POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

EXPLORING PERFORMANCE SCALABILITY OF TASK-AFFINITY ON DIFFERENT MULTICORE ARCHITECTURES

Autore:
Matteo MARINI
matr. 724626

Relatore:
Prof. William FORNACIARI

Tutor aziendale:
Wolfgang BETZ

Correlatore:
Ph.D. Patrick BELLASI

Anno Accademico 2010-2011

Contents

Chapter 1

Introduction

In the quest for the highest CPU performances, hardware developers have to deal with a difficult dilemma. On one hand, Moore's Law does not apply to computational power any more, that is, computational power is no longer doubling every 18 months as in the past. On the other hand, power consumption continues to increase more than linearly with the number of transistors included in a chip, and Moore's Law still holds for the number of transistors in a chip. Several solutions have been adopted to solve this dilemma. Some of them try to reduce the power consumption by sacrificing computational power, usually by means of frequency scaling, voltage throttling, or both. Other solutions try to increase the Instruction Level Parallelism (ILP) inside a processor, in order to get more computational power from the CPU without increasing power consumption. But nowadays the penalty of a cache miss (which may stall the pipeline) or of a miss-predicted branch (which may invalidate the pipeline) has become way too expensive. Already in the early 2000's, it was clear that the most effective way to increase computational power and reduce power consumption was to parallelize task execution. For this reason Simultaneous MultiThreading (SMT) was introduced. It is a technology that allowed to execute concurrently two threads on the same CPU. Nowadays it was introduced multicore technolgy that consist of N superscalar processors put in the same chip. This technology allows a great parallelization of task execution. With reference to memory organisation, multiprocessor systems are classified into two groups:

Centralised Shared Memory Architectures: in this architecture, there are multiple cores connected to a single shared memory. If all cores are equal this architecture is called simmetric multiprocessor (SMP)

Distributed Memory Architecture: in this architecture each processor has its own memory module and memory access time depends on the memory location relative to a processor. Non-uniform memory access (NUMA) processor are included in this category of architecture

Multicore architectures have been adopted by most chip manufacturers. Dual-core chips are commonplace, and numerous four and eight core options exist. In the coming years, per-chip core counts will continue to increase: Intel has claimed that it will release 80-core chips as early as 2013. The shift to multicore technologies is a watershed event, as it fundamentally changes the ”standard” computing platform in many settings to be a multiprocessor.

Even many embedded systems are starting to adopt multicore architectures, because theese processors can provide a large increment of computational power, with small increase in power consumption, that is a very important aspect for this type of systems. But there is an obstacle to the use of these architectures in this sector and in particular in Real-Time systems. In most multicore platforms, different cores share onchip caches. Imagine this situation: there are three Real-time tasks: A, B and C. A uses 512KB of memory, B uses 768KB and C uses 256KB. Our platform has a dual-core chip. Shared onchip cache is of 1MB. There are two possible case of scheduling. In first case A and C (or B and C) are scheduled. There is enough space in cache to alloc their resources, and it is good. In the second case A is scheduled with B: cache trashing occurs. Performance of two tasks could get worse respect the previous case, because there isn't any guarantee that A or B may find data in shared cache and, furthermore, it is impossible to predict A or B duration, because if A is scheduled with B, it will have a certain duration. Instead, if A is scheduled with C, it will have another duration. In other words: a task's duration depends from which other task is scheduled with it and, for this reason, using common Real-Time scheduling algorithms in multicore systems, well developed techinques for timing analysis of embedded software used in single processor systems are no loger useful, therefore new techinques are needed to estimate the worst-case execution time (WCET) of Real-Time tasks for this kind of platforms. It is clear that the scheduler plays an important role to improve performance and predictability of the applications. Nowadays, it is important to develop scheduling algorithms ”cache-aware”, that is a scheduler that, to choose on which cpu to put a task, it consider about how scheduled tasks use cache memory, in order to avoid cache trashing. This thesis is the prosecution of the work carried out by Lucas De Marchi. He has tried to make

the Linux Real-Time scheduler cache-aware introducing the concept of taskaffinity. In this work, I have tried to improve the concept of taskaffinity and I have studied the behaviour of this mechanism on different multicore architectures.

1.1 State of the art

Although the problem to design ”cache aware” scheduling algorithms is an old and well-known problem for over 20 years and multicore processors are largely used, nowadays operating systems don’t implement this kind of algorithms and in literature there are only a few works that study this problem. The most of recent research works related to this argument consist of profiling activities with the aim to demonstrate and build a model that show how an unfair cache sharing between concurrent threads may slow down them and cause sub-optimal throughput, cache trashing and, in some cases, thread starvation for threads that fail to occupy sufficient cache space to make good progress. The first well-documented work related to this kind of scheduling algorithms, was developed at the Stanford University. At the end of 80’s, the Computer Systems Laboratory at Stanford University designed a prototype of a shared-memory multiprocessor called DASH. Its architecture was very similar to that used in modern SMP processors. DASH was able to incorporate up to 64 high-performance RISC microprocessors. In order to exploit full potentiality of this machine, they developed a suitable runtime system to use with DASH, and they designed the COOL language. It was an extension of C++, that introduced some statements to facilitate expression of medium to large grain parallelism and to define which was the *data reference patterns* of the program. The COOL compiler was able to automatically extract fine-grain parallelism for architectures that, like DASH, supported such a level of concurrency, and extract information about use of cache made by applications. Using these informations, the runtime system could ensure parallelism desired by programmer and try to reduce cache miss rate of each task, because this system ”knew”, for each task, which were the objects referenced by it, so it distributed tasks and objects in order to make them close. In plain words, using additional informations provided by the programmer and exploiting the principle of data locality, the runtime system decided where to allocate objects and it assigned a task to a CPU that contained objects referenced by it in its cache. The COOL project shows how the smart use of cache is a problem that involves all aspect of software engineering, from the compiler to scheduler and

memory management system.

Interesting research activities made in recent years exploit another strategy. They don't introduce new programming language or sophisticated runtime environment, but they implement a raw profiler that, at runtime, it infers how much cache space a task requires, in order to infer which tasks could cause cache trashing if they would be executed concurrently. To make this job, the profiler executes a periodical tuning phase in which it analyzes miss rate of each task, in this way it is possible understand the amount of space used by a task. According to these informations, two or more task are scheduled on different CPUs only if they don't cause cache trashing. These works are not effective as COOL project, but they present good results with SPEC2000 and LITMUS [1] benchmarks, furthermore this kind of works was experimented with good outcome also in embedded systems.

1.2 Objectives of this thesis work

The main goal of this thesis is the optimization of the current version of task-affinity. In a first step, we will analyze the behaviour of task-affinity on different multicore architectures, in particular on Intel Xeon E5440 and on Intel i7 870. These architectures have two different cache architectures, furthermore also they have a different inter-chip communications system. With this analysis, we will find which are the aspects of task-affinity logic to improve in order to make the most of task-affinity on tested architectures, for example: in the current version of task-affinity, as described in [1], the migration policy is not very effective, because some task bounce from two CPUs at each iteration of benchmark. We believe that this issue degrades obtainable performance with task-affinity, because a task that migrates has to warm up the cache of CPU on which it has migrated, increasing miss rate. We expect that this analysis put in evidence, for each architecture, what is the incidence of this "migration pattern" on miss rates.

According to results obtained in the analysis phase, we will try to enforce the temporal locality of data, in order to diminish miss rate of the tasks. To do this, we will include in the task-affinity logic functions used by kernel to perform migration of tasks. In the last part of the optimization, we will synchronize the structures used in task-affinity. All measures executed on task-affinity and on vanilla were

[1] It is a Linux-based testbed developed by them that supports multiprocessor real-time scheduling policies within Linux

effectuated using benchmark developed in [1].

Therefore the objectives of this thesis can be summarized as:

1. Analyze the behaviour of the current version of task-affinity on different architectures in order to understand which aspects to improve.
2. Optimize the current version of task-affinity, improving the migration policy and improving the temporal locality of data guaranted by task-affinity.

All changes performed on Linux kernel are based on version 2.6.34 of the vanilla kernel

1.3 Organization of the thesis

Chapter 2: we first discuss the issues due to an incorrect use of cache. We will see how an unfair cache sharing may degrade performance and greatly reduce the determinism of applications. A survey of cache architectures is presented paying attention to those architectural details that often are not well documented, such as: cache coherency protocols, inclusive or exclusive cache etc. In the last section, a classification of recently cache aware scheduling algorithms is presented paying attention to advantages and drawbacks involved by each algorithm analyzed.

In Chapter 3: we discuss the optimization implemented in this work. The first part is an analysis of the behaviour of task-affinity on different architectures, in order to understand how task-affinity can be optimized. Then, the following sections describe the implementation in Linux.

Chapter 4: we presents the experimental results regarding the correct behavior of the solution and the improvements in respect to current version of task-affinity.

Chapter 5: we draw the conclusions on the work summarizing achieved results and proposing possible future development

Chapter 2

Impacts of caches on Scheduling Performance

In multicore platforms cache memory is a very important resource and often the issue is underestimate. Furthermore, in many cases, it is difficult know how to use cache memory in a correct way, because many important implementations details of cache memory are undocumented. The aim of this chapter is to show which are the problems due to an inaccurate use of cache memory and which are the most important factors of cache architecture that influence system performance. The last section of the chapter propose an overview of the most important cache aware policies studied in literature. For each policy is given a briefly description, some example that explain how to implement it and advatages and drawbacks of the policy.

2.1 Issues due to an incorrect use of cache

In most multicore platforms, different cores share onchip caches, usually is L2 or L3 cache. Recent research works show how an unfair cache sharing among concurrent tasks and cache trashing degrades performance. Modern operating systems (OS), attempt to schedule threads in a fair and low-overhead manner. Most simply, they balance the load across processor by migrating task to keep the run queues approximately equal. An OS assumes that, in a given time slice, resource sharing uniformly impacts the rates of progress of all the co-scheduled threads. Unfortunately this assumption is often unmet because how much cache space a thread may use is determined by threads with which it is co-scheduled.

An interesting activity developed by S. Kim et al. [10] cita shows the impact of an unfair cache sharing on performance and how this phenomena is dependent by co-scheduled threads. In Figure 2.1 we can see one of results is provided by that work.

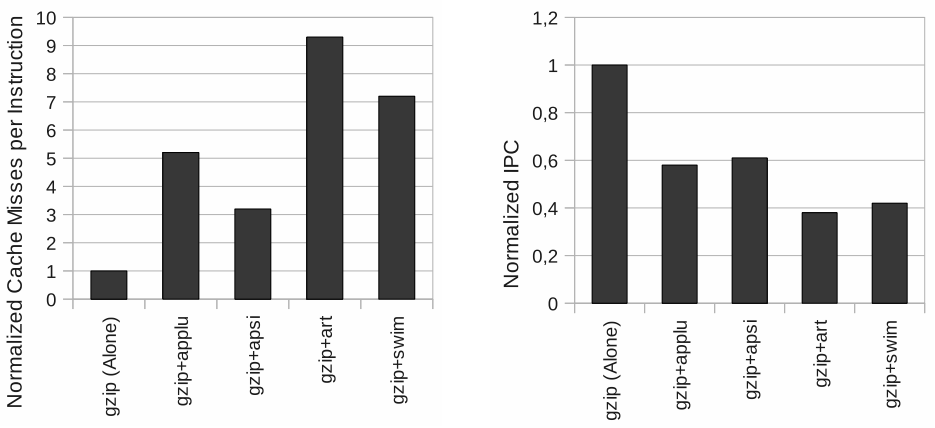


Figure 2.1: gzip miss rate [10]

The picture shows gzip's number of cache misses per instruction and instruction per cycle (IPC), when it runs alone compared to when it is co-scheduled with different threads, such as applu, apsi, art, and swim. All the bars are normalized to the case where gzip is running alone. It is interesting to note how gzip's number of cache misses per instruction increases significantly compared to when it runs alone. In fact, it increases by 3x when it runs with apsi and by 9.5x when it runs with art, 7.3x when it runs with swim. Consequently, the IPC is affected differently. It is reduced by 35% when gzip runs with apsi, but reduced by 63% when gzip runs with art. Although not shown in the figure, art, apsi, applu, and swim's cache miss per instruction increases less than 15% when each of them runs with gzip.

In terms of fairness, gzip's significant slow down can easily result in *priority reduction*. For example, if gzip has a higher priority than art, for gzip to achieve a higher progress rate, it has to be assigned more than three times the number of time slices compared to that assigned to art. Otherwise, to the end users, gzip may appear to be starved. Therefore, even if gzip has higher priority than art, it seems that art run more fast than gzip, it is as if priority of gzip is reduced. In terms of throughput, gzip's significant slow down reduces the overall throughput because the utilization of the processor where gzip runs on is also significantly reduced. Furthermore, it is possible that the co-scheduled threads's working sets severely

overflow the cache and create a *trashing* condition.

In briefly, there are at least three problems that may happen and render the OS schedule ineffective. The first problem is *thread starvation*, which happens when one thread fails in competing for sufficient cache space necessary to make satisfactory forward progress. The second problem is *priority reduction*, where a higher priority thread achieves a slower forward progress than a lower priority thread, despite the attempt by the OS to provide more time slices to the higher priority thread. It's as if the higher priority task and lower priority task have the same priority, that is the priority of higher priority task is reduced. This happens when the higher priority thread loses to the lower priority thread (or other threads) in dealing for cache space. To make things worse, the operating system is not aware of this problem, and hence cannot correct this situation (by assigning more time slices to the higher priority thread). The third problem is that the forward progress rate of a thread is *highly dependent* on the thread mix in a co-schedule. This makes the forward progress rate difficult to characterize or predict, making the system behavior unpredictable. Unfortunately, despite these problems, cache implementations today are thread-blind, producing unfair cache sharing in many cases.

As I said in the first chapter, in these years were developed some ideas about how to make a scheduler cache-aware. This chapter aims to show the general structure and strategies used to model cache behaviour followed by these algorithms, that are the most interesting part of these type of heuristics.

2.2 Survey on cache architecture

Over the years, cache architectures have always played an important role in system performance. Hundreds of research papers show how performance can be improved using multi-level caches on a single-processor machine. Multicore systems introduce new challenges for cache designer, because cache memory is a shared resource, therefore issues related to how a core access to cached data and how the coherence regarding the access from different processors to the same cached data is guaranteed are unavoidable. Furthermore, caches play an important role in management of communication between different core. This section aims to show which are most important factors introduced in cache architecture to take in consideration during the development of cache aware algorithms from scratch. These cache characteristics are common in both general purpose and embedded systems.

2.2.1 Cache coherent protocols

Usually in multicore architecture, there is a private L1 cache for each core, and there is a L2 or L3 cache shared among all cores (SMP), or among cores that belong to the same node (NUMA). Shared cache is one of the most critical resources in multicores.

As for all types of shared resources, also for shared cache it's necessary to ensure integrity of the shared data. When two or more CPUs operate on a shared variable and one of these modify the variable, it is necessary that the information regarding this change of value be communicated to the other CPUs. The mean used to communicate these informations is called *coherence protocol*, it defines the rules used to communicate memory update between different cores. Existing coherence protocols are classified based on the mechanism by which they ensure cache coherence:

Snooping based protocols: each cache monitors address lines of shared bus for every memory transaction made by remote processors. Appropriate action is undertaken when data locally cached is modified by this transaction, for example: a write by a remote processor into a data address locally cached results in an invalidation of the local cache copy.

Snarfing based protocols: a cache controller watches both address and data in an attempt to update its own copy of a memory location when a second master modifies a location in main memory. When a write operation is observed to a location that a cache has a copy of, the cache controller updates its own copy of the snarfed memory location with the new data.

Directory-based protocols: shared data are placed in a common directory that maintains the coherence between processor caches. This directory acts as a look-up table for every processor to verify coherence of data that is currently being read or updated.

The first two mechanisms are typical of the SMP architecture, while the last is used in large point-to-point inter-processor communication network architectures. Snooping protocols became popular and widely accepted with multiprocessors systems since it required minimal change to the pre-existing physical shared bus interface to the memory. The inherent broadcasting property of the snoop protocols makes it simple to implement but places an upper limit on scalability. Over the

years, several snooping based cache coherence protocols were developed. The most common is MESI protocol. With this protocol, every cache line is marked with one of the following states:

Modified: the cache line is present only in one of the local caches, and it has been modified from the value in main memory. Write on a modified cache line are allowed, reads are a bit complicated. The local cache that owns a modified cache line must intercept (snoop) all attempted reads (from all of the other caches in the system) at main memory location that correspond to modified cache line, forcing them to back off, then writing data to main memory and change state of cache line to shared state.

Exclusive: the cache line is present only in the current cache, and it matches main memory. It is possible to read/write lines in this state. A cache that owns lines in exclusive state must snoop all read and write transactions from all other cache and if it intercepts some read that regards owned cache line, it changes the state of that line from exclusive to shared.

Shared: indicates that this cache line may be stored in other caches and it matches the main memory. It is possible read cache line in this state. Writes to a shared cache line are allowed but before to perform the operation, it is necessary invalidate all other copies in other caches. A cache that holds a line in the Shared state must listen for invalidation from other caches, and discard the line (by moving it into Invalid state) on a match.

Invalid: indicates that this cache line is invalid, read/write operation on invalid cache line are denied

Cache coherence protocols play an important role to improve efficiency of the read/write in cache. There are a lot of variant of MESI protocol, the most recent variants are MESIF and MOESI.

The former protocol adds the Forward state. This state indicates that only one cache should act as a designated responder for any requests for the given line. With the standard MESI protocol, a request for cache line will receive a response from each cache that contains that line in the shared state. Instead, with MESIF protocol a request will be responded to only by the cache holding the line in the Forward state. A cache line shared among multiple processors will be in Forward state in only one L3 cache. This protocol is used in new Intel microarchitecture Nehalem and

it is designed for NUMA. The aim of this new state is to reduce communications between cache.

The latter protocol add Owned state. A cache line in this state holds the most recent, correct copy of the data. Only one processor can hold the data in the Owned state, all other processors that contains a copy of that data must hold the them in the shared state. A copy of data in main memory can be incorrect. A cache line in owned state may be changed to the Modified state after invalidating all shared copies, or changed to the Shared state by writing the modifications back to main memory, furthermore cache lines must respond to a snoop request with data. It is clear that the aim of this protocol is to avoid the need to write a dirty cache line back to main memory when another processor tries to read it. With the Owned state, processor can supply the modified data directly to the other processor. This is beneficial when the communication latency and bandwidth between two CPUs is significantly better than to main memory. An example of MOESI implementation is present in AMD Shanghai microprocessor.

2.2.2 Inclusive and exclusive cache

Another important architecture detail that affect performance is if a cache is inclusive or exclusive. An inclusive cache means that all data available in higher level caches are contained also in the last level cache, namely in shared cache. An exclusive cache means that data is present only in one cache.

It is clear that these two architectural policies are focused on two different aspects. The first policy greatly reduces snoop traffic because if a core doesn't find requested data in any of its cache level, it knows the data it is also not present in any other core's cache. The second policy, instead, allows to store more data than an inclusive cache, because for each data only one copy is stored.

