A Step Toward Exploiting Task-Affinity in Multi-Core Architectures to Improve Determinism of Real-Time Streaming Applications

Lucas Martins De Marchi
Polytechnic School
University of São Paulo
São Paulo, Brazil
lucas.de.marchi@gmail.com

Patrick Bellasi, William Fornaciari
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
{bellasi,fornacia}@elet.polimi.it

Betz Wolfgang
Advanced Systems Technology
STMicroelectronics
Agrate Brianza, Italy
wolfgang.betz@st.com

***Abstract*—Cooperative applications represent a class of multi-tasking programs where different tasks execute concurrently according to a producer-consumer pattern. This class of programs is increasingly adopted on multi-core architectures and especially on multimedia mobile devices based on MPSoCs. Indeed, it allows to better exploit the architectural parallelism. However, the run-time efficient usage of memory hierarchies still is mandatory to achieve really good performance. Moreover, when responsiveness and predictability of the application are required, obtaining strict real-time behaviors on these architectures is still an interesting research topic.**

This work proposes a software mechanism to enhance soft real-time behaviours of cooperative applications. Targeting the Linux kernel, this mechanism enhances its real-time scheduler by introducing the support for cache-aware scheduling. On some architectures, the experiments conducted on a synthetic benchmark allow to observe significant improvements, both on execution predictability and data throughput. Further improvements already under investigation are foreseen to extend the benefits to more architectures.

***Keywords*-Real-Time Scheduling, SMP, Cache-Affinity, Linux, Determinism, Throughput**

I. Introduction

During the last decade, the technology scaling has enforced strict constraints on the maximum operative frequency of hardware components. Thus the demand for more processing power has been satisfied by increasing parallelism at different levels. At the system level this led to the introduction of multi-core architectures [1]. Nowadays, Symmetric Multi Processors (SMP) are widely used on high-end systems and are going to be adopted for mobile multimedia systems as well.

To take advantage of the new multi-core architectures, software needs to be properly written [2]. Actually, there exist many ways to transform an application to exploit different levels of parallelism [3]. Although the classification of parallelism levels is quite a broad area, in a first instance and focusing just on the source code, we can identify two main classes: Pipeline-Level Parallelism (PLP) and Task-Level Parallelism (TLP). Each of them directly maps to a different programming model.
The TLP maps a *"competitive model"*, where multiple instances of a task execute in parallel, processing different data. Usually these tasks compete for the usage of shared resources, such as CPU, memory, communication channels, and peripherals. The PLP instead maps a *"cooperative model"*, where different tasks execute in sequence processing the same data. In this model tasks work by exchanging data according to a producer-consumer pattern.

A successfully decomposition of critical applications[1] into multiple tasks is required to not negatively impact on performance and soft real-time constraints. Thus, one of the main challenges is to identify a scheduling of tasks on the different Processing Elements (PE) that satisfy the contrasting requirements of *throughput* and *determinism*.
The scheduler available in modern versions of the Linux kernel has been designed with the main goal to be highly scalable and efficient in handling the execution of competitive tasks [4]. However, this scheduler is less effective on the management of cooperative applications, where a proper exploitation of the memory hierarchy cannot be negletted to better support the data communication among different tasks of a streaming application. Moreover, the scheduling techniques used for competitive tasks execution often lead to some undesirable side-effects, such as an overall loss of predictability [5], which is mandatory when dealing with real-time tasks.
Thus, the effective scheduling of cooperative tasks requires an improved efficiency on memory access. This allows to achieve not only better throughputs but also an improved determinism, thanks to a reduced number of cache-misses and corresponding penalties.

Within this paper we address the concept of cache-aware scheduling, and describe the implementation of a simple while effective *task-affinity* mechanism designed to boost the Linux kernel. The mechanism has been experimentally evaluated on two different real-world micro-architectures, both belonging to the X86 family but using a different memory hierarchy. On both systems we revealed a negligible impact

[1]We refers mainly to those related to multimedia processing, e.g. audio/video codecs, network protocols processing, etc.

on the scheduling overhead, while we obtained divergent results in terms of improvements of the real-time behavior. In one case we was able to observed at least one order of magnitude of improvement in the predictability of the execution latencies for the synthetic benchmark we adopted throughout the paper.

