# Overview of Multicore Requirements towards Real-Time Communication

Ina Podolski and Achim Rettberg

Carl von Ossietzky University Oldenburg
ina.podolski@iess.org, achim.rettberg@iess.org

**Abstract.** For embedded systems multicores are becoming more important. The flexibility of multicores is the main reason for this increasing extension. Considering embedded systems are often applied for real-time task, the usage of multicores implicates several problems. This is caused by the architecture of multicore chips. Usually such chips consist of 2 or more cores, a communication bus, I/O's and memory. Exactly the accesses to these resources from a core make it hard to ensure real-time. Therefore, well known mechanisms for resource access must be used, but for sure this is not sufficient enough. Because, a lot of design decisions depend on the applications. As a result, it is mandatory to analyze the requirements of the applications in detail and from the targeted multicore system. The aim of this analysis process is to derive a method for an optimal system design with respect to real-time support. This paper gives an overview of the requirement analysis for multicores and RT scheduling algorithms. Additionally, existing scheduling strategies are reviewed and proposals for new schedulers will be made.

## 1 Introduction

The trend of applying multicore systems in many embedded application the necessity of real-time for such systems becoming more and more important. Typical multicores are shown in figure 1.

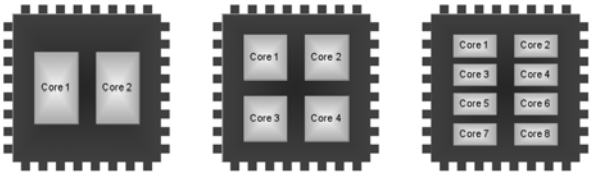


**Fig. 1.** Three exemplary types of multicore architectures

The authors of [1] argue that the transactional memory concept within multicore systems has attracted much interest from both academy [3] [4] and industry [5] as it eases programming and avoids the problems of lock-based methods. Furthermore they discuss, by supporting the ACI (Atomicity, Consistency and Isolation) properties of

S. Lee and P. Narasimhan (Eds.): SEUS 2009, LNCS 5860, pp. 354–364, 2009.

transactions, transactional memory relieves the programmer from dealing with locks to access resources, see [1]. Besides the locking problems it is necessary to find solutions for deadlock avoidance and take care on the priority inversion problem. As surely argued in [1], in the case of multicore systems, lock-based synchronization can reduce the data bandwidth by blocking several processes that try to access critical sections, thus reducing processors utilization. The resource access is one of the major problems. Shared resources are controlled by lock-based methods with the well-known disadvantages of serial access. Parallel access can be realized with transactional memory, see also [1]. That means, a transaction is either aborted when a conflict is detected, or committed in case of successful completion.

An exemplary mapping of functional modules to tasks is depicted in figure 2. The tasks itself are mapped to cores of a multicore architecture. Some tasks require operating systems with a scheduler.

As we can see from literature and the authors of [1] argue, real-time scheduling of transactions, which is needed for many real-time applications, is an open problem in multicore systems. The idea is to look at existing solutions for real-time scheduling of tasks in multiprocessor systems or transaction in systems are not suitable for multicore systems. The literature shows, real-time scheduling of tasks in multiprocessor systems does not consider important features of multicore systems, such that the presence of on-chip shared caches, see [1]. Caches are a big problem. If each core has is own cache the problems is the shared access from the caches to the main memory. In case of shared caches again locking mechanism have been used with all it's disadvantages. Again as argued in [1] real-time scheduling of transactions in systems has been around since the 80s but assuming either centralized or distributed systems, but both solutions are not suitable for multicore systems as well.

In this paper, we give an overview of related work of existing scheduling methods with resource access for multicore systems. We will briefly present a simple resource access strategy to demonstrate the problems and will discuss the main challenges for real-time scheduling in multicore systems.