An example of inclusive L3 shared cache is implemented in Intel Nehalem microarchitecture. Each cache line in the L3 contains additional "core valid" bits, one for each core in the system, denoting which cores may have a copy of that line in their private caches. If a "core valid" bit is set to 0, then that core cannot possibly have a copy of the cache line, while a "core valid" bit set to 1 indicates it is possible (but not guaranteed) that the core in question could have a private copy of the line. Since Nehalem uses the MESIF cache coherency protocol, if a cache line in L3 have more than one "core valid" bits set to 1, the cache line is guaranteed to be clean and it will be in Forward state in L3, in this way, the L3 is the only responder for

all request for that line. This greatly reduces the amount of traditional "snooping" coherency traffic between cores.

An implementation of exclusive cache can be found in AMD's Shanghai processors. These CPUs present an interesting implementation of this type of cache architecture, because L1 and L2 are exclusive cache, but last level shared cache (L3) is a not-inclusive cache. Not-inclusive architecture is a variant of exclusive architecture, because if a cache line is transferred from the L3 cache into the L1 of any core, the line can be removed from the L3. According to AMD this happens if it is "likely" that the line is only used by one core, otherwise a copy can be kept in the L3. About how the CPU can "understand" if a line will be used only by one core, AMD has not revealed details. Nowadays, an inclusive cache is preferred over an exclusive cache, because it simplifies the problem of cache coherence.

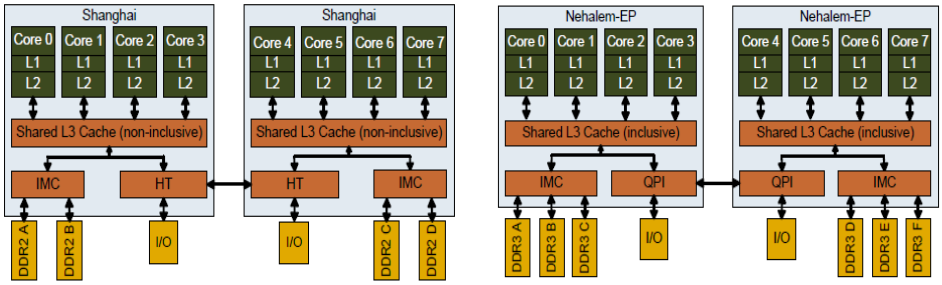


Figure 2.2: AMD Shanghai and Intel Nehalem [4]

To get a sense of how cache coherence impacts on system performance, it is possible to look over the work made by Molka et al. [4]. They have compared performance of MESIF protocol applied to inclusive cache (Intel Xeon 55** Nehalem) and MOESI protocol applied to an exclusive cache (AMD Shanghai), see figure 2.2. These processors tested are multithreaded and NUMA dual-socket SMP systems. In the table 2.3 read latencies recorded during test are showed.

Processor		Shanghai				Nehalem			
Source	State	L1	L2	L3	RAM	L1	L2	L3	RAM
Local	M/E/S	1.1 (3)	5.6 (15)	15.2 (41)	77	1.3 (4)	3.4 (10)	13.0 (38)	65
Core1 (on die)	Modified	44 (119)				28.3 (83)	25.5 (75)	13.0 (38)	
	Exclusive	66 - 77				22.2 (65)			
	Shared					13.0 (38)			
Core4 (other die)	Modified	83 (224)		83 (224)	118	102 - 109			106
	Exclusive	99 - 116				63 (186)			
	Shared					58 (170)			

Figure 2.3: Read latencies [4]

on-chip latencies These data measure access time to the cache of other cores on the same die. We will see that having an inclusive cache or not influence strongly this type of latency.

Nehalem: a *Shared* cache line in L3 is guaranteed to be valid and it could be in shared or forward state, it can be accessed within 13 ns. An *Exclusive* cache line that has core valid bit set in L3, may has been modified in higher level cache, this fact forces the L3 cache to check the coherency state in the core, therefore the latency increase up to 22.2 ns, furthermore, in the this type of cache, exclusive cache lines can be silently evicted from higher level cache and remain only in L3. The check on coherency state is made also for these lines that are only in L3 cache. A read of *Modified* cache line present in other on-chip L1/L2 requires 25.5/28.3 ns, but if a modified line is evicted from L1/L2, it is required a write-back in L3 and an update of the core valid bits. It means that future access at that line will has a latency of 13 ns again.

Shanghai: a *Shared or Exclusive* cache lines from higher level caches need to be fetched from the cores or main memory if no copy exists in the non-inclusive L3 cache. Latency showed in the table indicate that request for lines in these state are serviced by main memory. A *Modified* state it's the only state that allow to avoid access to main memory.

off-chip latencies These data measure access time to the cache of other cores on another die. These latencies include additional penalty for the QPI/HT data transfer. We will see that cache coherence protocol influence this type of latencies.

Nehalem: thanks to inclusive cache, the access to unmodified data is fast. Latency for *Exclusive* cache line include a snoop of one core (63 ns), *Shared* cache line don't require snoop (58 ns). Moreover, latency for *Modified* lines is higher ($> 100ns$), because the MESIF protocol requires a write back to the main memory.

Shanghai: cache lines in *Shared* state are fetched from main memory ($> 99ns$), L3 can satisfies request for *Exclusive* cache lines (83 ns). Cache line in *Modified* state can be forwarded to the requesting core from all cache levels, furthermore, thanks to MOESI protocol, a modified cache line in the remote cache can switch to the Owned state and avoid to be written back to main memory.

It is interesting to note how accesses to remote caches on Shanghai are slower

than accesses to the local RAM (83 vs. 77 ns).

2.2.3 Cache Hardware prefetcher

Finally, it is necessary to spend a few words about prefetching. Prefetcher is an hardware component that tries to predict which memory addresses are going to be used by the program, in order to load needed data in cache memory. Typical technical workloads often access memory in regular and sequential patterns, therefore, with a smart prediction mechanism, a prefetcher can select and preload correct data and, in this way, reduce memory latency. The key point to build a good prefetcher is to design prediction mechanism. In literature there are many ideas to solve this problem. An example of concrete solution is the Intel Smart Memory access. This system was introduced with Intel Core microarchitecture. In this system there are two prefetchers to the Level 1 data cache and the traditional prefetcher to the Level 1 instruction cache. In addition there are two prefetchers associated with the Level 2 cache. In total, there are eight prefetchers per dual-core processor. In order to improve the accuracy of the prediction, the prefetcher system tags the history of each load using the Instruction Pointer (IP) of the load. For each load, the prefetcher builds a history and keeps it in a suitable history array. Based on load history, the prefetcher tries to predict the address of next load accordingly to a constant stride calculation (a fixed distance or ”stride” between subsequent accesses to the same memory area). At this point, the prefetcher generates a prefetch request with the predicted address and brings the resulting data to the Level 1 data cache. In literature, this kind of prefetchers are called *strided* prefetcher. Other architectures, such as Power ISA_2.06, that use a strided prefetcher, introduces cache instructions to hint prefetch system for data prefetching. With this instructions an application can specify direction, depth, no of units and so on. In this way, the programmer has a low level control on data prefetched.

Another category of prefetcher are *non-strided* data prefetcher. They are very useful for accessing complex and irregular data structures as linked list, B-Trees etc. There are different techniques to implement these prefetcher, one of these is ”pattern history based prefetcher”. In this approach, the prefetcher tracks the addresses of misses and tries to identify specific patterns of misses that occur together. Once a pattern of misses has been detected, the prefetcher will find the first miss in the pattern. When this first miss occurs again, the prefetcher will immediately prefetch the rest of the pattern. For traversing a complex data structure like a linked list,

this would be a fairly effective approach. Recently AMD has announced that it will employs a non-strided prefetcher in its brand new Bulldozer microarchitecture, but it has not revealed details on implementations.

2.3 Classification of cache-aware Scheduling algorithms

Cache aware scheduling policies can be classified according to type of strategy followed. They are divided in *data locality* and *temporal locality* policies. The former type is focused on a smart allocation of resources in cache. These policies partition cache memory in order to every task may use a dedicated area in cache memory and then, reduce inter-thread cache interference and miss rate. The latter type is focused on cache reusability. These policies schedule tasks that subsequently will access at the same data, reducing miss rate.

2.3.1 Data locality policies

These policies are focused on the use of Last Level Cache (LLC) made by scheduled tasks. They can be used both for Real-time and Fair tasks. They don't take care about core-local cache (usually L1) or other shared resources like interconnects. In literature, data locality is the most developed type of cache aware policies, because they can be integrated in every OS and are relatively simple policies to develop. Now I will show two examples of data locality policies and how they are implemented.

Cache aware Fixed priority policy: to understand this type of policy assume this situation: consider a multicore platform consisting of M cores with an on-chip LLC, and a task set τ, in which each task T releases a thread J_i every period $p(T)$. Every released thread is characterized by a worst-case execution time (WCET) denoted by $C(T)$. It means that each thread released by task T has a maximum duration of $C(T)$. For each thread is defined the number $A(J_k)$ that represents the total cache space size used by it: we define this space as per-thread *working set sizes* (WSS). For simplicity we assume that each thread generated by a task T use the same number of partition, therefore it is possible to express $A(J_k)$ as $A(T)$. The quantity $e(T)/p(T)$ is called the utilization of T, denoted $u(T)$. The deadline $d(J_k)$ of a thread J_k coincides with the release time of thread J_{k+1}. If thread J_k completes its execution after time $d(J_k)$, then it is **tardy**. For some scheduling algorithms, tardiness may be bounded by

some amount B, meaning that any thread J_k will complete execution no later than time $d(J_k) + B$. Finally, assume that tasks have fixed priority. This model describes very well a typical Real-time application.

A thread J_k is scheduled for execution if:

1. J_k is the thread of highest priority among all waiting threads,
2. There is at least one core idle
3. There is enough cache space, i.e. at least $A(J_k)$, is available.

	$p(T)$	$C(T)$	$A(T)$
T_1	3	2	1
T_2	4	3	2
T_3	5	3	2
T_4	8	3	1

Table 2.1: An example task set

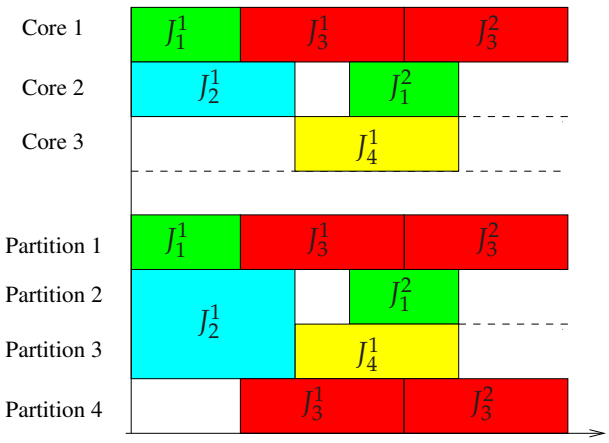


Figure 2.4: Example of scheduling performed by Cache aware Fixed priority policy

Figure 2.4 shows how the task set in Table 2.1 is scheduled by the policy. The index of each task identifies the task and indicates the priority of the latter. Higher priority is 1 and lower priority is 4. In the picture threads are represented in this way: J_α^β, where α represents which task has released the thread, and then also its priority, and β is an index that identifies the thread. We assume that co-scheduled threads use not-overlapped cache partitions.

At time 0 all threads are released. At this time, the thread J_4^1 can not be executed because it has lower priority than J_3^1 and the latter can not be scheduled since there is not enough idle cache partitions available.

It is clear that the aim of the described policy is to avoid cache trashing that occurs when the amount of space required by co-scheduled threads is greater than the dimension of LLC. A very important assumption made in this policy to achieve this goal is that all threads use not-overlapped partition of cache. This fact simplifies the scheduling problem, because it's enough to make a check on the amount of cache space occupied by co-scheduled threads to infer if cache thrashing occurs. Nevertheless, in practice, this is not always true, because it greatly depends on type of application executed.

Mechanisms and examples They key mechanism to reach this objective is profiler. At runtime, the profiler tries to infer what is the WSS of each scheduled task and provide these informations to scheduler. The challenge is how to do this thread. Calandrino et. al [2] propose an interesting implementation of co-scheduler and for this type of policy.

The implementation is focused on Soft Real-time tasks, its solution is quite simple: it makes EDF cache-aware. Standard EDF algorithm gives higher priority to the task with earlier deadline in order to schedule it as soon as possibile. A cache-aware EDF is very similar to classic EDF: it ”promotes”, that is increases priority, of the thread with the smallest WSS, in this way threads on a runqueue are ordered by WSS.

The algortihm uses two separate run queues for eligible threads: the former is EDF-ordered, the latter is a ”promotion” queue. In this queue, threads are ordered from smallest to largest WSS. The thread in front of this queue is ”promoted”, that is its priority was increased and, for this reason, it remains at the front of the queue and it will be the next task to execute. If there are tardy threads, they are inserted in EDF-ordered queue and they are scheduled according to EDF policy. The EDF-ordered queue contains task that have higher priority than tasks inserted in promotion queue. When the heuristic picks a task from promotion queue, it checks if it will cause cache thrashing. For this reason, the scheduler maintains a variable with the total amount of the WSSs allocated, that is the cache space allocated. If this value plus the WSS of the thread to schedule exceeds the cache space size, thrashing will

occur. In this case, the core remains idle and the thread waits for some thread that free cache resources. Obviously task priorities are respected, it means that a task with the smallest WSS is promoted only if it has the higher priority.

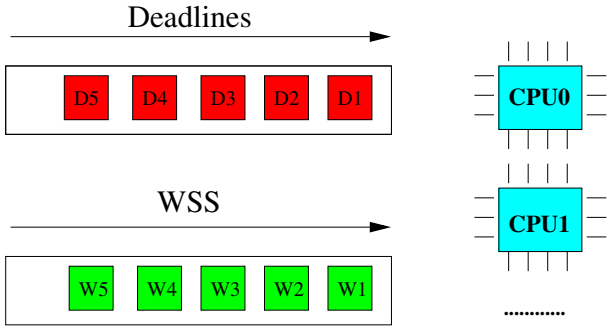


Figure 2.5: Queues used in the algorithm: tasks in "EDF-ordered" queue are ordered by their deadline, tasks in "promotion" queue are ordered by the size of their WSS. Tasks in EDF-ordered queue have priority greater than priority of the tasks in promotion queue.

To compute the WSS used by each task, the co-scheduler divides the total cache misses observed over all profiled threads by the total number of profiled threads, and multiplying the result by the cache line size. Cache misses and WSS are related, because, from experiments executed by Calandrino et al, follows that number of cache misses multiplied for dimension of a cache line gives a number proportional to WSS used by a thread. All the profiled threads belong to the same task. The co-scheduler discards measures obtained by threads that cause cache thrashing. Cache misses are recorded using performance counters. At the beginning of an application, there is not any measure about cache miss, therefore the profiler requires a bootstrapping phase in order to get necessary measures. This phase converge pretty fast to a reasonable result.

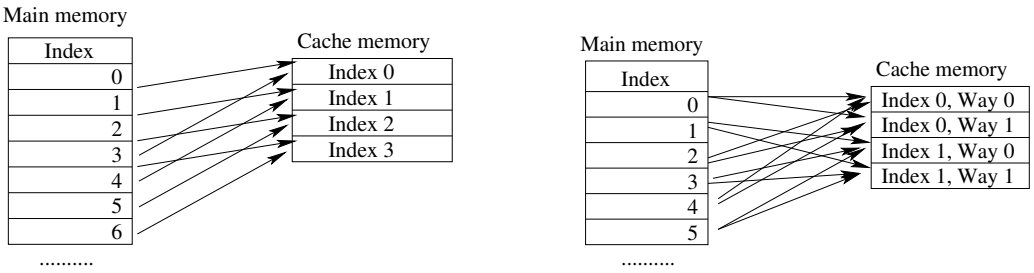
It is interesting to note, that in this implementation the only mean to avoid cache trashing is a check on amount of WSSs used by scheduled tasks. This is a sub-optimal solution because cache trashing can occurs also in those cases where the total amount of WSSs is less than total amount of cache space. To understand why, it is necessary to remind some consideration about cache architectures. When a cache miss occurs, it is necessary to make room in cache for the new entry to load. The heuristic that decide which is the entry to evict is called the replacement policy. According to replacement policy,

cache memory are classified in:

Direct-mapped: a line in main memory can be placed only in one cache line. There is an 1:1 relationship between main memory and cache memory.

Fully associative: a line in main memory can be placed in any cache line. There is a 1:N relationship between main memory and cache memory.

N-way set associative: it is a trade-off between direct-mapped and fully associative. Cache memory is divided in set, each set can contain N cache line. A line in main memory can be present only in one specific set and within it, that line can be placed in any line that belong to that set.



(a) Cache Direct mapped

(b) Cache N-Way set associative

Figure 2.6: Example of replacement policy in Direct-mapped cache and in N-Way Set associative Cache

Considering this architecture detail, it is clear that a check on WSS is sufficient to prevent cache thrashing only if the cache is fully associative, because every line in main memory can be mapped on any cache line but, in a N-way set associative cache, a line in main memory can be mapped only within a certain set. For this reason, if a task occupies cache line used by a co-scheduled task, cache trashing will occurs. Furthermore, there is another important detail: we have assumed non-overlapping WSS, but, in practice, this hypothesis is unlikely, therefore cache thrashing can occur also with a fully associative cache. Nevertheless, Calandrino et al. test their algorithm with Soft Real-Time multimedia application and, according to results showed [2], it seems that this sub-optimal solution is quite effective, at least with this kind of applications.

Advantages and drawbacks This policy presents a clear drawback: it may waste resources. According to rules previously showed, if there is an idle core, but not enough cache space, a task may not be scheduled and, in this way, degrade throughput.

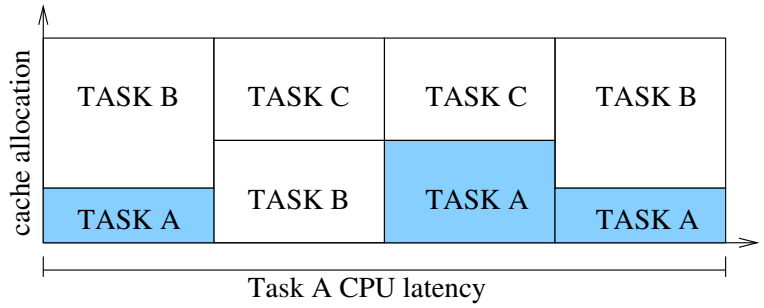
Nevertheless, the policy improves predictability of the application, because, theoretically, each thread should find in cache enough space for its resources, maintaining constant its execution time. This aspect make this policy very indicate for Real-time system, where predictability is a very important aspect.