The rest of this paper is organized as follows. In Sec. II we introduce some background on real-time scheduling and memory effects on its behavior. A short review of the prior-art is presented in Sec. II. Sec. IV describes the proposed task-affinity mechanism. The experimental setup is defined in Sec. V; where we introduce also the synthetic benchmark used for the measurements and corresponding results are commented in Sec. VI, Finally, in Sec. VII we draw some conclusions and outline future work.

II. BACKGROUND

The Linux scheduler [4] is based on a modular architecture designed to satisfy different requirements. Its *scalability* on multi-core architectures and *fairness* on multi-task scheduling have been perhaps the two most important goals. Moreover, it provides usable support for soft real-time tasks, which is also getting more and more attention in the last years. The present scheduler architecture is based on three different *scheduling classes*: `RT` for real time tasks, `FAIR` for best-effort tasks and `IDLE` just for idle CPU power optimization. Each scheduling class is essentially an algorithm to schedule tasks belonging to that class.

The RT class provides a soft real-time support according to two different policies: either in run-to-completion mode using the `FIFO` policy or in time-slotted mode using the round-robin `RR` policy. Within the current Linux kernel, real-time is considered just as the requirement to "getting started as soon a possible". This is the purpose of the `RT` scheduling class and especially of all the greatly advances recently provided by the "-rt patchset" [6].
The other kernel trend is to be also real-fast, that means "getting done quickly once started". This requirement is mostly related to the improvement of the system throughput.

Scheduling a task for execution requires some activities that introduce latency. Moreover, the actual execution time a certain task takes generally varies from one run to another. When we focus on real-time scheduling, which is sensible about determinism [5], these aspects cannot be disregarded. Specifically, it becomes important to precisely identify two metrics: the *startup latency*, i.e., the maximum latency required to start a task that is ready to run, and the *execution time*, i.e. the actual execution time it will take once started.

These two metrics are commonly used to compare the behavior of different scheduling algorithms and evaluate the determinism needed by real-time tasks. A well established *RT metric* (ρ) for the evaluation of the real-time Linux scheduler is:

$$\rho = \mu + 2 * \sigma \tag{1}$$

where μ is an average value (e.g., for the completion time of a task) and σ the corresponding variance. This formula is particularly interesting because it allows to consider both throughput and determinism. Indeed, the lower the average the better the throughput and the lower the variance the better the determinism. The value μ and σ could be affected by several sources of uncertainty, which depend both on the very nature of computer architectures and how the operating system running on top of them is implemented.

On SMP systems, where all the CPUs have the same memory space but generally with very different access times, the choice of the processing element in which to put a task could have a direct impact both on the cache miss rate and miss penalty, and thus on the task execution time. Indeed, a very simple characterization of the average memory access time can be expressed as [7]:

$$Avg(T_{mem\ access}) = T_{hit} + R_{miss} \cdot P_{miss}$$

where T_{hit} is the hit time, i.e., the time needed to retrieve data from cache, and P_{miss} is the miss penalty, i.e., the time needed to wait when data is not available in cache. Even such a simple charaterization shows that the mean time of memory access is defined both by the memory speed and the cache-miss rate. Moreover, if the cache-miss rate increases, the mean time of memory access increases as well. Accordingly, if we have more cache misses in a certain task, it will take longer to produce its results because the memory access will be slower.

The *wakeup locality*, i.e. scheduling a task on a certain processor, is certainly among the main factors impacting both system thgoughput and determinism. Improving this factor requires to investigate in two direction: the spatial and temporal locality of data.
The *spatial locality* addresses the reduction of the number of task migrations. Tasks could migrate from one processor to another, because of a) the way the application is designed (e.g., the locking mechanisms it uses) and b) the behavior of the Linux scheduler (e.g., when a real-time task wakes up on a busy CPU). Both of these causes could produce situations in which a task is moved off the processor that has a hot cache for it. Improving spatial locality requires to select the proper CPU in which to run a task, avoiding unnecessary and harmful migrations.
The *temporal locality* aims at reducing the cache miss-rate. Spatial locality by itself is not sufficient to grant reduced miss-rates. Indeed, as well as choosing the right CPU with hot cache data, a task must run at the proper time before its cache becomes dirty. Thus, addressing temporal locality is an extension feature of the spatial locality in order to improve the scheduler capability to better exploit the memory hierarchy.

