di 256 KB di cache L2. Il bus di sistema della macchina ha una frequenza di 400MHz. Il sistema operativo è Linux, versione del kernel 2.4.20. Il loro approccio porta ad incrementi nella velocità massima di esecuzione del 26 % rispetto allo scheduler di Linux.

Durante la fase di apprendimento, le informazioni relative all'utilizzo del bus vengono raccolte tramite performance counters, metodo principale in tutti questi lavori. Ci si potrebbe chiedere quali contatori siano più adatti ad essere sfruttati durante la fase di apprendimento; in [8] vengono analizzati diversi contatori, cercando di rispondere appunto a questa domanda. Il loro studio, effettuato su un sistema che sfrutta due processori dual-threaded, è incentrato sul confronto tra cinque scheduler: RFU scheduler (incentrato sull'utilizzo del register file), FRCA scheduler (incentrato sui conflitti nel register file), DCCS scheduler (incentrato sui conflitti nella L1 data cache), IPCS scheduler (incentrato sulle istruzioni per secondo), e RIRS scheduler (incentrato sulle operazioni pronte ed in esecuzione simultaneamente). Ognuno di questi scheduler sfrutta differenti contatori per caratterizzare le applicazioni e calcolare il co-scheduling ottimale. Eseguendo carichi di lavoro eterogenei, dimostrano che i degradi massimi di prestazione di RIR sono pari al 2 %, mentre raggiungono il 10 %, 13 %, 11 % e il 14 % con DCCS, RFU, RFC, IPCS rispettivamente. Così, informazioni generali come il numero di istruzioni pronte e in volo permettono di effettuare decisioni di co-scheduling molto più consistenti di quelle basate su cache di primo livello e register file.

In [13], l'attenzione si è spostata verso l'utilizzo della memoria. Qui due importanti concetti vengono introdotti: prima di tutto, eseguendo diversi benchmarks da

[1]Qui ci riferiamo ad architetture composte da due o più processori identici connessi a un'unica memoria condivisa. La comunicazione è basata su bus o crossbar.

SPEC CPU 2006 con uno scheduler di Linux modificato (kernel 2.6.16), mostrano che decisioni di co-scheduling basate sull'uso della memoria possono portare a prestazioni e miglioramenti dell'efficienza energetica. In secondo luogo, scelgono il prodotto energia-tempo di esecuzione (EDP) come metrica di valutazione per le loro scelte di co-scheduling. Il loro scheduler permette riduzioni massime di EDP pari al 10% rispetto allo scheduler standard di Linux, i test eseguiti utilizzando un sistema multiprocessore IBM xSeries 445 composto da otto processori Pentium 4 Xeon Gallatin a 2.2 GHz. Il sistema è costituito da due nodi NUMA con quattro processori multi-threaded a due vie su ciascun nodo. La scelta dello EDP come metrica per valutare la bontà di una scelta di co-scheduling è importante; [11] mostra che lo EDP è una metrica relativamente neutra rispetto all'implementazione e riesce a mettere in luce in modo chiaro i miglioramenti che portano beneficio sia alle performance, sia all'efficienza energetica. In altre parole, utilizzando lo EDP abbiamo a disposizione una singola metrica, che però ci aiuta a caratterizzare contemporaneamente prestazioni ed efficienza energetica.

Le opere di cui sopra mirano a dimostrare che alcune statistiche di utilizzo delle risorse sono particolarmente adatte ad essere prese in considerazione durante la fase di co-scheduling. Il principale punto debole di queste opere è che prendono in considerazione un solo tipo di risorsa; ogni lavoro dimostra che gli altri non tengono conto di risorse il cui uso eccessivo è stato dimostrato essere causa di degrado di prestazioni e maggiore consumo di energia. Per rispondere a questo problema, in [14] viene introdotto il concetto di *vettore di attività*. Un vettore di attività descrive in che misura un'applicazione utilizzi varie risorse legate al processore, e la dimensione di questo vettore è pari al numero di risorse vogliamo considerare. Ciascun componente del vettore denota il grado di utilizzo della risorsa corrispondente. L'obiettivo della loro politica è quello di co-schedulare le applicazioni in modo che ogni risorsa nel sistema venga utilizzata da applicazioni che hanno un grado di utilizzo di tale risorsa molto differente tra loro, in modo da evitare degradi di performance dovuti a colli di bottiglia e stalli. Per studiare gli effetti della contention sulle risorse e della selezione delle ottimali frequenze di funzionamento dei processori, scelgono un Processore Intel Core 2 Quad Q6600 a 2.4GHz. Quando non possono essere trovate combinazioni di applicazioni vantaggiose, la contention è mitigata dal frequency scaling. Questo permette di ottenere fino al 21% di risparmio in EDP relativamente all'esecuzione di varie applicazioni dal benchmark SPEC CPU 2006.