There is another important advantage related to improvement of predictability. On single-core systems there are well-developed techniques for timing analysis of embedded software. Using these techniques, the WCET of real-time tasks may be estimated, and then used for schedulability analysis. In multicore systems these techniques can't be used, because, as we have seen, cache behaviour (hit or miss) is unpredictable because it depends on which tasks are co-scheduled, therefore WCET can be variable and very difficult to estimate. With the analyzed policy, it is possible perform a proper cache isolation, that make cache behaviour more predictable and then it becomes feasible to execute a system-level schedulability analysis using existing WCET analysis techniques. For details on WCET analysis for Real-time system in multicore platform see [7]

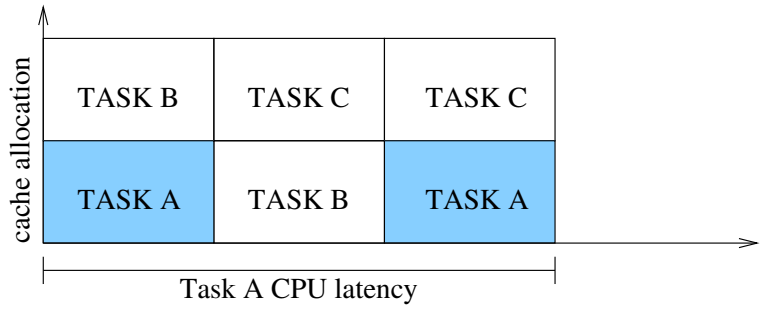
Fair cache sharing policy: this policy is focused on Fair tasks, therefore it considers an application model with less constraints than to that considered by Calandrino et al. This implementation ensures *fair cache sharing*, that is a strategy throughput oriented.

As we have previously seen, a bad combination of coscheduled tasks may cause cache thrashing and, consequently, a variation on task's instruction per cycle (IPC) that determines task's performance variation. But, if co-scheduled tasks sharing cache in a fair way, they can allocate needed resources in cache and they experience the fair IPC, that is the IPC that ensure the best overall performance. The mechanism used to ensure the fair IPC is to modify CPU timeslice assigned to a task, in this way, each task always run as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated.

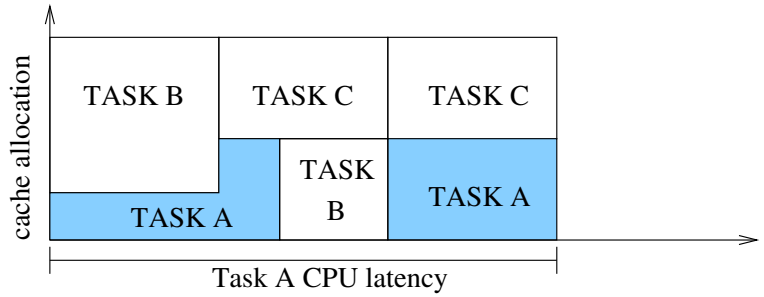
In figure 2.7 is represented what the algorithm tries to do. Each box corresponds to a task. The height of the box indicates the amount of cache allocated to the thread. The width of the box indicates the thread's CPU timeslice. The area of the box is proportional to the amount of work completed by the thread. Stacked thread boxes indicate corunners. In the case (a) is represented



(a) Real case: tasks don't share equally cache



(b) Ideal case: tasks share equally cache



(c) Modified time slice to ensure equally sharing of cache

Figure 2.7: Example of how Fair cache sharing policy modifies timeslice to ensure fair cache sharing [3]

a real case where tasks don't share equally cache and, for this reason, Task A needs more CPU time to execute all its job. In situation (b) is represented the ideal case, where all tasks share equally cache. Case (c) represents how is possible to ensure fair cache sharing: Task A can't make forward progress, so CPU timeslice of task A is increased and, at the same time it decreases timeslice of task B to maintain balanced CPU sharing. This is what the algorithm does.

Mechanism and examples: An implementation of this policy is developed by Fedorova et al. [3]. The algorithm is divided in two phases: sampling and scheduling. During the sampling phase, the co-scheduler gathers performance data and uses it to estimate the task's fair miss rate. During the scheduling phase, the scheduler periodically monitors the task's performance and adjusts the task's CPU timeslice if its actual performance deviates from its fair performance.

The fair miss rate and fair IPC of a task are estimated via performance counters, executing this experiment: different groups of corunners are formed, the task which we want to estimate fair miss rate is executed together to each group. At the end of execution, miss rates of each executed task are recored. These measures correspond to one data sample. It is necessary to collect at least ten data sample for each task, in order that each task has completed at least 100 million of instructions, to eliminate effects of compulsory misses. At the end of the sampling phase, the co-scheduler estimates the fair miss rate using a linear regression analysis and, according these informations, also fair IPC is computed. In the scheduling phase, the scheduler monitors the task's IPC, again via performance counters. According to the difference between IPC measured and task's fair IPC, the scheduler adjusts task's CPU timeslice.

In Figure 2.8 is reassumed how fair miss rate is estimated. Fedorova et al. found that that a linear function is the best approximation of relationship between co-runnners miss rates.

The overhead of sampling phase is fixed and it is repeated every time a thread has completed one billion instructions, while the check on a task's IPC is made every 50 million instructions executed. When to check task's IPC and when to perform sampling phase is decided according to empirical observations made by the developers of the algorithms.

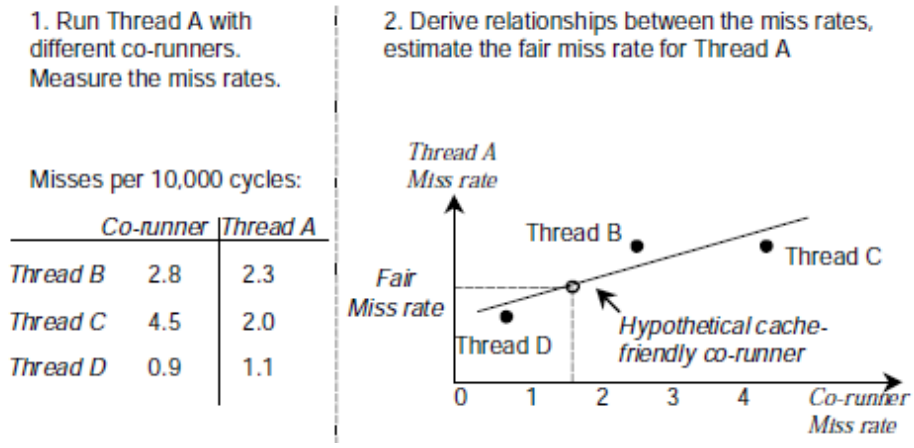


Figure 2.8: step of co-scheduler [3]

It's important to note that this algorithm does not establish a new CPU sharing policy but simply enforce existing policies. For example, if the system is using a fair-share policy, the cache-fair algorithm will make the cache-fair threads run as quickly as they would if the cache were shared equally, given the number of CPU cycles they are entitled to under the fair-share policy. Furthermore, the algorithm doesn't add any additional data structure, but simply requires only to record fair miss rate and fair IPC for each task.

Advantages and drawbacks Unlike previous example, this algorithm don't waste resource, moreover it use a more complex co-scheduler than that implemented by Calandrino et al, because it has to estimate fair miss rate of each task. According some statistical considerations, two corunner task experience fair cache sharing, if their miss rate is similar and that miss rate is the fair miss rate. To determine this value for each task, a possible approach consist of executing a task with all available corunner, but this approach is not feasible. The solution adopted by the co-scheduler, instead, consist of to run each task with a small number of possible corunner and use miss rates measured to derive a relationship between the miss rates of the task analyzed and its co-runners, and use that relationship to estimate task's fair miss rate. This procedure could introduce an higher unpredictability in the application and for this reason, that this algorithm couldn't be suitable for Real-Time systems, where predictability is required.

2.3.2 Temporal locality policies

The idea behind this type of policies is very simple, but in practice it is more complex than the idea behind data locality policies. These policies require complex infrastructure and nowadays only few research activities have developed policies of this type. An interesting implementation of this type of policy can be found in COOL project [8]. Recently Yang et al. [6] have implemented a scheduling algorithm inspired to these policies and naturally the concept of task-affinity is a simplified implementation of temporal locality policies.

cache reusability policy The key point to maximize reusability of data in private cache, that is in L1 cache, is data sharing among tasks. If two threads, or in general two task, sharing common data, they are scheduled subsequently on the same CPU, in this way, it's more likely to reuse the same data in private cache. According this observation, tasks are inserted in abstract list called *sharing groups*. Tasks that share the same resources are put in the same sharing group. The scheduler assigns the same CPU at all tasks that are in the same sharing group, in this way they will use always the same private cache.

Instead, to maximize reusability of data in shared cache, that is L2 or L3 cache, it is necessary to improve the opportunities that the subsequently scheduled tasks could reuse the data accessed by the current tasks at the scheduling point.

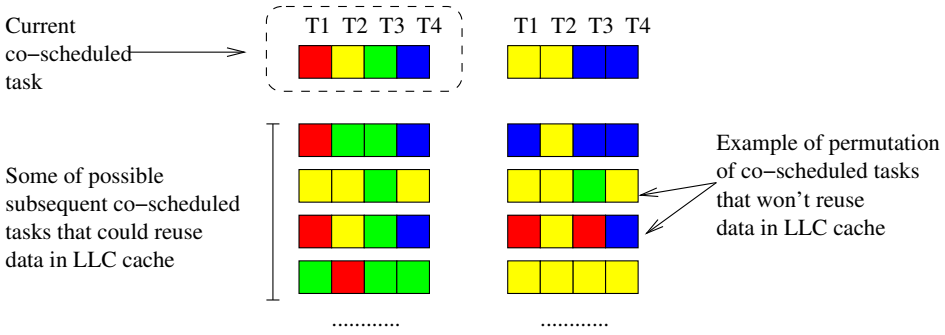


Figure 2.9: Data accessed by possible permutations of co-scheduled tasks. In the right figure co-scheduled tasks with green or red squares won't reuse data, because current co-scheduled tasks don't access to red or green squares

In figure 2.9 each square represents the data set used by a scheduled task, squares with different colours represent different data set. In case (a), four tasks are co-scheduled and they access to four different data sets. In the case

(b) four co-scheduled tasks access to only two different data sets. To reuse the data in the cache, it is necessary schedule tasks that use data sets that have been accessed by current co-scheduled tasks subsequently. Therefore, more data sets are accessed by current co-scheduled tasks and there will be more opportunities that the subsequent coscheduled tasks could reuse the data in the shared cache. For example: in situation (a) there are 4^4 permutations of co-scheduled tasks that could reuse the data, but only 2^4 in situation (b). According this observation, the scheduler should schedule the tasks using as many data sets as possible, but at one important condition: the total amount of data set used by co-scheduled task, must be less than the total cache space, in order to avoid cache trashing.

The interesting thing, is that to maximize reuse of data present in private cache and to maximize reuse of data present in shared cache are not in conflict with each other. The explanation is straightforward: each core executes a task that belongs to different sharing group, therefore co-scheduled tasks would access **different data sets**, in this way, the number of data sets accessed in shared cache is maximized.

According to this theory, it is possible to list the rules applied this kind of policy:

- ❏ Co-schedule tasks that belong to different sharing groups.
- ❏ Schedule a task only if it doesn't cause cache thrashing.

Mechanism and examples: An implementation of this policy is developed by Yang et al. [6].

This implementation use an array of linked list. In each list are inserted tasks that sharing the same resources, for simplicity we call the $i-th$ list SG_i. Each list is characterized by total working set used by task inserted in, it is denoted with TW_i. It is clear that these list are the implementation of the sharing groups previously described, for this reason, from now I will call these lists sharing groups. Before to start, the algorithm assigns each sharing group to one CPU: if number of core (NC) is less than number of sharing groups (NS), a CPU will execute more than one sharing groups, see fig. 2.10.

The heuristic is divided in two phases. In the first step, the algorithm chooses a task from each sharing group for a total of NC tasks, in this way, it has

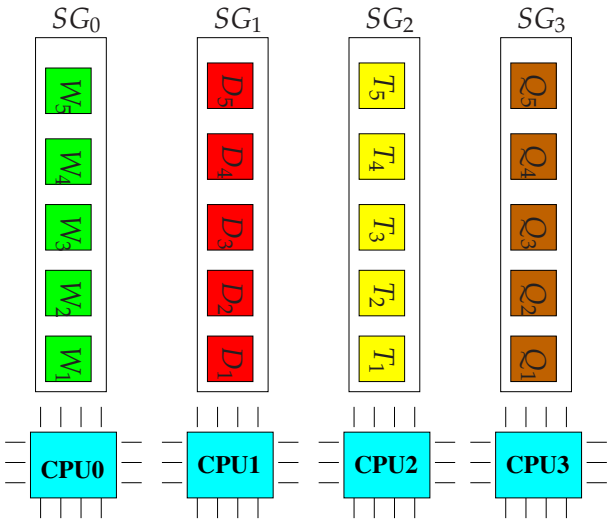


Figure 2.10: Example of how sharing groups can be assigned to different CPUs. A possible coschedule is: $PS_j = \{Q_1; T_1; W_1 D_1\}$

defined a possible co-schedule (PS_j). It computes the sum of working set size (SWSS) of the PS_j selected, if SWSS is greater than dimension of shared cache, PS_j is discarded. This procedure is repeated for all possible PS_j. At the end, all filtered PS_j that fit into the shared cache are put in a queue called C and waiting for the next phase.

Before to analyze the next step, define the *sharing number (SC_j)* as the number of different sharing groups belong to a possible filtered co-schedule PS_j. In the next step, elements present in queue C are ordered according their SC_j. In order to schedule tasks that access the most different data sets, the algorithm select tasks that belong to PS_j with the maximum SC_j.

If C is empty, it means that there isn't any possible co-schedule PS_j with the total working set size TW_j that fit in shared cache. Also in this case, the algorithm would aim to minimize the shared cache misses by choosing the possible co-schedule with the maximum SC_j and with smallest TW_j.

An interesting aspect of this implementation, is that it is possible to reduce also cache coherence overhead, the reason is very simple. Tasks that belong to two different sharing groups don't share data, otherwise they should be in the same group. It means that lines used in private cache by a co-scheduled tasks that are in different groups, won't never be in shared, but only in modified or exclusive state. in this way, the overhead due to management of cache coherent shared state is reduced.

Yang et al. implemented their algorithm in the scheduler present in Threading Building Blocks (TBB) that is a multithreading library developed by Intel. This library allows to programmer to specify which is the working set of each threads and this information is just what it needs to implement their policy.

Advantages and drawbacks This policy is an improvement of data locality policies, because, also in this case, it is essential to prevent cache trashing in LLC, exploiting the same mechanisms used in data locality policies.

An important aspect of this policy is that it deals with cache coherence, that, as we have seen in previous section, is a very important factor that influence system performance. A drawback, is that to implement this policy is required an infrastructure that provide the same informations used in data locality policies, that is how much cache space a task use, and in addition, which are addresses used by task, that is a very difficult information to obtain. To resolve this problem, the algorithm involves the programmer providing him a suitable API to influence the scheduling choices. This fact means to modify application code with additional functions.

The difference between temporal locality policies data locality policies is very slight. The temporal locality policies, try to reduce cache misses choosing which tasks will be scheduled in two different moments. In other words, this type of policy, for each cpu, choose which task that will be scheduled after the current task in order that the next task will reuse resources already allocated in cache of each cpu. Instead data locality policies, try to reduce cache misses choosing tasks that cause the lowest inter task cache interference. In other words, this type of policy make sure that all scheduled task in the same moment may allocate necessary resources in shared cache.

Chapter 3

Improving TaskAffinity

In this chapter, we analize the behaviors of the current task-affinity implementation considering different architectures and benchmarks. This analysis is two fold: on one side it allows to understand why the current task-affinity implementation is not effective on some architectures, e.g. the Intel Xeon, on the other which are the points of task-affinity logic that can be optimized in order to improve an application *throughput and predictability*.

3.1 Scheduler architecture on 2.6.34

Now I will briefly introduce which are the parts of the scheduling procedure interested by the task-affinity logic and which are the most important changes carried out from 2.6.31 kernel version to 2.6.34.

3.1.1 Task wake up management

The scheduling procedure for a task starts when it wakes up. A task can wake up for different reasons, i.e. a semaphore becomes unlocked or a new task creation. In all those cases different kernel functions are called, but at the end they call the `try_to_wake_up` function:

```
2381 static int try_to_wake_up(struct task_struct *p, unsigned int state,
2382             int wake_flags)
```

where p is the to be waken task. This function follows these steps:

1. Disables kernel preemption, locks the runqueue where p was last executed and check if p is not already waken up and it is not already on a runqueue.

In the first case the function releases lock and exits, in the second case the function check if a *push* is necessary. Further details about these if statements will be describe thereafter. If the two checks fail, *p*'s state is changed in the TASK_WAKING, the lock on runqueue is released and `select_task_rq`, a wrapper for a class-specific `select_task_rq_rt`, is called. The function `task_waking` at line 2420 is class-specific and regards only Fair tasks.

```
2391         this_cpu = get_cpu(); /* disable kernel preemption */
2392
2393         smp_wmb();
2394         rq = task_rq_lock(p, &flags);
2395         update_rq_clock(rq);
2396         if (!(p->state & state))
2397                 goto out;
2398
2399         if (p->se.on_rq)
2400                 goto out_running;
2401
2402         cpu = task_cpu(p);
2403         orig_cpu = cpu;
2404
2405 #ifdef CONFIG_SMP
2406         if (unlikely(task_running(rq, p)))
2407                 goto out_activate;
2408
2415         if (task_contributes_to_load(p))
2416                 rq->nr_uninterruptible--;
2417         p->state = TASK_WAKING;
2418
2419         if (p->sched_class->task_waking)
2420                 p->sched_class->task_waking(rq, p);
2421
2422         __task_rq_unlock(rq);
2423
2424         cpu = select_task_rq(p, SD_BALANCE_WAKE, wake_flags);
```

Figure 3.1: A portion of the `try_to_wake_up` method

2. `select_task_rq_rt` choose on which CPU *p* will be executed. It calls `find_lowest_rq` that returns the best CPU where to put *p*. Criteria used to choose the best CPU for *p* will be described soon. When `select_task_rq_rt` returns, check

for cpuaffinity [1] and if selected CPU is online, in that case returns, otherwise calls `select_fallback_rq` that returns an any online CPU that "respects" p's cpuaffinity.

```
2344 static inline
2345 int select_task_rq(struct task_struct *p, int sd_flags,
2346                                            int wake_flags)
2347 {
2348         int cpu = p->sched_class->select_task_rq(p, sd_flags,
2349                                            wake_flags);
2350
                .....
2359         if (unlikely(!cpumask_test_cpu(cpu, &p->cpus_allowed) ||
2360                      !cpu_online(cpu)))
2361                 cpu = select_fallback_rq(task_cpu(p), p);
2362
2363         return cpu;
2364 }
```

Figure 3.2: A portion of the `select_task_rq` method

3. acquires the lock on selected runqueue, updates some p's statistics, enqueues p on selected runqueue and calls `check_preempt_rq_rt`

4. checks if p has priority greater than priority of the task currently executed on the selected runqueue, in that case it calls the `need_resched` function in order to perform the context-switch on the selected runqueue at the end of `try_to_wake_up`.

5. updates p's state to TASK_RUNNING and calls the class-specific function `task_woken` to check if p must be pushed from the selected runqueue. `task_woken` has effects only for Real-time tasks.

The most important differences from version 2.6.31 related to Real-time tasks regard principally `try_to_wake_up`.