III. PRIOR ART

The importance of cache-aware scheduling is not a recent discovery. Already in the early '90 theoretical models were developed describing the problem [8]. To the best of our knowledge, one of the first and most comprehensive works related to scheduling for data locality is the COOL project [9], which is dated back to the 1993. This work proposed an extension to the C++ programming language which allows the programmer to feed information both to the compiler and to the run-time task scheduler. Unfortunately, the proposed extension has not become a standard feature of the language and the run-time was not based on the Linux operating system. However, this work introduced a number of interesting concepts which are at the base of our study.

With the advent and broad diffusion of SMP architecture, the problem got a new focus especially to target the optimization of modern operating systems. They started to exploit cache affinity for each thread in order to bias the SMP scheduler so that each thread keeps running on the same CPU [10]. However, to the best of our knowledges the support for multi-threading cache affinity is almost unexplored on real-world OS and Linux especially.

More recently, a number of works in literature try to improve the exploitation of the memory hierarchy to better support throughput-oriented scheduling [11], or the enforcement of other QoS requirements [12]. A deeper analysis of unfair and unpredictable performance of concurrently executing applications caused by resource sharing, with last-level cache being one of the most impacting resource, has been proposed in [13]. A cache optimization technique to reduce instruction and data cache misses for streaming applications is presented in [14]. This technique is based on multiple execution of each task before moving to the next one. Moreover, a dataflow model is used to trade-off the number of cache misses against end-to-end latency and memory usage. Among the most recent works, in [15] is discussed the implementation of an efficient cache-aware soft real-time scheduler within Linux which exploit an interesting automatic cache behavior profiling. Unfortunately, this work is based on an EDF scheduling policy which is not available on mainline kernel and cannot be exploited by non-profiled tasks.

Considering all these previous research contributions, the aim of this work is mainly to explore the possibility to port some of the proposed concepts within a modern operating system such as Linux. To the best of our knowledge, this is the first work that evaluates a Linux cache-affinity aware scheduler targeting soft real-time cooperative applications.

IV. TASK-AFFINITY SCHEDULER

This work focuses on the optimization of scheduling for real-time tasks. We target multi-tasking applications that exploit both TLP and PLP. These applications are composed by a set of tasks, some processing data in a pipeline and others

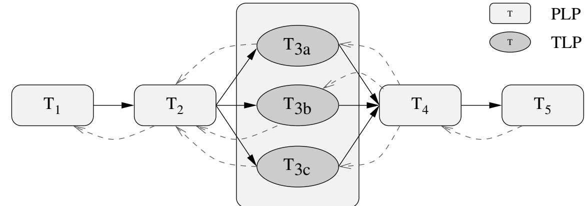

Figure 1: A mixed workload multi-tasking application.

processing them in parallel. The pipelined tasks implement a producer-consumer paradigm where data is exchanged using shared buffers. Thus, the performances of these tasks are especially influenced by an efficient exploitation of the memory hierarchy. All the tasks are assumed to run with real-time priority.
In this work we propose an optimization of the Linux scheduler which constraints the selection of the CPU on which to run a task. The objective here is to improve the data spatial locality of shared buffers and thus to better exploit the memory hierarchy. To this purpose we introduce the concept of *task-affinity*, which essentially allows to *keep track of data dependencies* between any two cooperative tasks.

Two tasks are affine to each other if they share some data and their execution depends upon reading or writing this data. The simplest case of affinity is the one with only one consumer and one producer. The consumer depends on data generated by the producer since it needs this data in order to be able to run. We represent task-affinity using an *affinity graph* where tasks are nodes and arrows represent a data dependency. An example scenario is represented in Fig. 1 where we provide the affinity graph (dashed arrows) of a mixed workload parallel application which exibits both TLP and PLP (black arrows represent the data-flow).

In order to improve memory accesses, affine tasks have to be executed by CPUs sharing the lowest possible level of cache. At the same time, we must also pay attention to not lower the application parallelism, since scheduling on near CPUs could imply a delayed execution when the selected processing element is already occupied. Thus, the aim of the affinity graph is to provide the Linux scheduler with the minimal information required to enforce the execution on near processors of tasks having a data dependency, whenever this operation could be convenient and considering other run-time imformation.