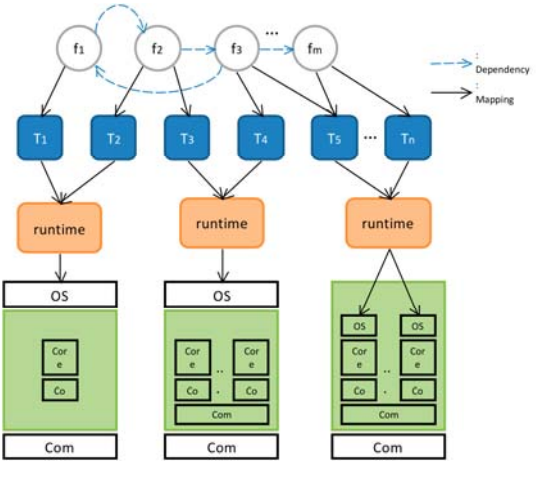


**Fig. 2.** Exemplary mapping of functions to tasks and multicores

## 2 Multiprocessor Scheduling Approaches with Resource Access Protocols

In this section we will demonstrate with a small example the problems of shared resource access for multicore systems. We make the following assumptions. For each task, a set of jobs is associated. At any time, each processor executes at most one job. The task has a period and an execution requirement.

When a job is released, it executes during the execution requirement of the task, and once the period is elapsed, another job of the task, is released.

On multiprocessors, EDF is not optimal either under the partitioned or the global approaches [10], called respectively P-EDF and G-EDF, see also [1]. There exist further classes of scheduling algorithms differs from the previous ones. A typical example is the Pfair algorithm [11]. Pfair based on the idea of proportionate fairness and ensures that each task is executed with uniform rate. All tasks are broken into so-called quantum-length subtasks and the time is subdivided into a sequence of subintervals of equal lengths called windows, see [1]. This means, a subtask has to be executed within the associated window. Additionally, migration is allowed for each subtask. As described in [1] an optimal Pfair variant is that from [12].

In figure 3 an example multicore architecture is shown with cores A and B. On core A two tasks $t_1$ and $t_2$ are executed and on core B $t_3$ and $t_4$ are running. Additionally, the tasks access resources $R_1$ and $R_2$. For example task $t_2$ has access to $R_1$.

As described in [1], the protocols managing resources in real-time systems are usually used in a hard real-time context, such as M-PCP and FMLP1 [13] under EDF. For Pfair scheduling, a lock-free algorithm has been proposed [12] to ensure that some task is always making progress. Indeed, classical lock-based algorithms cannot satisfy this property.

Obviously, it can be figured out that resource access conflicts can occur. If task $t_1$ on core A and task $t_4$ on core B run at the same time and try to access $R_2$ there is a resource conflict. Those conflicts exist due to the parallelism implicated by multicore architectures.

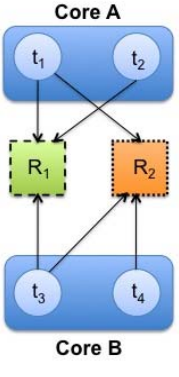


**Fig. 3.** Example multicore architecture with 2 cores A and B, 2 resources $R_1$ and $R_2$, and tasks $t_1$ to $t_4$

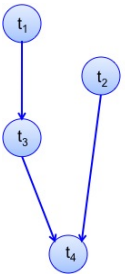


**Fig. 4.** Task graph for $t_1$ to $t_4$

**Table 1.** Task characteristics

| | $c_i$ | $d_i$ | $a_i$ |
|---|---|---|---|
| $t_1$ | 6 | 9 | 0 |
| $t_2$ | 6 | 16 | 0.5 |
| $t_3$ | 8 | 20 | 0 |
| $t_4$ | 5 | 22 | 9 |

Figure 4 shows a task graph consisting of four tasks $t_1$ to $t_4$. Data dependencies between the tasks are represented by edges. Arrival time $a_i$, computation time $c_i$ and deadline $d_i$ for the tasks are given in table 1.

On uni-core architectures there exist different resource access protocols. The most used one is the priority ceiling protocol (PCP) see [9]. The advantages of PCP are to prevent chained blocking and deadlocks. Each resource has a semaphore. A semaphore $s_k$ is assigned a priority ceiling $C(s_k)$ equal to the priority of the highest-priority job that can lock it. Note that $C(s_k)$ is a static value that can be computed off-line. Let $t_i$ be a task with the highest priority among all tasks ready to run; thus, $t_i$ is assigned the processor. Furthermore, let s* be the semaphore with the highest ceiling among all the semaphores currently locked by tasks other than $t_i$ and let C(s*) be its ceiling.