It is possible to have multiple istances of `try_to_wake_up` for the same task executed simultaneously. In the 2.6.31 kernel version, this problem is resolved by

[1] On Linux it is possible decide on which CPU a task can be executed. The set of CPUs that can execute a task is called cpuaffinity of that task. Each task owns a mask called *cpus_allowed* that include all CPUs where it can be executed, that is its cpuaffinity

```
2434          rq = cpu_rq(cpu);
2435          raw_spin_lock(&rq->lock);
2436          update_rq_clock(rq);
2437
                  ....
2447  #ifdef CONFIG_SCHEDSTATS
2448          schedstat_inc(rq, ttwu_count);
2449          if (cpu == this_cpu)
2450                  schedstat_inc(rq, ttwu_local);
                  ....
2460  #endif /* CONFIG_SCHEDSTATS */
2473          activate_task(rq, p, 1); /* enqueue task */
2474          success = 1;
                  ....
2491
2492  out_running:
2493          trace_sched_wakeup(rq, p, success);
2494          check_preempt_curr(rq, p, wake_flags);
```

Figure 3.3: A portion of the `try_to_wake_up` method

holding the runqueue lock. In the 2.6.34 kernel version, to deal with this issue a new task's state named TASK_WAKING was introduced.

TASK_WAKING is used to indicate someone is already waking up the task, in this way other instances of `try_to_wake_up` fail when executing the if statement at line 2396 of the `try_to_wake_up`, Fig. 3.1, because the input parameter *state* of `try_to_wake_up` is in the most cases equal to TASK_ALL and then, according to Fig. 3.6, `TASK_WAKING & TASK_ALL` returns 0 and `try_to_wake_up` exits. With this solution it is possible to reduce the time in which the lock of runqueue is held.

3.1.2 Migration policy

Another important part of scheduling procedure is the migration policy. Migration of Real-time tasks is made in two ways:

Push tasks: The push operation is implemented by `push_rt_task()`. The function receives in input a runqueue and looks at the highest-priority non-running runnable real-time task on the input runqueue and considers all the runqueues to find a CPU where it can run. It searches for a runqueue that is of lower priority, that is, one where the currently running task can be preempted

```
1173 static void check_preempt_curr_rt(struct rq *rq,
1174                     struct task_struct *p, int flags)
1175 {
1176         if (p->prio < rq->curr->prio) {
1177                 resched_task(rq->curr);
1178                 return;
1179         }
                .....
1196 }
```

Figure 3.4: A portion of the `check_preempt_curr` method

```
2496         p->state = TASK_RUNNING;
2497 #ifdef CONFIG_SMP
2498         if (p->sched_class->task_woken)
2499                 p->sched_class->task_woken(rq, p);
2500
                ....
2511 #endif
2512 out:
2513         task_rq_unlock(rq, &flags);
2514         put_cpu(); /* enable kernel preemption */
2515
2516         return success;
2517 }
```

Figure 3.5: A portion of the `try_to_wake_up` method

by the task that is being pushed.

The research and the choice of the best CPU for the task to push is executed by `find_lowest_rq` the same function used in `select_task_rq_rt`. This function builds a mask of cpus that contains the lowest-priority runqueues and returns the CPU on which the task to push has last executed, as it is likely to be cache-hot in that location. If this is not possible, the `sched_domain` map is considered to find a CPU that is logically closest to last CPU that has executed the task to push. If this fails too, a CPU is selected randomly from the mask.

The push operation is performed until a real-time task fails to be migrated or there are no more tasks to be pushed. Because the algorithm always se-

```
183 #define TASK_RUNNING            0
184 #define TASK_INTERRUPTIBLE      1
185 #define TASK_UNINTERRUPTIBLE    2
186 #define __TASK_STOPPED          4
187 #define __TASK_TRACED           8
188 /* in tsk->exit_state */
189 #define EXIT_ZOMBIE             16
190 #define EXIT_DEAD               32
191 /* in tsk->state again */
192 #define TASK_DEAD               64
193 #define TASK_WAKEKILL           128
194 #define TASK_WAKING             256
195 #define TASK_STATE_MAX          512
```

Figure 3.6: Task's states

lects the highest non-running task for pushing, the assumption is that, if it cannot migrate it, then most likely the lower real-time tasks cannot be migrated either and the search is aborted. No lock is taken when scanning for the lowest-priority runqueue. When the target runqueue is found, only the lock of that runqueue is taken, after which a check is made to verify whether it is still a candidate to which to push the task (as the target runqueue might have been modified by a parallel scheduling operation on another CPU). If not, the search is repeated for a maximum of three trials, after which it is aborted.

In order to decide which tasks must be pushed, a linked list named *pushable_list* is added to each runqueue. `push_rt_task()` selects tasks to push from this list. A task is inserted in this list when it is enqueued on a runqueue as show in the snippet below.

The current task of any runqueue can't never be in a pushable list, in fact, during a context switch the next task to be executed is removed from the runqueue's pushable list.

Pull task: The pull operation is implemented by `pull_rt_task()`. The algorithm looks at all the overloaded runqueues in the system and checks whether they have a Real-time task that can run on the current runqueue (that is, checks if the target CPU "respects" the cpuaffinity of the task to pull) and if that Real-time task is of a priority higher than the task the target runqueue is about to

```
902 static void
903 enqueue_task_rt(struct rq *rq, struct task_struct *p,
904                                         int wakeup, bool head)
904 {
                ....
912         if (!task_current(rq, p) && p->rt.nr_cpus_allowed > 1)
913                 enqueue_pushable_task(rq, p);
914 }
```

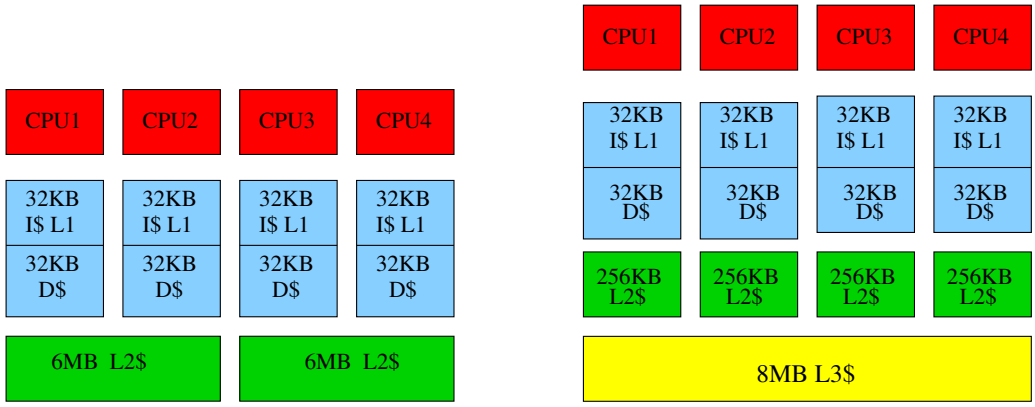
Figure 3.7: A portion of the `enqueue_task_rt` method

schedule. If so, the task is queued on the current runqueue. This search aborts only after scanning all the overloaded runqueues in the system.

In the 2.6.34 kernel version, the migration logic and all data structures involved are not changed with respect to 2.6.31 version.

3.2 Test computers and benchmarks

In this section I will briefly describe machines and the benchmark used to test task-affinity.



(a) Machine A: Intel Xeon E5440

(b) Machine B: Intel i7 870

Figure 3.8: Cache configurations on computers used in this work. Those with D$ are data cache, those with I$ are instruction caches and the others are unified, that is, cache serves to data and instructions.

Machine A The first machine is an Intel Xeon E5440 running at 2.83GHz. There is not any cache shared among all cores. The LLC consists in two big L2-caches,

of 6MB each, shared between sets of 2 cores cache hierarchy is shown in Fig. 3.8(a). On this machine there are two dies: CPU0 and CPU1 are on the same die, while other CPUs are on other die.

Machine B The second machine is an Intel Core i7 870 processor. It runs at 2.93 GHz and has the cache configuration as illustrated in Fig. 3.8(b). The LLC consists in one L3 of 8MB, which is shared by all cores. The L2-caches are private to each processor. All CPUs are on the same die.

The Benchmark used is the same as used in master thesis [1], in Fig.3.9 is represented its structure.

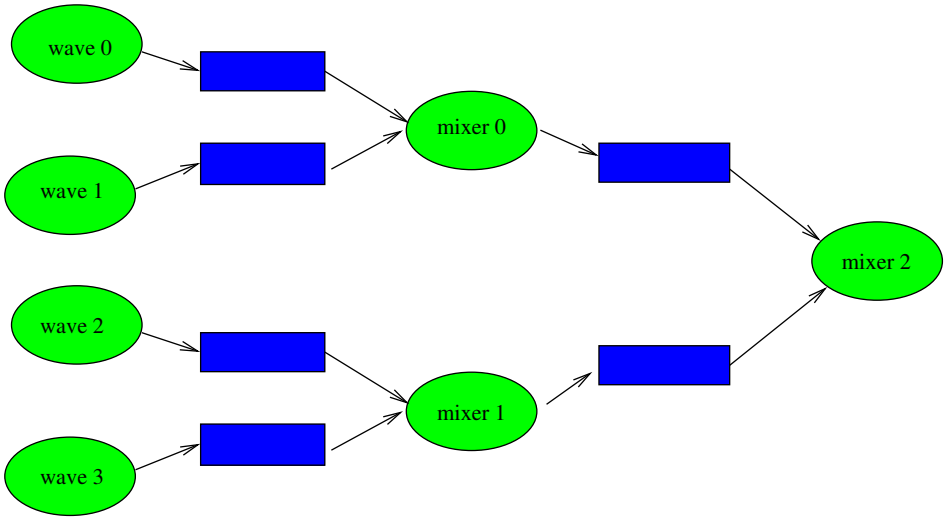


Figure 3.9: Structure of benchmark used: task are green coloured, buffer are blue coloured

The Execution of benchmark is divided in three steps:

1. Waves write their buffer

2. Mixers read data from two buffers filled by waves, they mix read data and write them on their buffer, for example *mixer0* reads data from the buffer filled by *wave0* and from the buffer filled by wave1, mixes the data and then writes its buffer.

3. The Last mixer, reads data from the buffers written by *mixer0* and *mixer1*, mixes the data and writes its buffer.

When *mixer2* has finished to write data on its buffer, we say that a *sample* was produced. The execution time to produce a sample depends on the buffers dimension, because each task has to fill its buffer. Note that waves fill their buffers with integers of 2 byte, therefore if buffer is of 4KB they will write 2048 integers in their buffer. Buffer dimension is always a power of 2. The metric used to evaluate benchmark performance is the same as used in [1]:

$$average + 2 * standarddeviation(A2S), \tag{3.1}$$

where *average* is the average of execution time to produce a sample and *standard deviation* is the standard deviation of execution time to produce a sample. With this metric is possible measure throughput and predictability of the application.

3.3 Analysis of Taskaffinity behaviour

This section analyzes which are the improvements and downsides that the current version of task-affinity shows on different test computers and various buffer dimensions.

In the following experiments only the Intel Xeon and Intel i7 architectures are considered, because their cache architectures are similar in structure. These architectures differing in inter-chip communication, the former uses Quick-path Interconnect (QPI) the latter uses Hyper-transport (HT). Furthermore, Intel i7 uses an inclusive LLC with MESIF protocol. To observe how these factors impact on task-affinity a complex analysis would be required and this is not the goal of this thesis.

3.3.1 Application's performance

The length of the buffers determine how long the work executed by producers and consumers takes. It was showed in [1] that if buffers used are too short and consequently tasks have very few work to do in user-space side, the parallelism provided by the SMP system is not well profited. For this reason, in [1] a buffer of 4KB was used, in this way, parallelism was well profited. In the following graphics, we compare vanilla and current version of task-affinity in terms of predictability and throughput on different architectures and using different buffer dimensions.

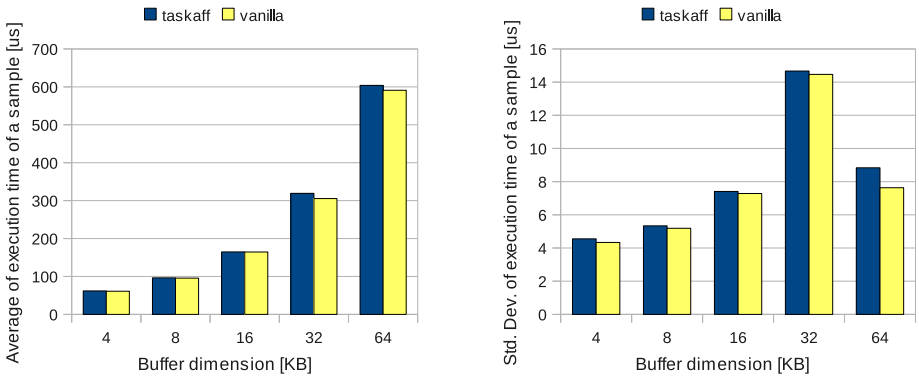


Figure 3.10: Average and Std. Deviation of execution time of a sample on Xeon (current version of task-affinity)

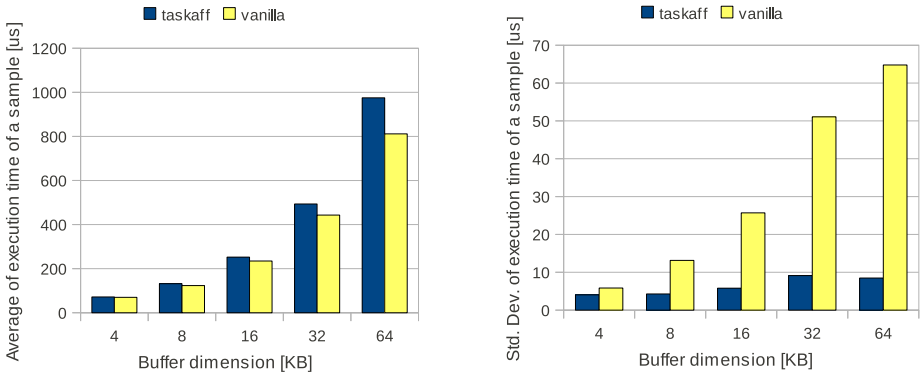


Figure 3.11: Average and Std. Deviation of execution time of a sample on i7 (current version of task-affinity)

A2S Improvement on Table 3.1 is calculated in this way:

$$\frac{A2S_{task\_affinity} - A2S_{vanilla}}{A2S_{vanilla}} \tag{3.2}$$

Where $A2S_{task\_affinity}$ and $A2S_{vanilla}$ are calculated using 3.1.

As shown in graphics and as summarized in Table 3.1, the current version of task-affinity is not effective on Intel Xeon. In terms of throughput task-affinity is not better than vanilla, because the average of execution time of a sample is about the same in both kernels and this is true for both the architectures. In term of predictability, on Intel i7, task-affinity is better than vanilla, especially with buffer greater than 32KB. Task-affinity should reduce both L1 and LLC cache misses, but this fact occurs only on Intel i7, for this reason, on Intel Xeon predictability is degradated,Fig. 3.12 and 3.13.

	A2S Improvement	
	Xeon	i7
$4KB$	-1.89%	2.33%
$8KB$	-0.82%	6.42%
$16KB$	-0.31%	8.42%
$32KB$	-3.96%	6.53%
$64KB$	-3.25%	-5.14%

Table 3.1: A2S Improvement obtained with task-affinity on different architectures with different buffer

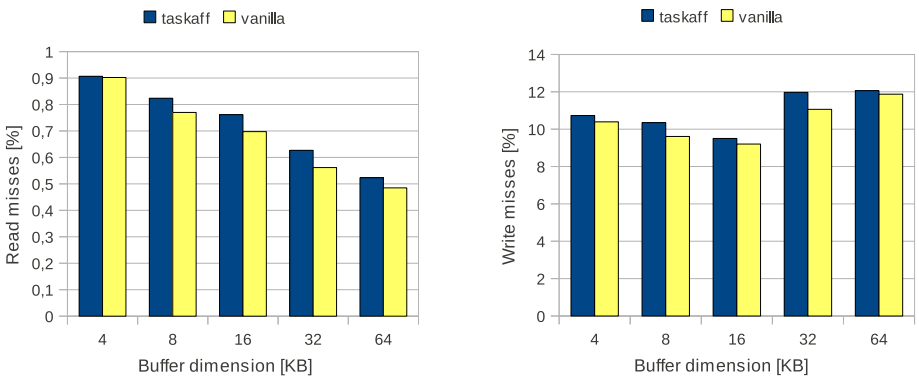


Figure 3.12: L1 Read and write misses on Xeon (current version of task-affinity)

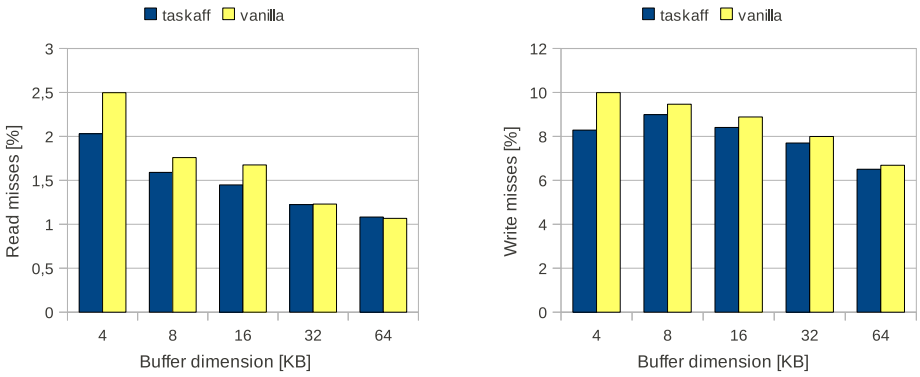


Figure 3.13: L1 Read and Write misses on i7 (current version of task-affinity)

3.3.2 Impact of task migration on execution time predictability

Migration of tasks is an important factor that influences cache misses. Current version of task-affinity is not very effective in terms of reduction of task's migrations. As already described in [1], during the benchmark's execution, *mixer0* or *mixer1* bounce between two different CPUs. This issue is not related to the architecture or the buffer dimension, but it is related to the logic of task-affinity. To understand the reason of this problem see Fig. 3.14.

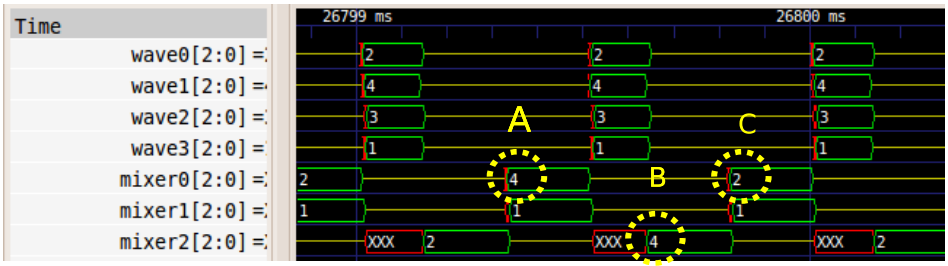


Figure 3.14: Step A: *mixer0* chooses CPU4. Step B: *mixer2* chooses CPU4. Step C: mixer0 chooses CPU2

This picture was made using *gtkwave*, a tool that show in a graphic way the scheduling performed. Consider step A. According to the current task-affinty logic, *mixer0* has to choose a CPU that has executed *wave0* or *wave1*, that is CPU2 or CPU4, *mixer2* has to choose a CPU that have executed *mixer0* or *mixer1*, that is CPU3 or CPU1. *mixer0* chooses the least loaded runqueue, therefore it chooses CPU4. At Step B *mixer2* can choose between CPU4 or CPU1, they have the same number of Real-time task, therefore *mixer2* choose the first CPU in its list, that is CPU4. At step C, *mixer0* has to choose CPU2 or CPU4 again, but it can't choose CPU4, as in step A, because it is still occupied by *mixer2*, so *mixer0* has to be executed on CPU2. This is the problem. Since *mixer0* or *mixer1* choose a CPU occupied by *mixer2*, one of them will have to migrate.