To support task-affinity we *a)* keep track of data dependencies and *b)* update the scheduler in order to exploit this data. The first modification is quite trivial and requires just to extend the data structures used to represent a task within the kernel, i.e. `struct task_struct`. Much more critical are the modifications to the scheduler. The main issue here is to introduce the new functionality without impacting too much on scheduling overheads. Essentially, this has required

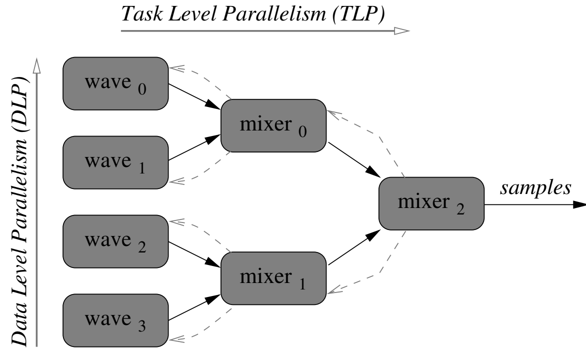

Figure 2: The structure of the synthetic benchmark used for measurements.

to update the function responsible to select the CPU on which a task will run. Our implementeation adopt a simple heuristic to extend the existing selection policy, which uses a *CPU mask* to define the sub-set of CPUs eligible for execution. The CPU mask is updated according to the task-affinity information and the possibility to find a hot cache. Among the allowed CPUs, it is chosen the one with less real-time tasks. This is a quite simple heuristic that aims at improving the temporal locality.

In order to have the task-affinity mechanism working on Linux, it is necessary to turn off other migration mechanisms for tasks providing affinity information. Indeed, all the three ways to migrate tasks: pull, push and the periodic scheduler may break task dependency. Finally, we also added a mechanism to inform the kernel that a multi-tasking application uses cooperative tasks and to set-up its affinity information. For the sake of simplicity, right now we just provide a simple syscall interface, which allows to test and validate the effectiveness of the scheduling policy and it is foreseen to be replaced in future by a more smart mechanism. The patchset produced is quite self-contained and desinged to introduce small run-time overheads that has been experimentally evaluated as presented in the following section.

V. Experimental Setup

Our evaluation strategy adopts a synthetic benchmark specifically designed to evaluate the ability of Linux to replace dedicated hardware blocks by multi-tasking applications. The structure of the considered multi-tasking application is depicted in Fig. 2. This chain represents a software mixer: `waves` are tasks producing some data packets while `mixers` are tasks that mix these data packets and produce new ones to be forwarded to the next level. At the end of this chain a single data packet is produced, which is named a `sample` and is associated to its *production time*. The benchmark has been implemented for Linux using the standard POSIX API for multi-threaded applications. Data packets exchange between tasks make use of buffers visible to both tasks. Each buffer accommodates just a single data packet. Besides communication, the synchronization among the tasks plays an important role. To this purpose, mutexes and semaphores are used to protect buffer hand-overs and producer/consumer synchronization.

We defined *production rate (PR)* to be the reverse of the time between two consecutive samples, i.e. $PR = 1/(t_{n+1} - t_n)$, where t_i is the production time of the $i - th$ sample. The main goals of such a chain are to increase the PR whithout incurring on increased variance. In other words, we want both *high throughput*, defined by the average of the production time of all the samples, and *determinism*, defined by the standard deviation of the production time distribution. Apart from the conflicting requirements, the proposed benchmark is particularly interesting because it allows to consider a mixed workload with both PLP and TLP. Thus, only tasks with an affinity can take advantage of the proposed mechanism, while competitive tasks should continue to be scheduled in a way to best exploit their level of parallelism. Moreover, the benchmark has not been run in isolation but concurrently to the set of tasks normally running on a standard Linux distribution.