To enter a critical section guarded by a semaphore $s_k$, task $t_i$ must have a priority higher than C(s*). If the priority $P_i$ of task $t_i$ is greater equal C(s*), the lock on $s_k$ is denied and $t_i$ is said to be blocked on semaphore s* by the job that holds the lock on s*.

When a task $t_i$ is blocked on a semaphore, it transmits its priority to the task, say $t_k$, that holds that semaphore. Hence, $t_k$ resumes and executes the rest of its critical section with the priority of $t_i$. Task $t_k$ is said to inherit the priority of $t_i$. In general, a task inherits the highest priority of the jobs blocked by it.

When $t_k$ exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on the semaphore is awakened. Moreover, the active priority of $t_k$ is updated as follows: if no other jobs are blocked by $t_k$, $p_k$ is set to the nominal priority $P_k$; otherwise, it is set to the highest priority of the jobs blocked by $t_k$ .

The schedule with PCP protocol for an uni-core processor is shown in figure 5. On a uni-core 26 time units are needed for the schedule.

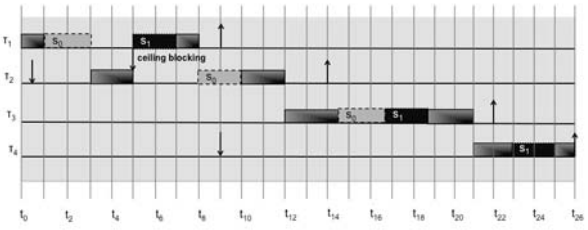


**Fig. 5.** Uni-core scheduling with PCP

The question is, if the PCP is adaptable for multicores? PCP solves the resource access within one core, but between cores is has to be modified, see the example in figure 6. There we show an optimal but unrealistic schedule. One conflict is the parallel access of to $t_1$ and $t_3$ on resource $r_2$. Therefore, it is necessary to adapt PCP to avoid conflicts caused by parallel access.

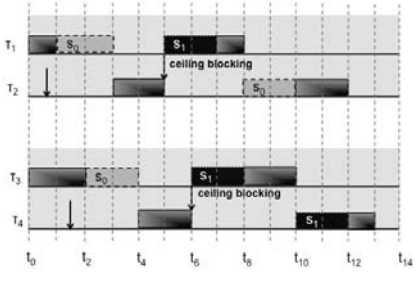


**Fig. 6.** Unrealistic schedule on a multicore with resource conflicts

A solution could be a globally controlled semaphore queue. With this queue we are able to avoid parallel access to resources. Lets say a task $t_2$ on core A access resource $r_1$, the semaphore $s_0$ for $r_1$ is included in the queue. Other tasks trying to access $s_0$ have to check within the queue if the global semaphore is not set within the global queue. Figure 7 shows the PCP solution with the global queue. The ceiling blocking is now visible between the cores, see the resource access from $t_2$ and $t_3$ on $s_0$.

In figure 7 we need 20 times units for the schedule in comparison to the 26 for the uni-core schedule. To reduce the long blocking we suggested another modification of PCP. Let us assume we have another resource $r_3$ inside the system that is used by $t_2$ on core A, see figure 8.

This is a sub-optimal solution, because now we have again a blocking time. This blocking time may be very long and lead to deadline misses. Another problem is the reduction of concurrency within the system. We see that core B has idle times caused by the blocking of $t_3$ by access to $s_0$.

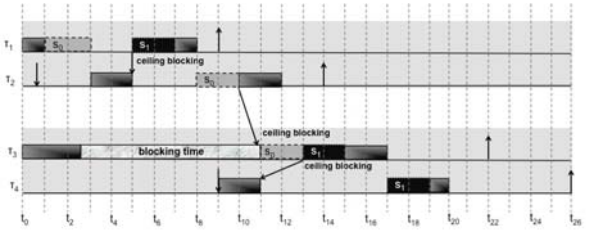


**Fig. 7.** Resource conflicts of PCP within multicore architecture

The main idea of this modification is the following. A resource is released whenever a task is finished with the usage. Furthermore, if a task is pre-empted by another task on the same core, the blocked resources of the pre-empted task are released. This enables tasks running on others cores to block the resources and starts with executing their critical sections. We have to ensure the fairness of our approach. First of all on each core we use PCP and for all tasks running on this core we have therefore a fair situation.