Task's migration can degrade performance because a migrated task could warm up a new cache and it could create new cache interference in a new location already occupied by other tasks. In order to measure how much this migration pattern increases miss rate of the application, the following experiment was performed: two run of the benchmark are performed, in the first run all tasks are pinned on a specific CPU and they can't migrate, in the second run only waves are pinned and mixers can migrate, Tab.3.3.2 summarizes the relative CPU assignments.

What we expect with all tasks pinned and with buffer less than 32KB is a de-

(a) All task pinned

Task	CPU
Wave0	1
Wave1	2
Wave2	3
Wave3	4
Mixer0	1
Mixer1	3
Mixer2	3

(b) Waves pinned

Task	CPU
Wave0	1
Wave1	2
Wave2	3
Wave3	4

Table 3.2: CPUs assignment

crease of L1 read and write miss rates and a reduction of accesses to LLC. With dimensions greater than 32KB, the buffers can't be loaded entirely in L1 cache, therefore we don't know effect on L1 miss rates. Anyway LLC miss rates should diminish. With buffer less than 32KB, accesses to LLC are very different between cases with all task pinned and case with only waves pinned and then, miss rates that occur in two cases are not longer comparable. Note that on Intel i7, LLC is L3 and between L1 and L3 there are private L2 caches that can store buffer of 32KB and 64KB, therefore also with buffers of 32KB and 64KB accesses to LLC should be very different in the considered cases. But as we will see in the following graphics, also on Intel i7, in the considered cases, the number of accesses to LLC is about the same with buffer greater than 32KB.

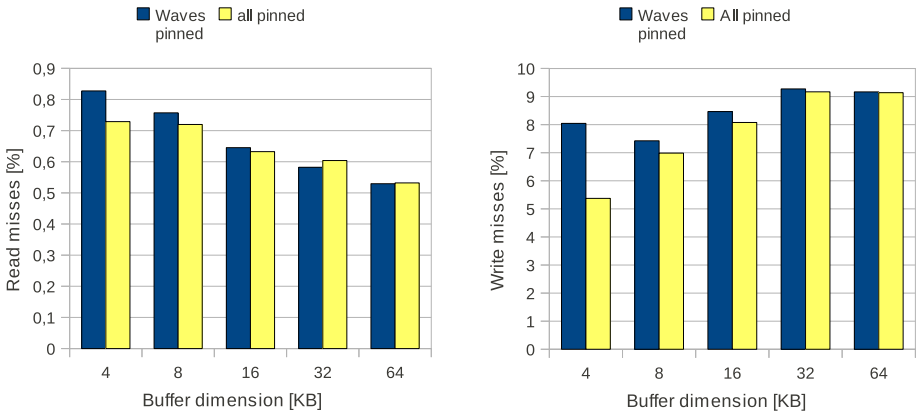


Figure 3.15: L1 Read and Write misses on Xeon (experiment with pinned tasks)

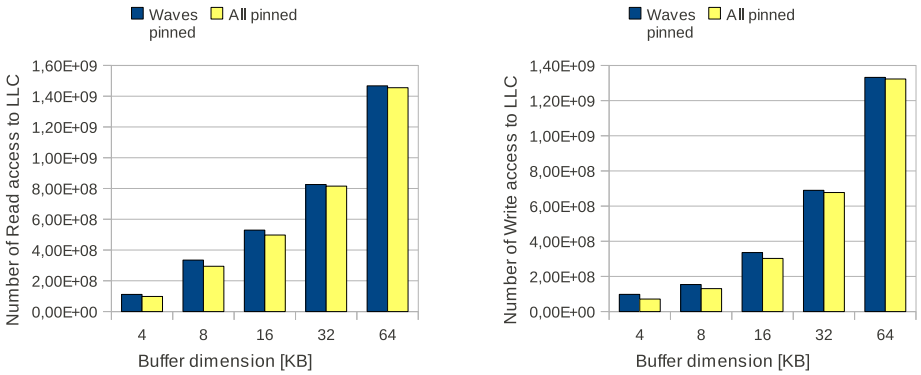


Figure 3.16: Number of LLC Read and Write accesses on Xeon (experiment with pinned tasks)

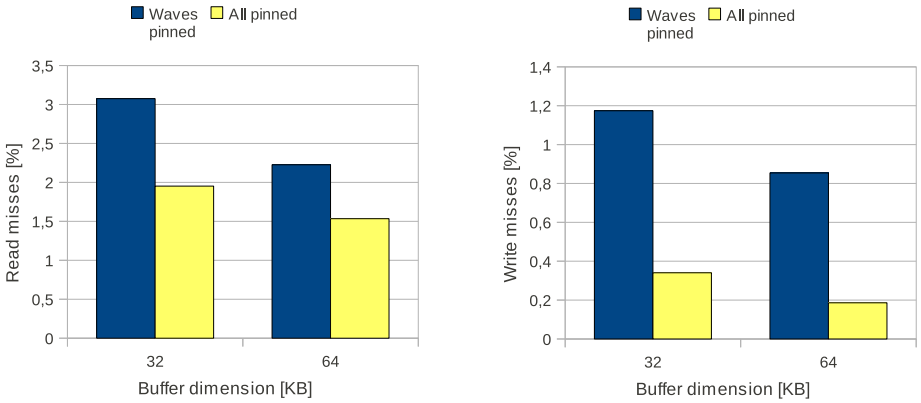


Figure 3.17: LLC Read and Write misses on Xeon (experiment with pinned tasks)

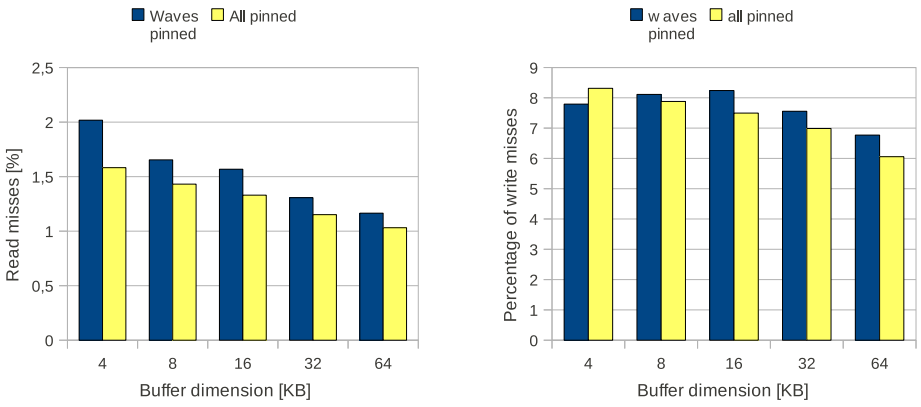


Figure 3.18: L1 Read and Write misses on i7 (experiment with pinned tasks)

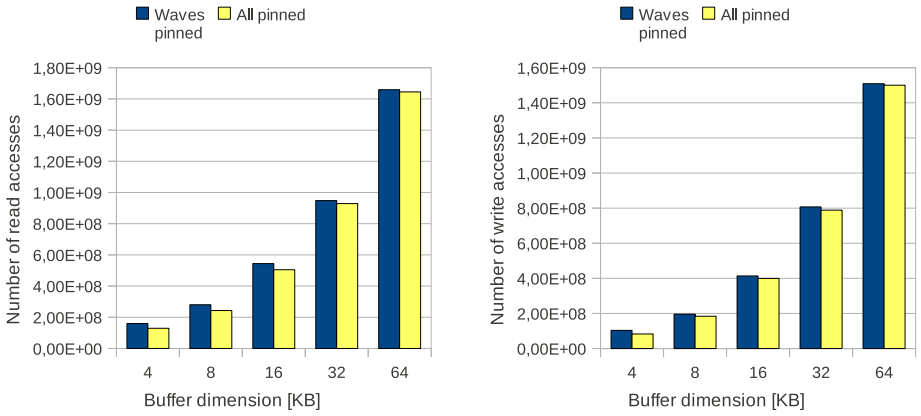


Figure 3.19: Number of LLC read and write accesses on i7 (experiment with pinned tasks)

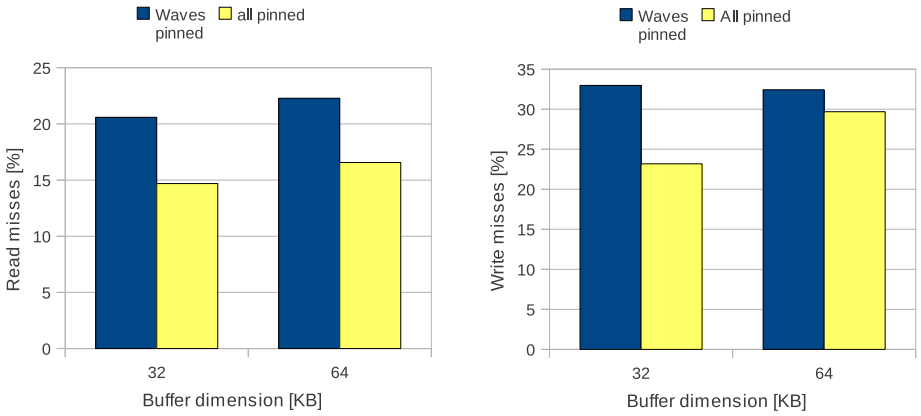


Figure 3.20: LLC miss rate of read and write instructions on i7 (experiment with pinned tasks)

3.3.3 Considerations on experimental results

We can infer from these graphics, that migration pattern have a great influence on read/write miss rate on LLC in both architectures considered.

On Intel Xeon, L1 and LLC read and in particular write misses are greatly increased, this fact is due to cache architecture. On Xeon cores are not all on the same die, for example: assume that core 1 and 2 are on the same die, while core 3 and 4 are on the other die. If a task is executed on core 1 and request a data on core 3, that request will result in a miss, therefore if a task bounces from CPUs that own to different dies, each time it has to warm up the cache. This issue regards especially write operations, because for read operations, hardware prefetchers mitigate this problem.

On Intel i7, instead all core share a common LLC. When a core requires a data, it sends a data request to L3 cache. If data is in L3 and there is a hit, the L3 cache query the ”core valid” bits of the cache line that contains requested data, in order to know which is the core that owns requested data. The core that owns requested data reply to the request with the most recent copy of data. For this reason, each core on i7 can use data in other private cache because all data are contained in LLC.

To get a sense on how much cache miss influence predictability of the application, see Fig. 3.21. In the graphics two run of the benchmark are compared: in the first run all tasks are pinned, while in the second run all tasks can migrate according migration policy of vanilla kernel.

It is interesting to note how the absence of migration improve the predictability

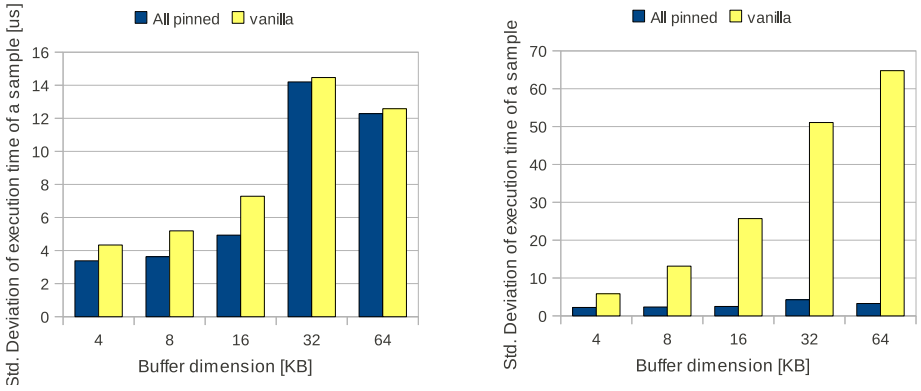


Figure 3.21: Std. Deviation of execution time of a sample on Xeon and i7 (all task pinned compared vanilla)

when buffer used are greater than 32KB that can't be stored in L1 but only in LLC cache or in private L2 (Intel i7).

Performance degradation on Intel Xeon is not only due to migration pattern. As I said before, cache architectures of Xeon and i7 are very different and from previous graphics seems that with cache architecture of Intel i7, task-affinity is more effective, but there is another important aspect that influences performace: it is **Inter-chip communication**. Intel i7 use the new Quick-path Interconnect that is a point-to-point processor interconnect developed by Intel to compete with HyperTransport. This first implementation of this bus achieve 25.6 GB/s, which provides exactly double the theoretical bandwidth of Intel's 1600 MHz FSB, that is the best performance obtainable with FSB. From datasheet, Intel Xeon E5440 use a FSB at 1333 MHz, therefore communication between chips are more faster on i7. Hence, it is clear that communication between cores that belong to different dies is very expensive on Intel Xeon. On Intel Xeon, *mixer0* or *mixer1* could find one buffer in L1 cache and other buffer in a bank of L2 that is shared by the CPU that have execute it. *Mixer2* could find one buffer on L1 one cache, while other buffer is *always* placed on bank of L2 that is not shared by the cpu that has executed *Mixer2*, therefore read latencies are very high, because data are placed on different dies. On i7, instead, *mixer2* could find one buffer in L1 cache and other buffer in L3 cache that is shared among all cores and it is inclusive, therefore to read data is less expensive on i7.

3.4 Task-affinity improvements

Before to explain how the new kernel patch works, it is necessary to remember the concept of task-affinity. We say that two tasks have a task-affinity relationship if they share data and their execution depends upon reading or writing this data [1]. In a producer-consumer application, the producer is the one that writes to the shared buffer, while the consumer is the one that reads it. The consumer depends on data generated by the producer since it needs it in order to be able to run, therefore we say that the consumer has a task-affinity relationship toward the producer.

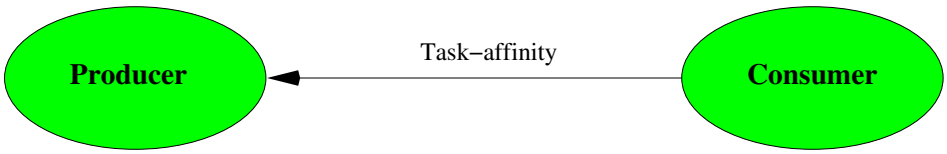


Figure 3.22: Task-affinity relationship between producer and consumer

Each task is provided with a linked list called *taskaffinity_list* that contains all tasks to which it has task-affinity relationship, briefly all its producers. To insert and delete a task in a *taskaffinity_list* two new system calls are provided:

sched_add_taskaffinity: This system call adds a dependency to the current task, i.e. the task that issued the call. It receives, as parameter, the pid of the task the current one will be dependent upon.

sched_del_taskaffinity: When a task does not want anymore to use the task-affinity mechanism, it is possible to remove it through this call. As in the case of inclusion of dependency, it suffices to pass as parameter the pid of the task one wants not to follow anymore.

Thanks to these system calls, the scheduler "knows" which tasks have task-affinity and, in this way, it can schedule consumers after producers in order to enforce reuse of cache memory.

The task-affinity logic influences wake up and migration of a task. As we have previously seen, in *try_to_wake_up* the choice of CPU where to put the to be waken task is made by `select_task_rq_rt` 3.2. This function is modified in this way: given the input task *p*, the function doesn't call `find_lowest_rq` but it loops for all element present in the *p*'s *taskaffinity list* and build a mask, called *affinity_mask*, with CPUs that have executed a task present in *p*'s *taskaffinity_list*. Finished the

loop, the function returns the CPU with the runqueue that has the lowest number of Real-Time tasks. Only if the mask is empty, the function calls `find_lowest_rq` to choose a CPU in the standard way.

In the current version of task-affinity, a task that "respect" task-affinity, that is a task that was enqueued according to its task-affinity relationships, isn't able to migrate. In plain words, `push_rt_task` and `pull_rt_task` can't move tasks that "respect" task-affinity.

The aim of this policy is clear: when a task wakes up, the policy tries to select the best CPU for that task and, if it finds it, it blocks the task on the best runqueue until the task's execution. For this reason the key point of this task-affinity logic is the `select_task_rq_rt`. In the optimal case, producers and consumers will be executed subsequently always on the same CPU.

Nevertheless, in practice, the chosen CPU for p is next to never the optimal CPU. The reason is very simple. The choice of the best CPU, and the enqueuing of task are performed in different moments. When `select_task_rq_rt` is called, it doesn't hold any lock. During the loop, the function has to read what is the content of different runqueues present in the system, these reads are not synchronized. Once the CPU is selected, `select_task_rq_rt` returns and `try_to_wake_up` **takes a lock** on the chosen runqueue, in order to call `activate_task` to perform the enqueuing of the task. From when `select_task_rq_rt` selects the CPU to when the lock is taken, a task with equal or higher priority than p's priority can be inserted in the selected runqueue and, in this way, the next task that will be executed won't be p.

The current version of task-affinity ensures only weak concept of temporal locality because it doesn't ensure, when it is possible, that the next task executed after a producer is a consumer. Another problem of the current version of task-affinity is the migration policy. It is not very flexible. Pull and push functions maintain the system balanced, and guarantee that every CPU executes always the higher priority Real-time task present in its runqueue. Therefore, the denial to pull and push can improve predictability of the application and can degrade significantly the throughput of the application. Predictability is an important aspect for Real-time systems, but if we have a very bad throughput we don't exploit the potentiality of multicore platforms.

The aim of the patch developed in this thesis is to improve the concept of temporal locality in order to execute a consumer immediately after a producer, when

it is possible and to improve the migration policy in order to use also the functions involved in the migration mechanism to exploit the concept of task-affinity. Furthermore, the patch makes task-affinity more robust synchronizing the accesses to data structures used. With this patch we try to improve throughput and predictability of the application.

3.5 Patch structure

The proposed patch is divided in two parts. The first part improves the temporal locality, while the second part introduces mechanisms to synchronize accesses to task-affinity data structures.

3.5.1 Temporal locality

The implemented logic to improve temporal locality is divided in the following parts:

last_tsk field To ensure that a consumer will be the next executed task after a producer, it is necessary to change what `select_task_rq_rt` "sees". As I previously said, during its loop, `select_task_rq_rt` checks for CPUs that **have executed** a task in p's *taskaffinity list*. It means that, in that moment, those CPUs could be executing a task that is not a producer and such, the L1 cache could already be dirty. For this reason, at each runqueue was added a field named `last_tsk` that contains the last task executed in a runqueue. This field is updated at each context switch if the next task to be executed is different from idle. In this way, if current task on runqueue is not idle, this field represents, the task in execution. With this additional field, `select_task_rq_rt` "knows" which is the task currently executing on each runqueue. In this way, CPUs that during `select_task_rq_rt` are executing a task that is not in p's *taskaffinity_list* are not inserted in `affinity_mask`.

enqueue on head This change is not enough. Consider this situation: two different CPUs that we call CPU_A and CPU_B are executing two different instances of `try_to_wake_up`. Respectively, they are called for task p and task q: the former has task-affinity relationship, the latter is a generic Real-time task, both tasks have the same priority. Suppose that the current task on CPU_A is a task in p's *taskaffinity list* and then `select_task_rq_rt` choose CPU_A for p.