We considered two quad-core systems to evaluate our proposal on different micro-architectures and memory hierarchies. The `machine A` uses an Intel Core i7 920 processor running at 2.67GHz. The memory hierarchy provides a unified 8MB L3 cache, shared by all cores, while each core has a private unified 256KB L2 cache and two 32KB L1 caches for instructions and data respectively. The `machine B` is based on an Intel XEON E5440 processor clocked at 2.82GHz. The memory hierarchy of this machine does not provide an L3 cache but only two banks of L2 cache, each one with 6MB and shared between pairs of cores. The L1 caches instead have the same characteristics as those of machine A. We exploited the Linux support to off-line some cores of an SMP system in order to test the proposed implementation with 2 or 4 cores enabled.

The proposed task-affinity mechanism has been implemented on top of the vanilla Linux kernel version 2.6.31. Several configuration settings have been considered in order to ensure reliable measurements. In particular, we disabled the support for frequency scaling and real-time group scheduling. We used *ftrace* and its `sched_switch` plug-in to efficiently collect a trace of scheduler events with related timings. Finally, cache measurements have been obtained using the “perf” tool to read the hardware event counters provided by the target hardware architectures. Every test executed the synthetic benchmark to generate 150000 samples.

VI. Measurements and Results

Two types of tests have been made. The first is used to prove that the mechanism of task-affinity is working as expected. The second type serves to compare the behavior

```
  wave0: 302121331102303   111111111111111
  wave1: 223303013333212   222222222222222
 mixer0: 322210301110320   111111111111111
  wave2: 033332232302003   333333333333333
  wave3: 110211120211101   000000000000000
 mixer1: 001003321202220   030303030303030
 mixer2: 230101020201202   303030303030303
```

(a) (b)

Figure 3: Migrations map for (a) the vanilla Linux kernel and (b) the same kernel with our patch for task-affinity scheduling.

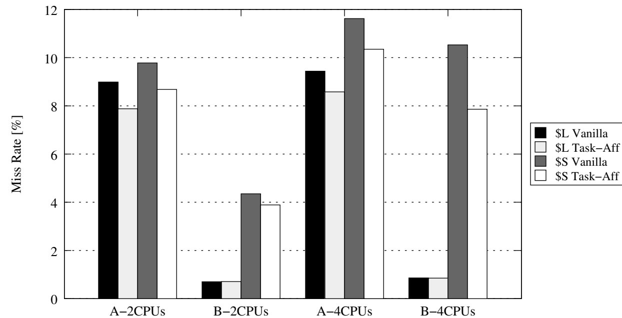

Figure 4: Cache-miss rate of load and store instructions for vanilla and task-affinity on Machine A (i7) and B (Xeon).

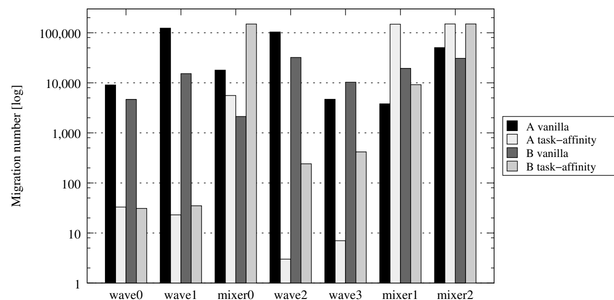

Figure 5: The number of migrations per task (4 core configuration).

of the kernel with an without the changes proposed in this work.

Being independent of the specific architecture, the functional verification has been done using a *trace* of the generated scheduler events. Each event allows to know when a task has been woken-up, which run-queue has been selected and how long it had to wait on that run-queue before actually running. The goal of this analysis is to verify that the task-affinity information is correctly handled by the modified scheduler. A portion of the migrations map, obtained running the test benchmark with a 4 CPU configuration, is represented in Fig. 3. For each task, the *migrations map* shows on which CPUs a task was running. The complete map has a column for each sample, but for verification purposes we reported only a portion. These maps shows that in the vanilla Linux kernel tasks are continuously moving among the available processors, while using the task-affinity scheduler they tend to be pinned on the same CPU. Moreover, with task-affinity each task having an affinity always runs on one of the CPUs where its producers have run previously, e.g., `mixer0` always runs on CPU1 where `wave0` was running before.

Besides a functional verification of the proposed task-affinity scheduler implementation we are mainly interested in analyzing: a) the effectiveness of the memory hierarchy exploitation; b) the migration behavior – which should be less aggressive than in mainline; and c) the system performance in terms of throughput and determinism. In order to evaluate the generalization of the proposed approach, we considered different architectures.