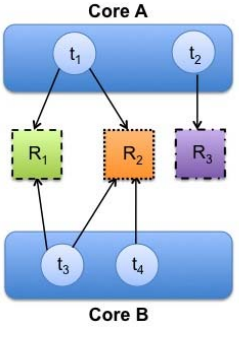


**Fig. 8.** Modified multicore example with additional resource $R_3$

The inter-core PCP respectively our modified version has to ensure real-time requirements on all cores. Let $t_i$ a task on core A that is pre-empted by a task with a higher priority. In this case the semaphores blocked by $t_i$ are released. Let's assume $t_i$ hold only one semaphore $s_k$. The time period $t_i$ is pre-empted is as long as the higher priority task is running, let's say this time-period is $b_i$. Semaphores are released only for the time-period $b_i$. A task $t_j$ on core B needed $s_k$ for time $b_j$ can now block the semaphore $s_k$, but only if $b_i \geq b_j$.

If $t_i$ and $t_j$ tries to access $s_k$ at the same time, the task with the earliest deadline will get the allowance to block semaphore $s_k$. Let $d_i$ the deadline for $t_i$ and $d_j$ for $t_j$. If $d_i < d_j$ task $t_i$ will have access to $s_k$ first. Now it is necessary to calculate the new deadline for $t_j$ as follows: $d_j = d_j + l_i$, whereas $l_i$ is the time $s_k$ is needed by $t_i$. This is a calculation for only one blocking. Figure 9 shows the schedule with the modified PCP.

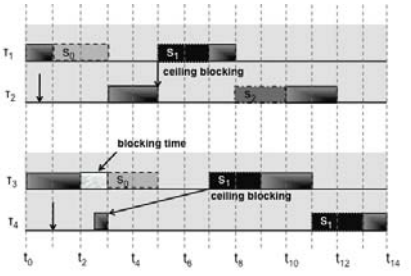


**Fig. 9.** Optimal schedule with the modified PCP

For evaluation the utilization factor for a schedule with our approach is as follows:

$$U = \sum_{i=0}^{n} \frac{C_i + B_i}{T_i}$$

Whereas $C_i$ is the computation time task $t_i$ and $T_i$ is the period of the task. The blocking time $B_i$ for all semaphores of the task is added to the $C_i$. Additionally we have to update the deadline of $t_i$, because due to the blocking. That means, if the $t_i$ is not blocked $d_i$ is not modified, otherwise the new deadline is added by the sum of all blocking times of $t_i$.

$$d_i = \{d_i, d_i + \sum b_k\}$$

This short example demonstrates the shared resource access problem. It is well-known that for uni-processor systems based on static or dynamic priority assignment of task Earliest Deadline First (EDF) is optimal [9].

## 3 State-of-the-Art

In this chapter we will give a short overview of the related work in this research field and enriches the discussion started in [1].

The authors of [2] claims also that the shared memory bus becomes a major performance bottleneck for many numerical applications on multicore chips, understanding how the increased parallelism on chip strains the memory bandwidth and hence affects the efficiency of parallel codes becomes a critical issue. They introduce the notion of memory access intensity to facilitate quantitative analysis of program's memory behavior on multicores, which employ stateof-the-art prefetching hardware.

The paper in [10] deals with the scalability of the scheduling algorithms presented above, on multicore platforms. One main conclusion of the authors is that on multicore platforms bandwith have negative impact on the algorithms, allowing migrations. The global approach, the scheduling overheads greatly depend on the way of implementing the run queues. On the other hand, without resource sharing, P-EDF performs well for this study, see also [1].

In [6] the authors present cache-efficient chip multiprocessor (CMP) algorithms with good speed-up for some widely used dynamic programming algorithms. They

consider three types of caching systems for CMPs: D-CMP with a private cache for each core, S-CMP with a single cache shared by all cores, and multicore, which has private L1 caches and a shared L2 cache. Furthermore they derive results for three classes of problems: local dependency dynamic programming (LDDP), Gaussian Elimination Paradigm (GEP), and parenthesis problem.