Suppose that `try_to_wake_up` that wakes up q chooses CPU_A and enqueue task q on the runqueue of CPU_A. Task p is not yet enqueued, therefore when it will be enqueued, it will be preceded by q and then the next task that will be executed on CPU_A is q.

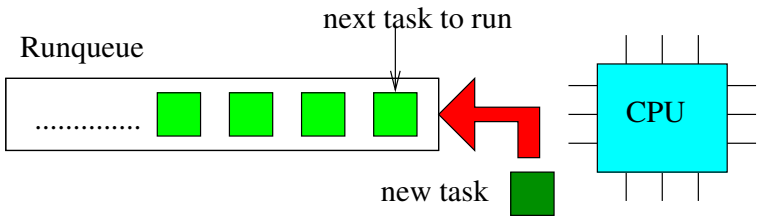


Figure 3.23: Enqueue on head

To resolve this problem a task that "respects" task-affinity is enqueued on the top of a runqueue and not on its tail. In this way, if two Real-time tasks are on the same runqueue and have the same priority, but one of them "respects" a task-affinity, the next task that will be executed is the task that "respects" a task-affinity.

choice of current task Even if a task with task-affinity is enqueued on head in a runqueue, it will be the next executed task only if it is enqueued before it is executed a task that is already on that runqueue. For this reason, it is necessary to optimize the choice made in *select_task_rq_rt*. In `select_task_rq_rt`, when a task with task-affinity, that we call p, has built its *affinity mask*, it checks if in that mask there is the CPU that is executing the `try_to_wake_up` that are waking up it. If this is true, it means that the current task on that CPU is a p's producer and now on that CPU a `try_to_wake_up` is in execution, therefore any other task enqueued on that CPU in that moment can't be executed, because during `try_to_wake_up` kernel preemption is disabled and then it is necessary wait that `try_to_wake_up` finish. Choosing that CPU, p will be the next task executed on that CPU, because it is enqueued on head and it can precede all task enqueued on that CPU during *select_task_rq_rt*.

migration mechanism There is another a problem. It could happen that a task with higher priority is enqueued on the same runqueue where a task with task-affinity is enqueued. In this case, the next task to execute won't be the task with task-affinity. To resolve this problem, migration mechanism is used. When the `try_to_wake_up` has enqueued task p it calls function

`task_woken` in order to checks if *p* can be immediately executed on the selected runqueue or not. If on the runqueue there is a task with priority equal or higher than the *p*'s priority and this task precedes *p*, `push_rt_task` is called and *p* can be pushed on another CPU. To select the CPU where to push *p*, the same mechanism used in `select_task_rq_rt` is adopted. Therefore, *p* will be pushed where it is in execution a task in *p*'s *taskaffinity_list*, if it is impossible standard push criteria are adopted and *p* will be pushed on a CPU that executing a task with lower priority than *p*. Obviously, in order to push a task that "respect" task-affinity, it is necessary insert it in a *pushable list*.

3.5.2 Synchronization

In the current version of task-affinity, accesses to data structures used to manage task-affinity are made by user or by kernel. The resource that must be synchronized is `taskaffinity_list`

Access from user space: the user space can access task-affinity data structures using syscalls `sched_add_taskaffinity`, and `sched_del_taskaffinity`. These functions access the pid of the task received in input in synchronized way, by using the read-write lock tasklist_lock that protects the kernel's internal task list. For this reason, at every moment, only one instance of these syscalls can modify the task-affinity data structure of a task.

Access from kernel space: here the situation is more complex. There are two functions that can access to task-affinity data structures, they are:
`task_affinity_notify_exit` and `select_task_rq_rt`. The former function frees task-affinity data structures and is called when a task is exiting. During this phase, all resources used by a task, pid included, are deallocated. Therefore, when `task_affinity_notify_exit` is called, *tasklist_lock* is acquired. This is not enough because in that moment `select_task_rq_rt` could access the task-affinity data structures of the exiting task, for this reason another layer of synchronization is needed. To resolve this problem, each task has its own read-write lock named *taskaff_lock* to protect its task-affinity data structures.

Chapter 4

Experimental Results

In this chapter I will show the behaviour of task-affinity on different architectures. First of all, we will check if the expected scheduling is performed. After which we will analyze how the optimization proposed influences performance and L1 and LLC miss rates of the application.

In Fig. 4.1 the scheduling that vanilla and the developed patch should perform are represented.

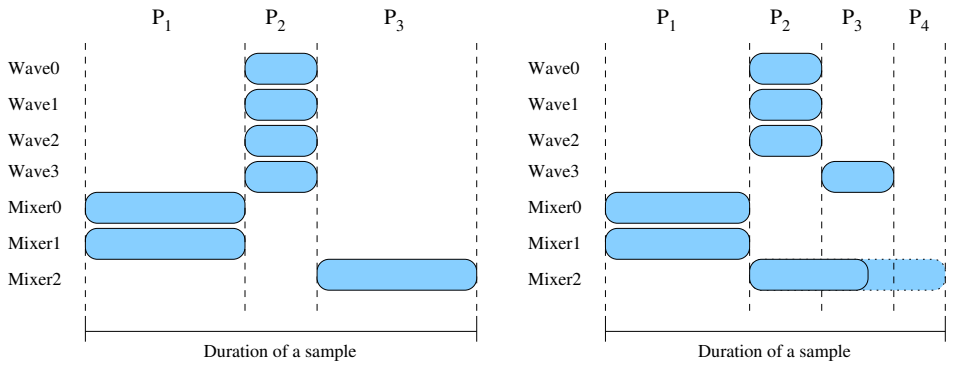


Figure 4.1: Ideal Scheduling performed by vanilla (left) and task-affinity (right).

We have assumed that *mixer0* and *mixer1* have the same duration, all *waves* have the same duration. In the real benchmark, durations of the *waves* are slightly different, durations of the *mixer0* and *mixer1* are slightly different. Furthermore we assume that *mixer0* and *mixer1* never start when *waves* are still in execution.

Considering the time normalized to the duration of one sample, it is possible to express durations of different tasks in relative terms. Using relative durations, it is

possible to estimate the improvement given by task-affinity using *amdhal's law* [1]:

$$Speedup = \left(\frac{P_1}{S_1} + \frac{P_2}{S_2} + ...\frac{P_n}{S_n}\right)^{-1} \tag{4.1}$$

Where P_i is the $i-th$ parallelized portion of the program and S_i is the correspondent parallelization factor. Furthermore the following constraint must be respected:

$$\sum_{i=1}^{N} P_i = 1 \tag{4.2}$$

Before to explain in which portions the serial execution of the benchmark is subdivided, it is necessary define some variables:

T_W**:** average of execution time of a *wave*

$T_{M_{0,1}}$**:** average of execution time of *mixer0* and *mixer1*

T_{M_2}**:** average of execution time of *mixer2*

$T_{W,M}$**:** difference between T_{M_2} and T_W that is $T_{W,M} = T_{M_2} - T_W$

The serialized execution of benchmark can divided into the following portions:

	P_1	P_2	P_3	P_4
vanilla	$T_{M_{0,1}} * 2$	$T_W * 4$	T_{M_2}	–
taskaff	$T_{M_{0,1}} * 2$	$3 * T_W + T_{W,M}$	$T_W * 2$, $\pi_4 > 0$ $T_{W,M} * 2$, $\pi_4 < 0$	$\pi_4 = \lvert T_{M_2} - 2 * T_W \rvert$

Table 4.1: Different Portions of serialized execution of the benchmark

We will see that on Intel Xeon $T_{M_2} \geq 2 * T_W$, while on Intel i7 $T_{M_2} < 2 * T_W$. The showed portions have the following parallelization factors:

	S_1	S_2	S_3	S_4
vanilla	2	4	1	–
taskaff	2	4	2	1

Table 4.2: Parallelization factors of different portions

The index of each parallelization factor represents the portion at which it corresponds

4.1 Comparing to vanilla

In this section, we analyze the behaviour of optimized task-affinity on different architectures. First of all, we desire to know if scheduling performed by task-affinity on different architectures approximates the ideal scheduling showed in Fig. 4.1 and if, as we expected, this scheduling improves throughput of the application. The improvement of throughput was measured using the Amdhal's Law, Eq. 4.1. We have measured the theoretical speedup given by task-affinity, executing the following experiment:

1. We have executed benchmark with only one CPU online using different buffer dimensions. In this way, benchmark is executed in a serialized way.

2. We have recorded average execution time of each thread of the benchmark, and the average execution time of a sample.

3. We have determined to which percentage of the serial execution each portion corresponds. Since we have a serial execution for each buffer dimension used, we have different percentages for each portion, for example: P_0 could be corresponds to 35% on the execution with 4KB, 34% on the execution with 8KB and so on. For this reason, for each portion, we have calculated the average percentage.

4.1.1 Consideration on experimental results

The experiment described in the previous section has been executed for each tested architecture. Besides throughput, we want to know if the predictability of the application is improved and, consequently, if cache miss rates and number of migration are reduced. Finally, since also the migration mechanism is involved in task-affinity, we want to investigate which is the overhead that *pull_rt_task* and *push_rt_task* involve. We have tested task-affinity with Intel Xeon and Intel i7. Here we summarize the main results for these architectures, while detailed measurements are reported in the following subsections.

Throughput: In both Intel Xeon and Intel i7 parallelism is improved, Fig. 4.6 and 4.13. On Intel Xeon, we can see an increment of throughput especially with 32KB and 64KB. While at 4KB there isn't any increment, in fact speedup with

task-affinity is 2.33 while with vanilla is 2.38, Fig. 4.3. This fact occurs because, using buffer of 4KB, parallelism performed in task-affinity is very similar to parallelism performed in vanilla, Fig. 4.2 and 4.3. On Intel i7, instead, we see that task-affinity improves parallelism for every buffer dimension used, Fig. 4.4.

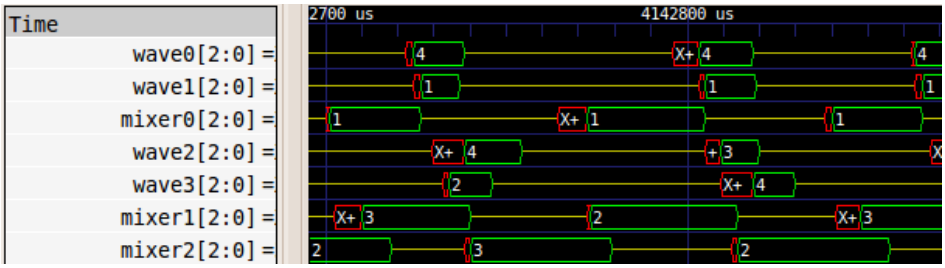


Figure 4.2: trace of benchmark execution on Xeon with buffer of 4KB using task-affinity

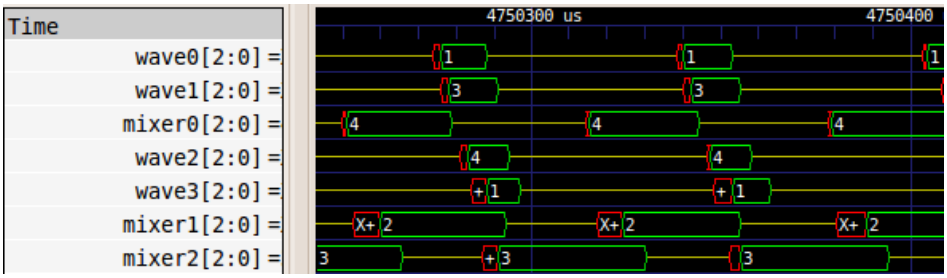


Figure 4.3: trace of benchmark execution on Xeon with buffer of 4KB using vanilla

For what regard for speedups, in both Xeon and i7, the estimated average speedups overestimate the real parallelism performed by task-affinity, nevertheless they are a good estimate. In fact, for Xeon, we have the estimated average speedup of 2.23 and the real average speedup of 2.39: they differ by $\sim$ 6%, Eq. 4.3 and Tab. 4.3. While for i7, we have an estimated average speedup of 2.45 and a real average speedup of 2.50: they differ by $\sim$ 2%, Eq. 4.5 and Tab. 4.4. Therefore, we can conclude that scheduling performed by task-affinity can be well approximated by the scheduling described in Fig. 4.1.

Instead, for vanilla, the estimated average speedups underestimate the real parallelism performed. In fact, for Xeon, we have an estimated average speedup of 1.86 and a real average speedup of 2.19: they differ by $\sim$ 18%, Eq. 4.4 and Tab. 4.3. While for i7, we have an estimated average speedup of 2.11 and a

real average speedup of 2.26: they differ by $\sim$ 7%, Eq. 4.6 and Tab. 4.4. It is clear from these data, that parallelism of the scheduling performed by vanilla is greater with respect to parallelism of the ideal scheduling. In fact if we see Fig. 4.3 we note that there is a fraction of *mixer2* that is executed concurrently with others *waves*, while in ideal scheduling *mixer2* should be executed alone.

Migrations: Number of migration is greatly increased with respect the vanilla. This is not due to architectural details or different buffer dimensions. This fact happens because, at each sample, *waves*, *mixer0* and *mixer1* are excuted on the same CPUs. Since in the next sample *waves* are woken up during the execution of *mixer0* and *mixer1*, they must be scheduled on CPUs different from which that have executed them in the previous sample. For this reason, at each sample, *waves* are executed on different CPUs and, consequently, also other tasks are executed on different CPUs at each sample.

Cache misses: Because of worsening of L1 and LLC cache misses, Fig. 4.8 and 4.9, on Intel Xeon predictability of the application is degradated, especially using small buffer dimension such as 4KB or 8KB, Fig. 4.6. LLC miss rate is greatly increased in task-affinity, because, if a task migrates frequently between two different dies, at each migration it will have to warm up LLC cache and cache misses will occur. The same goes for L1 cache misses, also in that case a migration between CPUs that are in different dies increase L1 miss rates. Nevertheless with dimension greater than 8KB and especially greater than 32KB predictability is improved, Fig. 4.6. On Intel i7, instead, thanks to the inclusive shared LLC, a core can access data contained in all caches of other cores, consequently, L1 and above all LLC cache misses are reduced, Fig 4.15 and 4.16. The diminishing of cache misses impacts significantly on application predictability, Fig.4.13.

Migration functions: As we can see from Fig. 4.10 and 4.17, in both Xeon and i7, with task-affinity the number of calls to `push_rt_task` is greatly reduced with respect to vanilla. This fact happens because if a CPU is selected according to task-affinity criteria and the enqueued task with task-affinity is the next to be executed on the selected runqueue, `push_rt_task` for that runqueue is not called. Other tasks on that runqeueue that have to migrate will be moved by `pull_rt_task`. Instead, with *pull_rt_task* task-affinity is not very effective, in fact, with task-affinity, *pull_rt_task* executes more work than in vanilla, Fig.

4.11 and 4.18.

```
1553 static int pull_rt_task(struct rq *this_rq)
1554 {
                .......
1559          if (likely(!rt_overloaded(this_rq)))
1560                  return 0;
1561
1562          for_each_cpu(cpu, this_rq->rd->rto_mask) {
                .....
```

Figure 4.4: A portion of the `pull_rt_task` method

The explanation is simple: at line 1559, `pull_rt_task` checks for overloading runqueues. If in the system there is any overloaded runqueue, *pull_rt_task* searches which runqueues are overloaded and tries to pull a task from them. With task-affinity, only 3 *waves* are executed concurrently and one *wave* has to wait that *mixer2* finishes. Therefore at each sample there is an overloaded runqueue in the system. For this reason, when *pull_rt_task* is called, almost certainly it will enter in loop at line 1562 in order to pull tasks from an overloaded runqueue. With vanilla instead, all waves are executed concurrently, therefore there are less probabilities to have overloaded runqueues and thus *pull_rt_task* will exit at line 1560. The overhead of *pull_rt_task* influences predictability of an application especially with buffer of 4KB because execution time of each thread is relatively small.

In conclusion, to get a sense of how task-affinity improves the performance of the benchmark, look at table 4.3, where are reported values of metric A2S (Eq. 3.1) used to characterize how task-affinity improve throughput and predictability with respect to vanilla. As we can see, task-affinity is well exploited by Intel i7, look at 4.4, while on Intel Xeon we have some advantage with buffer greater than 4KB. A2S improvements are calculated using Eq. 3.2

4.2 Intel Xeon

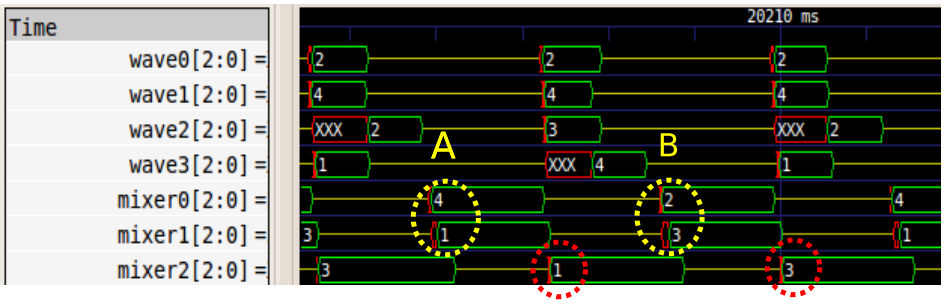


Figure 4.5: Scheduling performed by new version of task-affinity on Intel Xeon.

As we can see in Fig. 4.5, the scheduling performed is correct. We see that *mixer2* can precede one of the waves and improve parallelism. We see that *mixer0* chooses the best cpu in term of temporal locality, for example: in step A *mixer0* chooses CPU4 and not CPU2, because on CPU2 was executed *wave2*, therefore L1 cache could be dirty, instead on CPU4 the last task executed is *wave1*, therefore L1 cache should be clean. Also in step B, it is possible to note how *mixer0* take care about the last task executed on CPU4 choosing CPU2.