Memory hierarchy exploitation: We measured the *cache miss rate* as the ratio between the total number of cache access and the number of misses, considering both store and load operations. Ten benchmark runs have been considered, using the “perf” tool to collect measurements and to compute the average values for both the vanilla Linux kernel and our task-affinity implementation. These values are represented in Fig. 4. These graphs show that the task-affinity scheduler always reduces the overall number of cache misses. In case of Xeon architecture (machine B), with four enabled cores, the miss rate is improved (i.e., it is reduced) by about 1% both for store and load. This correspond to an relative improvement with respect to the mainline kernel by about 11% and 9.1% respectively. Instead, for the i7 architecture (machine A) only the store cache rate is improved by as much as 2% in the 4 enabled cores; which correspond to a 20% relative improvement with respect to the mainline kernel.

The average number of migrations for each task in the 4 core configuration is represented in Fig. 5. The plot shows that the number of migrations is significantly reduced for producer tasks (i.e., waves), while in the case of consumers it can be even much higher (note the logarithmic scale). It is worth remarking that using the task-affinity scheduler the overall number of migrations is in general higher. In our experiments we counted up to 70% more migrations. This is due to the production of migration patterns, like the one between `mixer1` and `mixer2` represented in Fig. 3, that generally originate when the number of real-time tasks ready to run is higher then the number of available cores.

The reduction of cache misses should correspond to a better exploitation of the memory hierarchy. This hypothesis is confirmed if the time each task takes to complete a single sample is diminished. To support this analysis we can refer to the measurements in Fig. 6, which shows the average time each task takes to complete its work for a single sample and the corresponding variance. These values have been computed by running the benchmark to generate 150,000

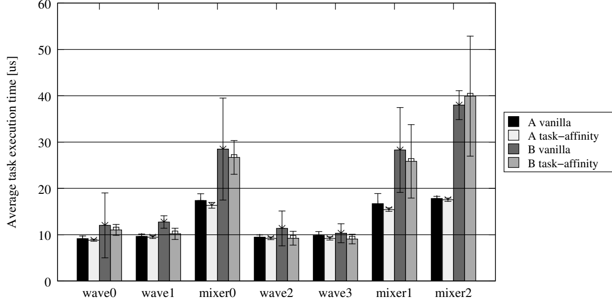

Figure 6: The average execution time of each task and its variance (4 core configuration).

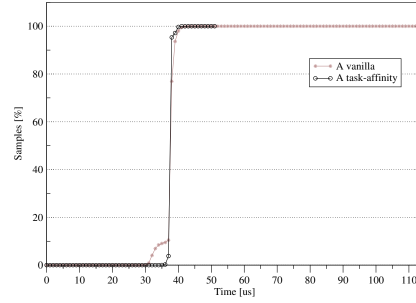

Figure 7: The empiric repartition function for the distribution of samples completion time (i7, 4 core configuration).

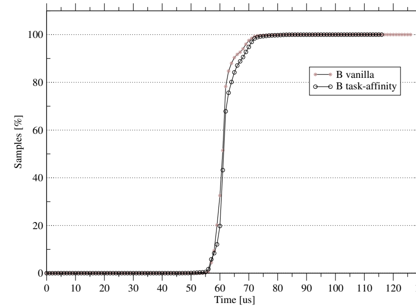

Figure 8: The empiric repartition function for the distribution of samples completion time (Xeon, 4 core configuration).

samples on each platform with all four CPUs enabled.

As reported by the plot, when running on i7 (machine A) the task affinity allows for a better exploitation of the memory hierarchy. Indeed, we can observe the reduction of both average execution time and corresponding variance for all the real-time tasks.
On the contrary instead, when running on Xeon (machine B) the average execution time and the variance are almost always improved except for some mixer tasks. We are still investigating this behavior, which seems to be primarily related by the specific memory hierarchy of the XEON machine and the limited support for temporal locality enforcement in the current task-affinity implementation.