Negli ultimi anni sono stati proposti molti approcci al problema del co-scheduling orientato all'utilizzo ottimale delle risorse per la riduzione di consumo energetico e degradi di performance. Tuttavia, molti problemi devono ancora essere risolti:

❏ Implementazioni reali non vengono spesso presentate. Infatti, spesso i sistemi vengono validati attraverso simulatori.

❏ Nessuno dei sistemi realmente implementati risulta portabile; gli scheduler sono spesso creati modificando il kernel di Linux.

❏ L'unico lavoro che presta attenzione a set ordinariamente grandi di risorse, non offre un metodo per comprendere quali risorse del sistema siano davvero interessanti dal punto di vista della riduzione dei degradi di performance. Peraltro, non vi è modo di calcolare quali siano le risorse più importanti, ad esempio la gerarchia di cache, e quali invece debbano assumere un'importanza minore durante la scelta del co-scheduling ottimale.

❏ Spesso, le informazioni raccolte durante la fase di apprendimento sono dipendenti dall'architettura (ad esempio, il floorplan del processore).

❏ Un flusso standard e completo che guidi l'utente del sistema dalla creazione di un'applicazione (e raccolta dei dati necessari allo scheduling) alla valutazione dell'efficienza dello scheduler non è stato ancora proposto.

A nostro parere, ciò di cui abbiamo bisogno è il design di una politica di scheduling che tenga in considerazione un set di risorse che sia stato provato essere strettamente correlato a degradi di performance e consumo di potenza. Questo approccio dovrebbe essere caratterizzato da portabilità e adattabilità a qualsiasi piattaforma multi-core. Inoltre, deve essere introdotto un flusso completo che aiuti l'utente a compiere tutti i passi che lo porteranno verso un co-scheduling caratterizzato da bassi consumi di potenza e degradi di performance minimi, a partire dalla creazione e ottimizzazione dell'applicazione arrivando al testing delle proprie applicazioni su sistemi reali.

## 6.3 Obiettivi del lavoro di tesi

L'obiettivo di questo lavoro è quello di dimostrare che una politica di co-scheduling che tenga conto dell'utilizzo di risorse delle applicazioni può portare a una vasta gamma di vantaggi, come ad esempio incrementi di prestazioni, miglioramento dell'efficienza energetica, riduzione delle temperature. La politica sarà progettata e implementata come estensione di una delle politiche di co-scheduling sfruttate da BarbequeRTRM [7], un resouce manager altamente modulare, estensibile e portabile sviluppato presso il DEIB - Politecnico di Milano - nell'ambito del Progetto Europeo 2PARMA [17] che fornisce il supporto per la gestione di più applicazioni in competizione per'uso di uno (o più) dispositivi multi-core. Dimostreremo che concentrandosi non solo su quali siano le migliori applicazioni da schedulare ma anche su come co-schedularle, le prestazioni e l'efficienza energetica di tale resource manager, già migliori rispetto a quelle del resource manager di Linux, possono essere portate a livelli ancora più alti. Valideremo la politica, che chiameremo COWs (CO-scheduling WorkloadS), su due diverse piattaforme: la prima sfrutta un processore Intel Core i7 quad-core con hyperthreading, la seconda è un dispositivo NUMA con quattro nodi, ognuno dotato di un processore AMD 10h Opteron 8378 quad-core con gerarchie di cache distinte per ogni nodo. La validazione si basa sul confronto dello EDP per diversi carichi di lavoro composti da applicazioni del benchmark PARSEC v.2.1 dell'università di Princeton, per i quali BarbequeRTRM ha dimostrato di essere in grado di migliorare notevolmente prestazioni ed efficienza energetica rispetto al caso Linux. Durante i test, quindi, le risorse verranno gestite da Linux, da BarbequeRTRM e da BarbequeRTRM con il supporto di COWs. Oltre a ciò progetteremo un flusso standard che guidi l'utente attraverso la creazione di un'applicazione che possa interagire con BarbequeRTRM, la fase di addestramento necessaria all'utilizzo di COWs, l'ottimizzazione dell'applicazione e dello scheduler stesso attraverso un'esplorazione del design space e dei possibili carichi di lavoro, e la valutazione dello EDP.