According the showed data, Tab. 4.3, it possible to estimate the average speedups for task-affinity and vanilla.

$$Speedup_{taskaff} = \left(\frac{0.4}{2} + \frac{0.35}{4} + \frac{0.18}{2} + \frac{0.07}{1}\right)^{-1} = 2.23 \tag{4.3}$$

$$Speedup_{vanilla} = \left(\frac{0.4}{2} + \frac{0.35}{4} + \frac{0.25}{1}\right)^{-1} = 1.86 \tag{4.4}$$

(a) Average of execution times of each task on serialized execution (us). The speedups for parallel execution are computed as: serialized execution time divided by sample time.

	waves	$mixer_{0,1}$	$mixer_2$	serialized	Sample time		Speedup	
				exec. time	va	ta	va	ta
4*KB*	12.18	29.59	38.84	146.76	61.58	63.08	2.38	2.33
8*KB*	17.74	42.72	55.57	211.98	95.54	90.60	2.22	2.34
16*KB*	31.47	70.35	88.53	355.13	165.23	150.00	2.15	2.37
32*KB*	60.12	128.72	155.34	653.27	306.69	266.42	2.13	2.45
64*KB*	115.42	246.2	285.62	1239.69	592.30	502.09	2.09	2.47
Average							2.19	2.39

(b) Portions of serialized execution

	P_1		P_2		P_3		P_4	
	va	ta	va	ta	va	ta	va	ta
4*KB*	40%	40%	33%	33%	26%	17%	–	9%
8*KB*	40%	40%	33%	33%	26%	17%	–	9%
16*KB*	40%	40%	35%	35%	25%	18%	–	7%
32*KB*	39%	39%	37%	37%	24%	19%	–	5%
64*KB*	40%	40%	37%	37%	23%	19%	–	4%
Average	40%	40%	35%	35%	25%	18%	–	7%

(c) Sample production time (us)

	taskaff			vanilla			A2S
	avg	var	A2S	avg	var	A2S	Improvement
4*KB*	63.08	44.70	76.45	61.58	12.42	68.63	-11%
8*KB*	90.60	24.18	100.43	95.54	20.31	104.55	4%
16*KB*	150.00	25.75	160.15	165.23	42.47	178.27	10%
32*KB*	266.42	27.17	276.84	306.69	43.31	319.85	13%
64*KB*	502.09	74.54	519.35	592.3	81.45	610.35	15%

Table 4.3: Data used to calculate speedups and A2S improvements of new version of task-affinity and vanilla on Xeon

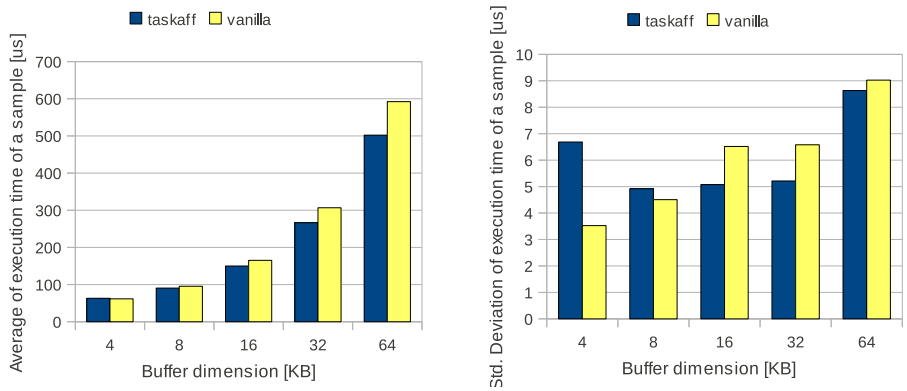


Figure 4.6: Average and Std. Deviation of execution time of a sample on Xeon (new version of task-affinity)

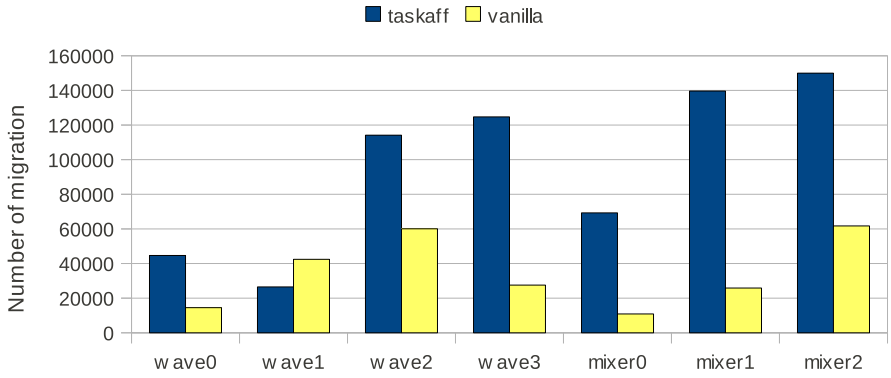


Figure 4.7: task migration on Xeon (new version of task-affinity)

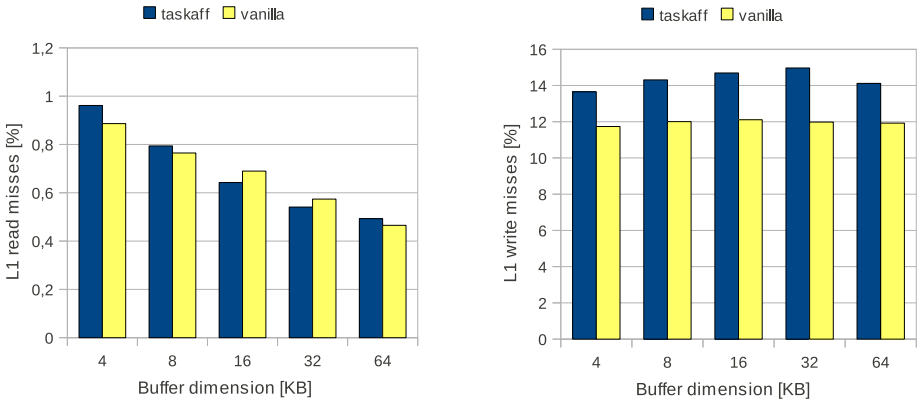


Figure 4.8: L1 Read and Write misses on Xeon (new version of task-affinity)

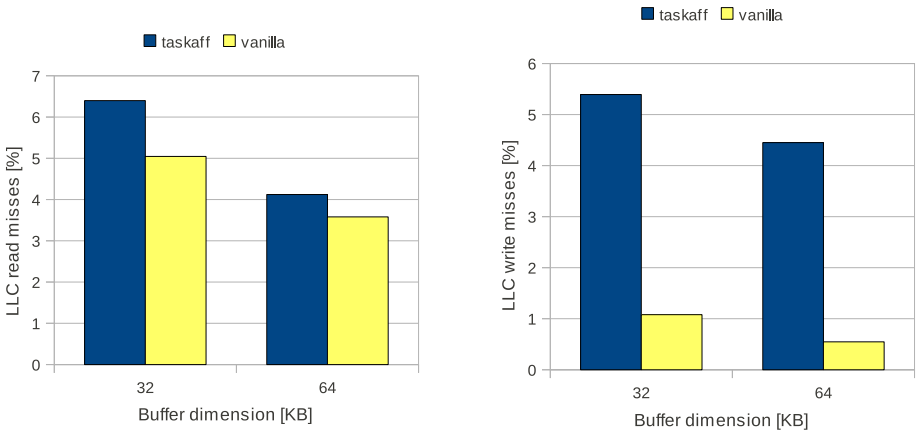


Figure 4.9: LLC Read and Write misses on Xeon (new version of task-affinity)

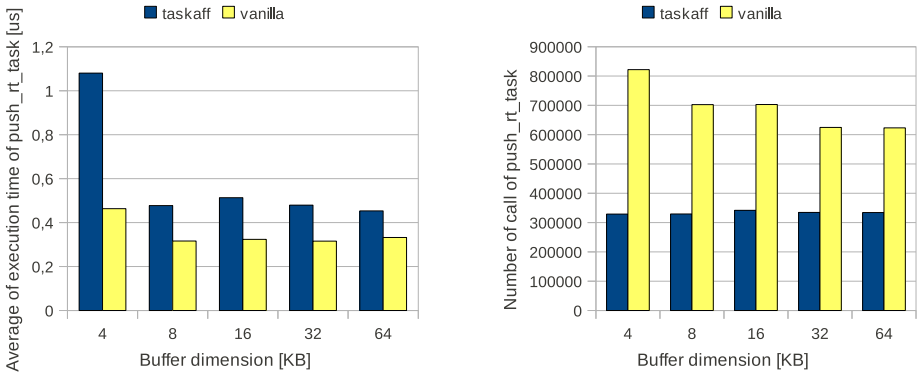


Figure 4.10: Average of execution time of a call to push_rt_task and number of call to push_rt_task on Xeon (new version of task-affinity)

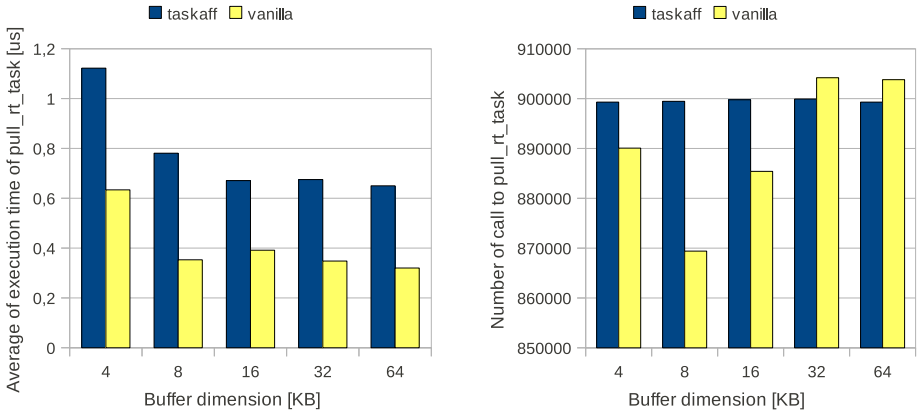


Figure 4.11: Average of execution time of a call to pull_rt_task and number of call to pull_rt_task on Xeon (new version of task-affinity)

4.3 Intel i7

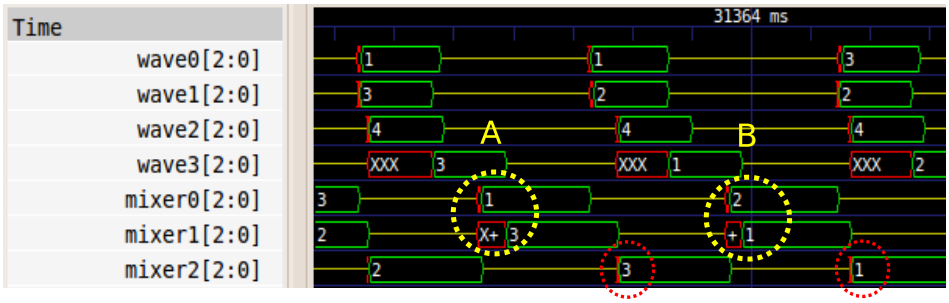


Figure 4.12: Scheduling performed by new version of task-affinity.

We can see from Fig.4.12, that also in this case the scheduling performed can be approximated with the ideal scheduling, Fig. 4.1. We can see how in step A and B *mixers* choose the correct CPUs according to their task-affinity relationships.

Also in this case, according to recorded measurement the estimated average speedups are:

$$Speedup_{taskaff} = \left(\frac{0.32}{2} + \frac{0.49}{4} + \frac{0.13}{2} + \frac{0.06}{1}\right)^{-1} = 2.45 \tag{4.5}$$

$$Speedup_{vanilla} = \left(\frac{0.32}{2} + \frac{0.49}{4} + \frac{0.19}{1}\right)^{-1} = 2.11 \tag{4.6}$$

(a) Average of execution times of each task on serialized execution (us). The speedups for parallel execution are computed as: serialized execution time divided by sample time.

	waves	$mixer_{0,1}$	$mixer_2$	serialized	Sample time		Speedup	
				exec. time	va	ta	va	ta
4*KB*	17.86	27.31	29.25	155.31	68.25	61.44	2.28	2.53
8*KB*	33.17	46.39	51.88	277.32	122.42	108.22	2.27	2.56
16*KB*	63.65	82.49	96.54	516.11	236.35	206.03	2.18	2.51
32*KB*	124.85	156.46	182.30	994.59	444.38	401.53	2.24	2.48
64*KB*	250.27	279.39	356.96	1916.82	826.83	793.50	2.32	2.42
Average							2.26	2.50

(b) Portions of serialized execution

	P_1		P_2		P_3		P_4	
	va	ta	va	ta	va	ta	va	ta
4*KB*	35%	35%	46%	46%	19%	15%	–	4%
8*KB*	33%	33%	48%	48%	19%	14%	–	5%
16*KB*	32%	32%	49%	49%	19%	13%	–	6%
32*KB*	31%	31%	50%	50%	19%	12%	–	7%
64*KB*	29%	29%	52%	52%	19%	12%	–	7%
Average	32%	32%	49%	49%	19%	13%	–	6%

(c) Sample production time (us)

	taskaff			vanilla			A2S
	avg	var	A2S	avg	var	A2S	Improvement
4*KB*	61.44	20.38	70.47	68.25	34.02	79.91	12%
8*KB*	108.22	25.54	118.32	122.42	167.94	148.34	20%
16*KB*	206.03	77.33	223.61	236.35	586.69	284.80	21%
32*KB*	401.53	328.09	437.76	444.38	2688.34	548.08	20%
64*KB*	793.50	1423.98	868.97	826.83	6469.23	987.70	12%

Table 4.4: Data used to calculate speedups and A2S improvements of new version of task-affinity and vanilla on i7

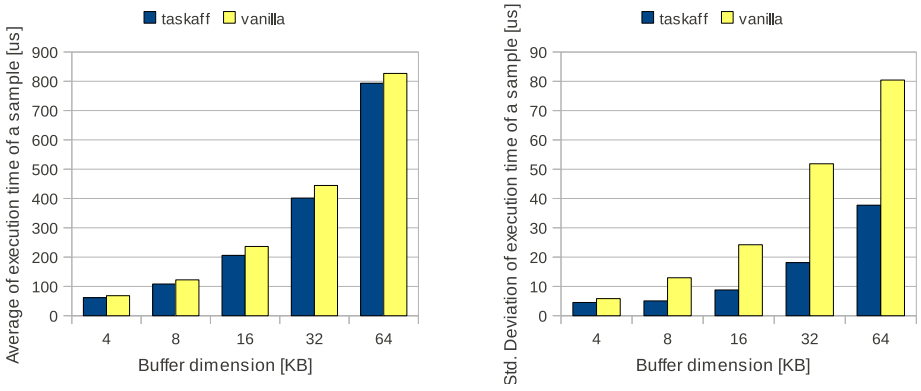


Figure 4.13: Average and Std. Deviation of execution time of a sample on i7 (new version of task-affinity)

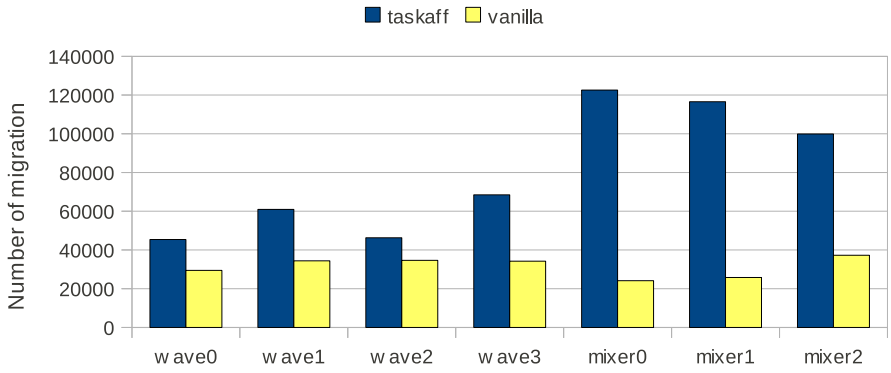


Figure 4.14: task migration on i7 (new version of task-affinity)

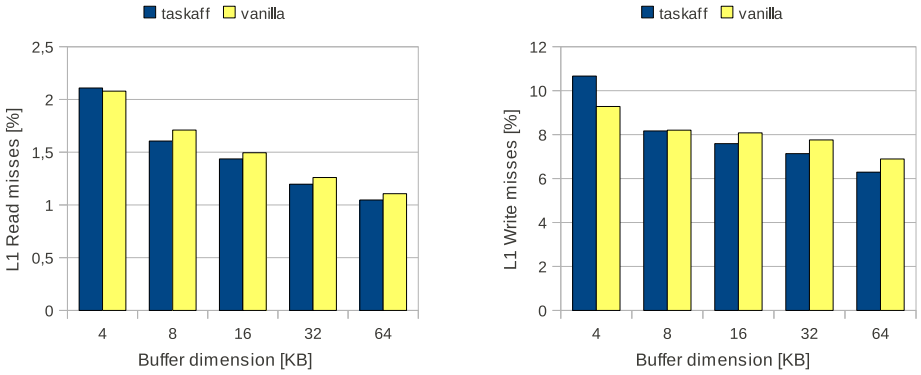


Figure 4.15: L1 Read and Write misses on i7 (new version of task-affinity)

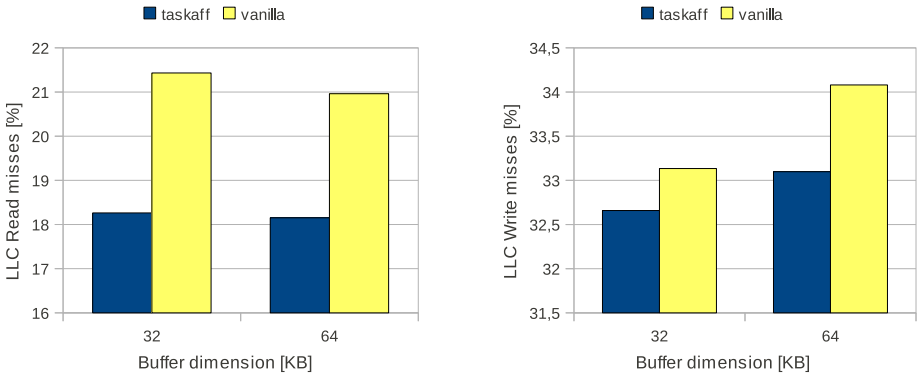


Figure 4.16: LLC Read and Write misses on i7 (new version of task-affinity)

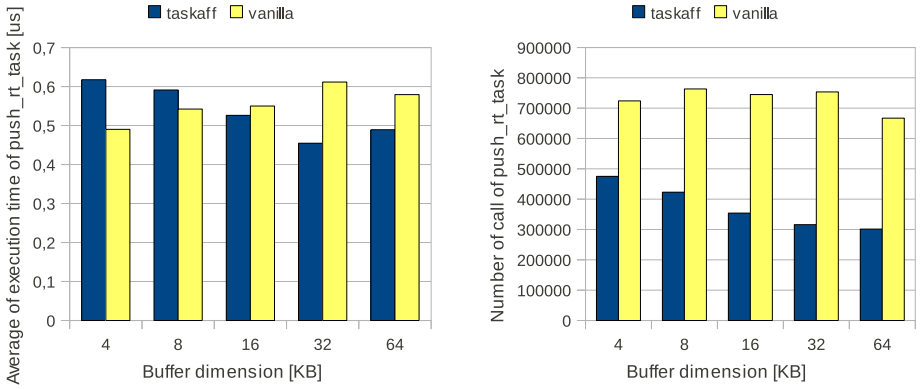


Figure 4.17: Average of execution time of a call to push_rt_task and number of call to push_rt_task on i7 (new version of task-affinity)

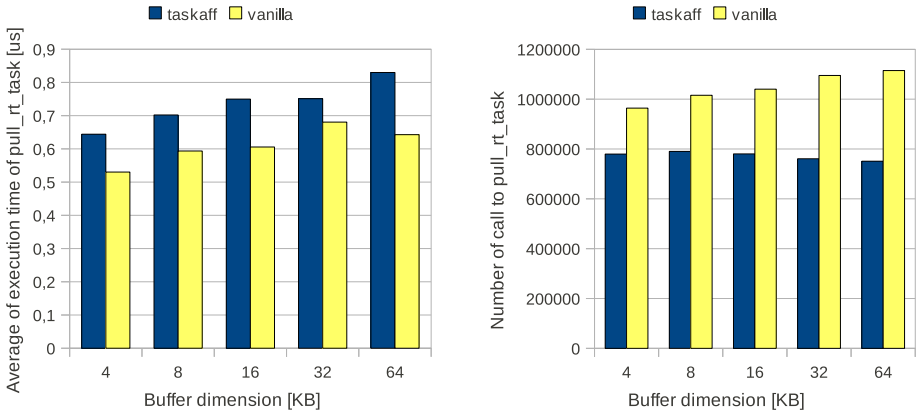


Figure 4.18: Average of execution time of a call to pull_rt_task and number of call to pull_rt_task on i7 (new version of task-affinity)

Chapter 5

Conclusions and future developments

In this work we have investigated behaviour of task-affinity on different architectures. Task-affinity is based on assumption that, if tasks share a great amount of data and we have an SMP architecture, then occasionally executing them on the first available processors will increase the cache-miss rate. If we move tasks to be executed near to where data was produced, then we spare some cache-misses [1]. We have demonstrate that this assumption is not always true. Migrations of tasks play an important role on predictability of the applications. When a task migrates, data could be moved from one cache to another. As we have seen on Intel Xeon and as it is demonstrated in [4], this operation may involve high latency that depends on cache architecture, inter-chip communications and other hardware factors. Therefore it is clear that it is not enough to execute tasks that share common data on the same CPUs, but it is necessary guarantee that tasks that share common data find what they need possibly in L1 cache. For this reason, we have improved the concept of temporal locality.