The overall system performance is defined in terms of throughput and determinism. This requires to evaluate not only the average sample production time but also its determinism. For this purpose Tab. I summarizes the results obtained using our task-affinity approach also with respect to the relevant metrics for determinism. We can notice, that, although the overall average of sample production time is always slightly lower in the vanilla Linux kernel, with the task-affinity scheduler the production times of all samples are much closer to the average value in the i7 architecture (machine A). In this architecture, the net effect of the proposed task-affinity implementation is a good improvement in the throughput (the overall benchmark execution time is reduced by a 5.94%) and a much more significant $x14$ improvement in determinism of the sample production time. Unfortunately, on the Xeon architecture (machine B), both throughput and determinism are negatively affected by the poor performances of some consumer tasks.

To better assess these results, we represented the empiric repartition function of the sample production times for both versions of the kernel and both machines used. Fig. 7 is related to i7 (machine A) while Fig. 8 refers to Xeon (machine B). For the first architecture, this graph shows that when using the task-affinity scheduler, most of the samples are much closer to the real-time metrics value, which itself has an improved value with respect to the value obtained when running the benchmark on the vanilla Linux kernel.

VII. Concluding Remarks

In order to improve the memory hierarchy exploitation when scheduling soft real-time cooperative applications, we extended the mainline Linux scheduler with an implementation of the task-affinity concept. This extension allows to keep track of data dependencies between any two cooperative tasks in order to bais the real-time scheduler CPU selection policy.

The proposed mechanism has been implemented to be smoothly integrated into the Linux mainline kernel and evaluated considering two micro-architectures with different memory hierarchies. This mechanism allows us to better

Table I: Performances improvement using task affinity

	Average	**Variance**	**RT metrics**	**Speedup**
A vanilla	37.826	3.225	41.420	
A task-affinity	38.038	0.214	38.960	*5.94%*
A improvement	*-0.01%*	*1400.7%*		
B vanilla	62.22	10.42	83.07	
B task-affinity	63.27	19.76	102.8	*-23.76%*
B improvement	*-1.69%*	*-89.61%*		

exploit the task data locality and its evaluation has shown that it improves not only the determinism but also the system throughput on architectures with a deep memory hierarchy.

We are now targeting temporal locality to improve exploitation of hot-caches. Even more important we are moving towards the use of a more extensive benchmark set in order to better asses the results obtained so fare and possibly to address forther refinements.

REFERENCES

[1] G. Blake, R. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, November 2009.

[2] H. Kim and R. Bond, "Multicore software technologies," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 80–89, November 2009.

[3] M. D. McCool, "Scalable Programming Models for Massively Multicore Processors," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 816–831, May 2008.

[4] D. Bovet and M. Cesati, "Understanding The Linux Kernel," 2005.

[5] R. F. Berry, P. E. McKenney, and F. N. Parr, "Responsive systems: an introduction," *IBM Systems Journal*, vol. 47, no. 2, 2008.

[6] P. E. Mckenney, "SMP and Embedded Real Time," *Linux Journal*, no. 153, 2007.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.

[8] M. Squillante and E. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 131–143, 1993.

[9] R. Chandra, A. Gupta, and J. L. Hennessy, "Data locality and load balancing in COOL," *ACM SIGPLAN Notices*, vol. 28, no. 7, p. 249, 1993.

[10] B. J. Aas, "Understanding the Linux 2.6.8.1 CPU Scheduler," *Silicon Graphics International SGI*, vol. 22, no. February, pp. 1–26, 2005.

[11] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Throughput-oriented scheduling on chip multithreading systems," 2004.

[12] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten, "Modeling Cache Sharing on Chip Multiprocessor Architectures," *2006 IEEE International Symposium on Workload Characterization*, pp. 160–171, October 2006.

[13] X. Zhou, W. Chen, and W. Zheng, "Cache Sharing Management for Performance Fairness in Chip Multiprocessors," *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 384–393, September 2009.

[14] A. Moonen, M. Bekooij, R. van Den Berg, and J. van Meerbergen, "Cache Aware Mapping of Streaming Applications on a Multiprocessor System-on-Chip," *2008 Design, Automation and Test in Europe*, pp. 300–305, March 2008.

[15] J. M. Calandrino and J. H. Anderson, "On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler," *Euromicro Conference on Real-Time Systems*, 2009.