For each class of these problems, they propose a generic CMP algorithm with an associated tiling sequence.

A fundamentally new approach to increase the timing predictability of multicore architectures aimed at task migration in embedded environments is described in [7]. A task migration between two cores imposes cache warm-up overheads on the migration target, which can lead to miss deadlines for tight real-time schedules. The authors propose novel micro-architectural support to migrate cache lines. The developed scheme shows dramatically increased predictability in the presence of cross-core migration.

Another scheduling method for real-time systems implemented on multicore platforms that encourages certain groups of tasks to be scheduled together while ensuring realtime constraints is proposed in [8]. This method can be applied to encourage tasks that share a common working set to be executed in parallel, which makes more effective use of shared caches.

Another good overview of resource access protocols can be found in [14] and is as follows: “Rajkumar et al. [15] were the first to propose locking protocols for real-time multiprocessor systems. They presented two multiprocessor variants of the priority-ceiling protocol (PCP) [16] for systems where partitioned, static-priority scheduling is used. In later work, several protocols were presented for systems scheduled by P-EDF. The first such protocol was presented by Chen and Tripathi [17], but it is limited to periodic (not sporadic) task systems. In later work, Lopez et al. [18] and Gai et al. [19] presented protocols that remove such limitations, at the expense of imposing certain restrictions on critical sections (such as, in [19], requiring all global critical sections to be non-nested). A scheme for G-EDF that is also restricted was presented by Devi et al. [20]. More recently, Block et al. [21] presented the flexible multiprocessor locking protocol (FMLP), which does not restrict the kinds of critical sections that can be supported and can be used under either G-EDF or P-EDF. In the FMLP, resources are protected by either spin-based or suspension-based locks. The FMLP is the only scheme known to us that is capable of supporting arbitrary critical sections under G-EDF. Furthermore, the schemes in [20, 19, 18] are special cases of it. Thus, given our focus on G-EDF and P-EDF, it suffices to consider only the FMLP when considering lock-based synchronization.”

In [1] they discussed that pure global algorithms will not scale, and thus real-time global policies need to be revisited for many-core architectures. More particularly, the scheduler should be able to control more precisely the sharing of processor's internal resources (i.e. cache levels) by real-time tasks with on-chip shared caches, both the small size of the caches and the memory.

In [2] the overview of the memory access is as follows: “Memory bandwidth has been a fundamental issue for decades and has now become a major limitation on multicore systems ([22], [23], [24]). In S. Carr’s paper ([25]), methods are proposed to balance computation and memory accesses to reduce the memory and pipeline delays for sequential code on *uniprocessor* machines. The authors statically estimate the

ratio of memory operations and floating point operations for each loop and use them to guide loop transformations (e.g. unroll and jam). The methodology from [2], which is based on the new notion of the memory access intensity, targets parallel programs on multicore systems which employ sophisticated prefetching hardware.

Several existing papers investigate the scalability problem on multicore ([26], [27], [28]). They make observations that the memory bandwidth constraint can hamper program performance. However, no quantitative analyses are performed in those studies."

The authors of [29] present cache-efficient chip multiprocessor (CMP) algorithms with good speed-up for some widely used dynamic programming algorithms. They consider three types of caching systems for CMPs: D-CMP with a private cache for each core, S-CMP with a single cache shared by all cores, and Multicore, which has private L1 caches and a shared L2 cache. They derive results for three classes of problems: local dependency dynamic programming (LDDP), Gaussian Elimination Paradigm (GEP), and parenthesis problem. For each class of problems, they develop a generic CMP algorithm with an associated tiling sequence. They then tailor this tiling sequence to each caching model and provide a parallel schedule that results in a cache-efficient parallel execution up to the critical path length of the underlying dynamic programming algorithm.

## 4 Summary

Within this paper we start the discussion of real-time scheduling approaches and the resource access for multicore systems. Existing scheduling mechanisms have been adapted for multicore systems. Obviously new rules and policies have been required, which leads to new requirements depending on the features of the multicore architecture. With this position paper we want to give an overview based on the given literature.