The experimental results show that task-affinity is effective on Intel i7, where there is an average speedup in term of A2S of $\sim$ 17%. We have increased task migration, nevertheless we have improved throughput and predictability. It is clear that architecture of Intel i7 mitigates the side effects of the high number of migrations of tasks. On Intel Xeon task-affinity is effective only with buffer greater than 4KB, in that case there is an average speedup in term of A2S of $\sim 10,5\%$. It is clear that the effect of migrations of tasks is more significant on this architecture.

We could obtain results still better with a more effective migration policy. In this patch, in order to improve the temporal locality, we have included *push_rt_task* in the task-affinity logic and we have seen that, because of the scheduling performed, *pull_rt_task* has an higer overhead than vanilla.

We conclude that the mechanism proposed brought, in fact, an improvement of throughput and on determinism of the entire application, especially with buffers of big dimension as 32KB. It is interesting that these improvements take place even if L1 and LLC miss rates increase (Intel Xeon) therefore it is clear that, according to the cache architecture used, cache misses have a different impact on performance of application.

5.1 Future Works

During the development of task-affinity, NUMA architectures were never considered. In the final part of this thesis we have started to analyze the behaviour of task-affinity on AMD Opteron, a NUMA architecture. In a first trial, using some kernel facilities, we have held all tasks to be executed only on one node, in order to simulate an SMP architecture. We have obtained the following results:

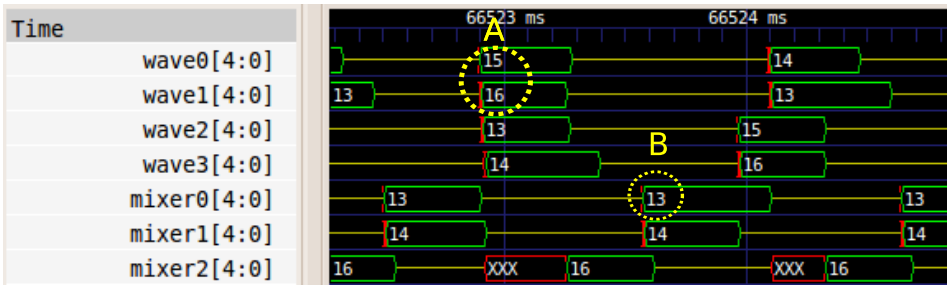


Figure 5.1: Scheduling performed by new version of task-affinity on Opteron (new version of task-affinity)

Task-affinity on this machine doesn't work properly. We can see in step B that *mixer0* doesn't choose the correct CPU. The incorrect behaviour of task-affinity is also reflected by Fig. 5.2, where we can see a worsening of throughput and predictability. It is possible that the task-affinity doesn't work, because of a lot of kernel threads used to manage load balancing and others kernel activities in NUMA architectures: but this is only an hypothesis. According these results, it is clear that task-affinity needs the support for NUMA architectures, this is the first step to improve task-affinity.

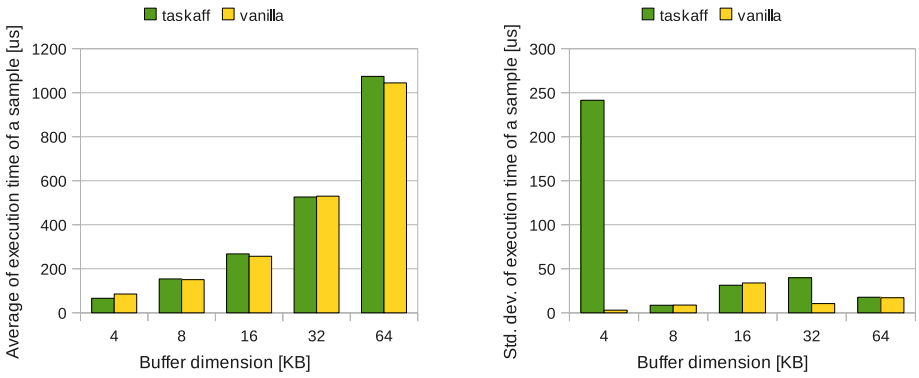


Figure 5.2: Average and Variance of execution time of a sample Opteron (new version of task-affinity)

In this work, we have seen that cache misses have a different impact on determinism of application on different architectures, it is necessary to estimate how much cache misses impact on application performance and in particular on determinism, in order to understand if the task-affinity could improve significantly or not the determinism of application on a given architecture.

Another possible improvement is to optimize the migration policy, in order to include also *pull_rt_task* in task-affinity logic and finally, it would be better not to use system calls to define dependencies among tasks, but, instead, to rely in other mechanisms, such as a profiler [2], that infers automatically the dependencies among tasks. The reason is that it is not desirable to modify present applications to use the task-affinity. The mechanism of adding and removing dependencies would be rather the same: only the user interface would need to be changed.

Chapter 6

Estratto in lingua italiana

Nella sfida per fabbricare la CPU piú performante, i progettisti hardware si devono confrontare con un problema molto complicato. Da un lato la Legge di Moore non si puó piú applicare alla capacitá computazionale di una macchina, cioe: la capacitá computazionale non si raddoppia piú ogni 18 mesi, come nel passato. Dall' altro lato, il consumo di potenza continua ad aumentare piú che linearmente con il numero di transistors inclusi in un chip e la Legge di Moore vale ancora per il numero di transistor in un chip. Parecchie soluzioni sono state adottate per risolvere questo problema. Alcune die esse cercano di ridurre il consumo di potenza sacrificando la capacitá computazionale, di solito usando il frequency scaling, il voltage throttle o entrambi. Altre soluzioni cercano di in aumentare l' Instruction Level Parallelism (ILP) all'interno di un processore, in modo da avere piú capacitá computazionale da un processore senza aumentare il consumo di potenza. Tuttavia, oggigiorno il costo di un cache miss (che potrebbe mandare in stallo la pipeline) o di uno sbaglio nella predizione di un branch (che potrebbe invalidare la pipeline) sono diventati troppo elevati. Giá nei primi anni 2000, era chiaro che la via piú efficace per aumentare la capacitá computazionale e ridurre il consumo di potenza consisteva nel parallelizzare l' esecuzione dei task. Per questo motivo é stato introdotto il Simultaneous MultiThreading (SMT). Questa tecnologia permetteva di eseguire due thread in modo concorrente sulla stessa CPU, in questo modo é possibile avere una grande parallelizzazione dell' esecuzione dei task. In base a come sono organizzate le memorie, i sistemi a multiprocessore sono classificati in due gruppi:

Centralised Shared Memory Architectures: in queste architetture, ci sono diversi core connessi ad una memoria condivisa. Se tutti i cores sono uguali, queste

architetture si chiamano simmetric multiprocessor (SMP).

Distributed Memory Architecture: in queste architetture, ciascun processore ha il suo proprio modulo di memoria e il tempo di accesso alla memoria dipende dalla locazione di memoria relativamente al processore che vi accede. In questa categoria di architetture sono incluse le Non-uniform memory access (NUMA).

Le architetture multicore sono state adottate da molte industrie di microprocessori. I chips Dual-core sono ormai una soluzione comune e numerose opzioni da 4 e 8 cores sono disponibili. Negli anni a venire, il numero di core integrati in un chip continuerá a crescere: Intel ha annunciato che rilascerá per chips a 80 core per il 2013. Il passaggio alle tecnologie multicore é un evento di svolta, poiché esso porta le piattaforme di computazione standard ad essere multiprocessori.

Anche molti sistemi embedded stanno iniziando ad usare architetture multicore, perché questi processori forniscono un grande incremento di capacitá computazionale con un basso incremento di consumo di potenza e questo é un aspetto importante per questo tipo di dispositivi. Ma cé un' ostacolo all' uso di queste architetture in nel settore embedded e in particolare nei sistemi Real-Time. Immaginate questa situazione: ci sono 3 tasks Real-time: A, B e C. A usa 512KB di memoria, B ne usa 768KB e C ne usa 256KB. La nostra piattaforma é un Dual-core provvisto di cache condivisa onchip di 1MB. Ci sono due casi possibili di scheduling. Nel primo caso A e C (o B e C) sono schedulati concorrentemente. C'é abbastanza spazio in cache per allocare le risore dei due task, perció non c'é nessun problema. Nel secondo caso A é schedulato insieme a B: avviene cache thrashing. Le performante dei due task potrebbero peggiorare rispetto al caso precedente, perché non c'é alcuna garanzia che A e B possano trovare i dati necessari nella cache condivisa e, inoltre, é impossibile prevedere la durata di A o B, perché se A é schedulto insieme a B, esso avrá una certa durata. Invece, se A é schedulato insieme a C, esso avrá un' altra durata, in altre parole: la durata di un task dipende da quale altro task é schedulato con esso e, per questo motivo, utilizzando i comuni algoritmi di scheduling Real-Time, le tecniche di analisi dei tempi per il software embedded usate nei sistemi a single core non sono piú utilizzabili nei sistemi multicore. Per questa ragione, sono necessarie nuove tecniche per stimare il worst-case execution time (WCET) dei tasks Real-Time.

Risulta chiaro quindi, che lo scheduler gioca un ruolo importante nel migliorare la predicibilitá e le performance delle applicazioni. Oggigiorno é importante sviluppare algoritmi di scheduling "cache-aware", cioé uno scheduler che, per scegliere la CPU dove mettere un task, considera come i tasks schedulati usano la memoria cache, in modo da evitare il cache thrashing. Questa tesi é il proseguimento del lavoro svolto da Lucas De Marchi, egli ha provato a rendere lo scheduler Real-Time di Linux cache-aware, introducendo il concetto di task-affinity.

6.1 Stato dell'arte

Sebbene il problema di progettare un algoritmo di scheduling cache-aware sia vecchio e ben conosciuto da oltre 20 anni e i multicore siano ampiamente diffusi, attualmente i sistemi operativi in commercio non implementano questo tipo di algoritmi e in letteratura sono presenti solo pochi lavori che studiano questo problema. I lavori di ricerca piú recenti relativi a questo argomento consistono in attivitá di profiling, che hanno lo scopo di dimostrare come una condivisione scorretta della cache da parte di task concorrenti possa rallentarli e causare: throughput sub-ottimale, cache thrashing e, in alcuni casi, task starvation per i task che non riescono ad occupare spazio sufficiente nella cache necessario per fare buoni progressi nella loro esecuzione. Il primo lavoro ben documentato relativo a questo tipo di scheduling é stato sviluppato all' universitá di Stanford. Alla fine degli anni 80, il Computer Systems Laboratory di Stanford sviluppó un prototipo di un multiprocessore a memoria condivisa chiamato DASH. La sua architettura era molto simile a quella usata nel moderni processori SMP; DASH poteva incorporare fino a 64 processori RISC. Per poter sfruttare le piene potenzialitá della macchina, venne sviluppato un runtime system ad hoc ed un linguaggio: COOL. Esso era un' estensione del C++ che introduceva alcuni statements per facilitare la definizione del parallelismo a grana fine o grossa e per definire quali erano i pattern di accesso alla memoria eseguiti dall' applicazione. Il compilatore di COOL era in grado di estrarre automaticamente informazioni sul parallelismo del programma che potevano eesere usate da architetture che, come DASH, supportavano un tale livello di concorrenza, ed informazioni sull' uso della cache fatto dall' applicazione. Usando queste informazioni, il runtime system poteve assicurare il parallelismo desiderato dal programmatore e cercare di ridurre il miss rate di ciascun task, perché il sistema "sapeva", per ciascun task, quali fossero gli oggetti referenziati da esso; in questo modo schedulava

oggetti e tasks in modo da renderli vicini. In parole povere, usando informazioni aggiuntive fornite dal programmatore e sfruttando il principio di localitá dei dati, il runtime system decideva dove allocare gli oggetti e assegnava ad un task una CPU che conteneva nella sua cache gli oggetti referenziati da esso. Il progetto COOL mostra come un uso intelligente della cache sia un problema che coinvolge tutti gli aspetti dell' ingegneria del software, dal compilatore allo scheduler e il sistema di gestione della memoria.

Altri Lavori di ricerca svolti in questi anni sfruttano un' altra strategia. Essi non introducono un nuovo linguaggio di programmazione o sofisticati ambienti di runtime, ma implementano un profiler grezzo che, a runtime, inferisce quanto spazio nella cache é richiesto da un task, in modo da inferire quali tasks potrebbero causare cache thrashing, se fossero schedulati concorrentemente. Per fare questo lavoro, il profiler esegue una fase periodica di taratura in cui analizza il miss rate di ciascun task, in questo modo, é possibile capire l' ammontare di spazio usato da un task. Sulla base di queste informaizoni due o piú tasks sono schedulati su diverse CPUs solo se non causano cache thrashing. Questi lavori non sono efficaci come COOL, ma presentano buoni risultati con i benchmark presenti nella suite SPEC2000, inoltre alcuni di questi lavori sono stati sprimentati anche in sistemi embedded con esiti soddisfacenti.

6.2 Obiettivi di questa tesi

Lo scopo principale di questa tesi é l' ottimizzazione della versione attuale della task-affinity. In un primo passo, analizzeremo come la task-affinity si comporta su diverse architetture multicore; in particolare sull' Intel Xeon E5440 e sull'Intel i7 870. Queste due architetture sono state scelte perché hanno una architettura delle cache molto diversa tra di loro e inoltre, anche il sistema di comunicazione tra chip é molto diverso in termini di prestazioni. Con l'analisi che si vuole eseguire, cercheremo di capire quali sono gli aspetti dell'attuale logica della task-affinity da migliorare per poter sfruttare al meglio le architetture testate, per esempio: attualmente, come descritto in [1], la politica di migration effettuata non é molto efficace, perché alcuni task tendono a rimbalzare da una CPU all'altra, ad ogni iterazione del benchmark. Riteniamo che questo fenomeno degradi le peformance ottenibili con la task-affinity, perché spesso un task che migra deve anche eseguire il warm up della cache della CPU su cui esso é migrato, e questo aumenta il miss rate. Quello

che ci aspettiamo é che l'analisi metta in luce, per ogni architettura, quanto questo "migration pattern" incida sul miss rate di un task.

Tenendo conto dei risultati presentati nella fase di analisi, proporremo un' ottimizzazione volta a migliorare la localitá temporale dei dati, in modo da diminuire il miss rate di un task; per fare ció includeremo nella logica della task-affinity anche le funzioni usate per la migrazione dei task. Nell'ultima parte dell' ottimizzazione introdurremo la sincronizzazione per le strutture dati usate nella task-affinity. Tutte le misure effettuate sulla task-affinity e sul vanilla sono state eseguite usando il benchmark usato in [1].

Quindi gli obbiettivi di questa tesi possono essere riassunti come:

1. analizzare il comportamento dell'attuale versione della task-affinity su diverse archietture per capire quali aspetti migliorare.

2. ottimizzare la versione attuale della task-affinity, migliorando la politica di migrazione dei task e migliorando la localitá temporale dei dati garantita dalla task-affinity.

Tutte le patch sviluppate in questa tesi si basano sulla versione 2.6.34 del kernel Linux

6.3 Organizzazione della tesi

Chapter 2: Nella prima sezione del capitolo, discutiamo i problemi derivanti da un'uso sbagliato della cache. Vedremo come un'uso scorretto della cache possa degradare le performance e ridurre notevolmente il determinismo delle applicazioni. Nella sezione successiva, abbiamo effettuato uno studio sulle diverse architetture della cache focalizzandoci su quei dettagli architetturali che spesso non sono ben documentati, come: i protocolli di coerenza, cache inclusive o esclusive ecc. Nell'ultima sezione, é presente una classificazione degli algoritmi di scheduling cache aware sviluppati negli ultimi anni, analizzando vantaggi e svantaggi di ciascun algoritmo presentato.

In Chapter 3: Nella prima sezione analizziamo l'ottimizzazione implementata in questa tesi. La prima sezione analizza il comportamento della task-affinity su diverse architetture, in modo da capire come la task-affinity possa essere ottimizzata. La sezione successiva tratta la sua implementazione in Linux.

Chapter 4: presenta i risultati sperimentali che riguardano: la correttezza della soluzione, cioé se gli scheduling eseguiti sono approssimabili con gli scheduling ideali definiti all'inizio del capitolo, e i miglioramenti rispetto alla versione attuale della task-affinity.

In Chapter 5: illustreremo le conclusioni del lavoro svolto, riassumendo quali risultati sono stati raggiunti e possibili sviluppi futuri.

Bibliography

[1] L. DE MARCHI, Multi-core scheduling optimizations for soft real-time multi-threaded applications. A cooperation aware approach., 2009.

[2] J. M. CALANDRINO and J. H. ANDERSON, On the Design and Implementation of Cache-Aware Multicore Real-time Scheduler, 2009.

[3] A. FEDOROVA, M. SELTZER, and M. D. SMITH, Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler, 2007.

[4] D. MOLKA, D. HACKENBERG, and W. E. NAGEL, Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems, 2009.

[5] D. MOLKA, D. HACKENBERG, R. SCHONE, and M. MULLER, Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor, 2009.

[6] T.-F. YANG, C.-H. LIN, and C.-L. YANG, Cache-Aware Task Scheduling on Multi-Core Architecture, 2010.

[7] N. GUANN, M. STIGGE, and W. YI, Cache-Aware Scheduling and Analysis for Multicores, 2009.

[8] R. CHANDRA and A. GUPTA, Data Locality and Load Balancing in COOL, 1993.

[9] L. PENG, J.-K. PEIR, and T. PRAKASH, Memory hierarchy performance measurement of commercial dual-core desktop processors, 2008.

[10] D. CHANDRA, F. GUO, and S. KIM, Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture, 2005.

[11] B.-F. QIAN and L.-M. YAN, The Research of the Inclusive Cache used in Multi-Core Processor, 2008.

[12] T. ROLF, Cache Organization and Memory Management of the Intel Nehalem Computer Architecture, 2009.

[13] J. H. ANDERSON and J. M. CALANDRINO, Parallel Real-Time Task Scheduling on Multicore Platforms, 2006.

[14] A. FEDOROVA, M. SELTZER, and M. D. SMITH, Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler, 2007.

[15] V. BABKA and P. TUMA, Investigating Cache Parameters of x86 Family Processors, 2009.

[16] White Paper Intel next generation microarchitecture Nehalem.

[17] Intel Corporation. Intel 64 and IA-32 architectures software developer's manuals.