## 6.4 Organizzazione della tesi

**Nel capitolo 2** verrà introdotto il framework BarbequeRTRM. Una vista di alto livello di questo framework aiuterà il lettore a comprendere i vantaggi offerti da BarbequeRTRM e le motivazioni che ci hanno portato a ritenerlo un eccellente punto di partenza per il nostro lavoro. Dunque, uno degli attuali scheduler di BarbequeRTRM, chiamato YaMS, verrà introdotto. Alla spiegazione del suo funzionamento e dell'idea che sta dietro alla scelta del co-scheduling ottimale seguirà una analisi dei suoi punti deboli in termini di sfruttamento delle caratteristiche di utilizzo di risorse delle applicazioni, e i primi concetti su come tale scheduler possa essere esteso grazie alla nostra politica verranno introdotti.

**Nel capitolo 3** verrà definito il concetto di co-scheuduling basato sull'utilizzo di risorse, concentrandosi su quali contatori possano essere i più utili durante la fase di scelta del co-scheduling. Quindi seguirà la progettazione dello scheduler COWs, definendo le metriche a cui farà riferimento durante la scelta del co-scheduling ottimale e descrivendo la sua interazione con BarbequeRTRM. Il flusso che porterà dalla creazione dell'applicazione alla valutazione dello EDP verrà dunque introdotto.

**Nel capitolo 4** l'implementazione verrà validata, concentrandosi su ogni passo del flusso e spiegando l'interazione tra i vari tools utilizzati. Dunque, i risultati verranno discussi. Vedremo come BarbequeRTRM riesca a garantire prestazioni e consumi di potenza migliori rispetto al caso Linux, e come il supporto di COWs riesca a migliorare ulteriormente questi risultati.

**Nel capitolo 5** trarremo le conlusioni sul lavoro, spiegando brevemente cosa abbiamo fatto e come abbiamo raggiunto tali risultati. Le possibili applicazioni future verranno quindi discusse.

**L'appendice A** introdurrà i maggiori tools utilizzati durante il lavoro, spiegando il loro funzionamento e il vantaggio da loro offerto al nostro lavoro.

# Bibliography

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*.

[2] *Multi-Objective System Tuner - Design Of Experiments Modules Manual*.

[3] *Software Optimization Guide for AMD Family 10h and 12h Processors*.

[4] *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 2005.

[5] C. Antonopoulos, D. Nikonopoulos, and D. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for smps. In *Conference on Parallel Processing*.

[6] P. Bellasi, G. Massari, and W. Fornaciari. A rtrm proposal for multi/many-core platforms and reconfigurable applications. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*.

[7] DEI. Barbequertrm framework. http://bosp.dei.polimi.it/.

[8] A. El-Moursy, R. Garg, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Parallel and Distributed Processing Symposium*.

[9] D. G. Feitelson and L. Rudolph. Co-scheduling based on run-time identification of activity working sets. *International Journal of Parallel Programming*, 1995.

[10] B. Goel, M. Själander, and S. A. McKee. Per-core power estimation for cmps. Chalmers University of Technology, SE.

[11] R. Gonzales and M Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 1996.

[12] IBM. Power4. http://www.research.ibm.com/power4.

[13] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proceedings of the Workshop on Power Aware Computing and Systems*.

[14] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Fifth ACM SIGOPS EuroSys Conference*.

[15] P. et Al. Muthana. Packaging of multi-core microprocessors: tradeoffs and potential solutions. In *Electronic Components and Technology Conference*.

[16] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *IEEE Conference on Distributed Computing Systems*.

[17] 2PARMA Project. Parallel paradigms and run-time management techniques for many-core architectures. http://www.2parma.eu/.

[18] O. Semenov, A. Vassighi, and M. Sachdev. Impact of self-heating effect on long-term reliability and performance degradation in cmos circuits. In *IEEE transactions on device and materials reliability*.

[19] Princeton University. Parsec benchmark. http://parsec.cs.princeton.edu/.

[20] Website. Ipmitool. http://ipmitool.sourceforge.net/.

[21] Website. Perf linux tool. https://perf.wiki.kernel.org/index.php/Main_Page.