## References

[1] Sarni, T., Queudet, A., Valduriez, P.: Real-time scheduling of transactions in multicore systems. In: Proc. of Workshop on Massively Multiprocessor and Multicore Computers (2009)

[2] Liu, L., Li, Z., Sameh, A.H.: Analyzing memory access intensity in parallel programs on multicore. In: Proceedings of the 22nd Annual international Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, pp. 359–367. ACM, New York (2008), http://doi.acm.org/10.1145/1375527.1375579

[3] Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. the 20th Annual International Symposium on Computer Architecture, May 1993, pp. 289–300 (1993)

[4] Shavit, N., Touitou, D.: Software transactional memory. In: Proc. the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 204–213 (1995)

[5] Tremblay, M., Chaudhry, S.: A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc r processor. In: IEEE International Solid-State Circuits Conference (February 2008)

[6] Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, Munich, Germany, June 14-16, pp. 207–216. ACM, New York (2008), http://doi.acm.org/10.1145/1378533.1378574

[7] Sarkar, A., Mueller, F., Ramaprasad, H., Mohan, S.: Push-assisted migration of real-time tasks in multi-core processors. SIGPLAN Not. 44(7), 80–89 (2009), http://doi.acm.org/10.1145/1543136.1542464

[8] Anderson, J.H., Calandrino, J.M.: Parallel task scheduling on multicore platforms. SIGBED Rev. 3(1), 1–6 (2006), http://doi.acm.org/10.1145/1279711.1279713

[9] Buttazzo, Giorgio, C.: Hard real time computing systems. Kluwer Academic Publishers, Dordrecht (2000)

[10] Calandrino, B.B.J., Anderson, J.: On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: Proc. The 29th IEEE Real-Time Systems Symposium (December 2008)

[11] Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15, 600–625 (1996)

[12] Leung, J.: Handbook of scheduling: algorithms, models, and performance analysis. Chapman & Hall/CRC, Boca Raton (2004)

[13] Brandenburg, B.B., Calandrino, J.M., Block, A., Leontyev, H., Anderson, J.H.: Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In: IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 342–353. IEEE Computer Society, Los Alamitos (2008)

[14] Brandenburg, B.B., Calandrino, J.M., Block, A., Leontyev, H., Anderson, J.H.: Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, April 22-24, pp. 342–353. IEEE Computer Society, Washington (2008), http://dx.doi.org/10.1109/RTAS.2008.27

[15] Rajkumar, R.: Synchronization in Real-Time Systems: A Priority Inheritance Approach. Kluwer Academic Publishers, Dordrecht (1991)

[16] Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: An approach to real-time system synchronization. IEEE Transactions on Computers 39(9), 1175–1185 (1990)

[17] Chen, C., Tripathi, S.: Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, Univ. of Maryland (1994)

[18] Lopez, J., Diaz, J., Garcia, D.: Utilization bounds for EDF scheduling on real-time multiprocessor systems. Real-Time Systems 28(1), 39–68 (2004)

[19] Gai, P., di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., Marceca, P.: A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In: Proceedings of the 9th IEEE Real-Time and Embedded Technology Application Symposium, pp. 189–198 (2003)

[20] Devi, U., Leontyev, H., Anderson, J.: Efficient synchronization under global EDF scheduling on multiprocessors. In: Proceedings of the 18th Euromicro Conference on Real-Time Systems, pp. 75–84 (2006)

[21] Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: A flexible real-time locking protocol for multiprocessors. In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 71–80 (2007)

[22] Smith, A.J.: Cache Memories. Computing Surveys 14(3), 473–530 (1982)

[23] Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2006-183 (December 18, 2006)
[24] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. (2007)
[25] Carr, S., Kennedy, K.: Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. ACM Transactions on Programming Languages and Systems 16, 1768–1810 (1994)
[26] Zhang, Q., et al.: Parallelization and Performance Analysis of Video Feature Extractions on Multi-Core Based Systems. In: Proceedings of International Conference on Parallel Processing, ICPP (2007)
[27] Alam, S.R., et al.: Characterization of Scientific Workloads on Systems with Multi-Core Processors. In: International Symposium on Workload Characterization (2006)
[28] Chai, L., et al.: Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In: Cluster Computing and the Grid (2007)
[29] Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, Munich, Germany, June 14-16, pp. 207–216. ACM, New York (2008), http://doi.acm.org/10.1145/1378533.1378574