Efficiently Migrating Real-Time Systems to Multi-Cores*

Farhang Nemati, Moris Behnam and Thomas Nolte
Mälardalen Real-Time Research Centre
Mälardalen University, Box 883, 72123, Sweden
{farhang.nemati, moris.behnam, thomas.nolte}@mdh.se

Abstract

Power consumption and thermal problems limit a further increase of speed in single-core processors. Multi-core architectures have therefore received significant interest. However, a shift to multi-core processors is a big challenge for developers of embedded real-time systems, especially considering existing "legacy" systems which have been developed with uniprocessor assumptions. These systems have been developed and maintained by many developers over many years, and cannot easily be replaced due to the huge development investments they represent. An important issue while migrating to multi-cores is how to distribute tasks among cores to increase performance offered by the multi-core platform. In this paper we propose a partitioning algorithm to efficiently distribute legacy system tasks along with newly developed ones onto different cores. The target of the partitioning is increasing system performance while ensuring correctness.

1. Introduction

Due to the problems with power consumption and related thermal problems, multi-core platforms seem to be the way towards increasing performance of processors. Multi-core is today the dominating technology for desktop computing.

The performance improvements of using multi-core processors depend on the nature of the applications as well as the implementation of the software. To take advantage of the concurrency offered by a multi-core architecture, appropriate algorithms have to be used to divide the software into tasks (threads) and distribute tasks fairly on cores to increase the performance. Real-time systems can highly benefit from the multi-core processors, as critical functionality can have dedicated cores and independent tasks can run concurrently to improve performance and thereby enable new functionality. Moreover, since the cores are located on the same chip and typically have shared memory, communication between cores is very fast. Since embedded real-time systems are typically multi threaded, they are easier to adapt to multi-core than single-threaded, sequential programs, which need to be parallelized into multiple threads to benefit from multi-core. If the tasks are independent, it is simply a matter of deciding on which core each task should execute. For embedded real-time systems, a static and manual assignment of cores is often preferred for predictability reasons. However, many of today's existing "legacy" real-time systems are very large and complex, typically consisting of millions of lines of code which have been developed and maintained for many years. Due to the huge development investments, it is normally not an option to throw them away and to develop a new system from scratch. However introducing new functionalities into the legacy systems may require more powerful processors, therefore, to benefit from multi-core processors, they need to be migrated from single-core architectures to multi-core architectures.

A significant challenge when migrating legacy real-time systems to multi-core processors is that they have been developed for single-core processors where the execution model is actually sequential. This assumption may introduce complications in a migration to multi-core [6]. Thus the software may need adjustments where assumptions of single-core have impact, e.g., non-preemptive execution may not be sufficient to protect shared resources.

Migrating legacy systems to multi-core processors is discussed in [9]. Advantages and disadvantages of different target architectures of multi-core processors are compared.

In this paper we present an algorithm for migration based on a heuristic *partitioning* which allocates tasks to the cores. Tasks can be both legacy tasks extracted from the legacy system as well as newly developed ones. The algorithm identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, which impact multi-core migration. The algorithm tries to increase the performance by reducing the overheads (e.g., blocking times and cache miss overheads) by assigning tasks to appropriate partitions. Partitioning is a bin-packing problem which is known to be a NP-hard problem in the strong sense; therefore finding an optimal solution in polynomial time is not realistic in the general case. Heuristic functions have been considered to find near-optimal solutions. In this paper we extend a bin-packing algorithm with task constraints which considers

* This work was partially supported by the Swedish Foundation for Strategic Research (SSF) via the strategic research centre (PROGRESS) at Mälardalen University.

performance as well as schedulability of partitions assigned to the cores.

1.1. Related Work

An approach for migration to multi-core is presented by Lindhult in [10]. The author presents the parallelization of sequential programs as a way to achieve performance on multi-core processors. The targeted language is PLEX, Ericsson's in-house developed event-driven real-time programming language used for Ericsson's telephone exchange system.

A work related to ours is presented in [15] where a scheduling framework for multi-core processors is presented. The framework tries to balance between the abstraction level of the system and the performance of the underlying hardware. The framework groups dependant tasks, which for example share data, to improve the performance. The paper presents Related Thread ID (RTID) as a mechanism to help the programmers to identify groups of tasks. However the framework targets new development systems and does not mention migration of existing legacy systems with single-core assumptions.

Liu *et al* [11] present a heuristic algorithm for allocating tasks in multi-core based massively parallel systems. Their algorithm has two rounds; in the first round processes (groups of threads - partitions in this paper) are assigned to processing nodes, the second round allocates tasks in a process to the cores of a processor.

The grey-box modeling approach for designing real-time embedded systems [14] is of relevance to our work. In the grey-box task model the focus is on task-level abstraction and it targets performance of the processors as well as timing constraints of the system. In this approach the design problems that are targeted at task-level are (1) task concurrency extraction from the system specifications, (2) automatic scheduling algorithm selection, (3) allocation and assignment of processors, and (4) resource estimators, high level timing estimators and interface refinement. However, in our approach, except specifications of the new tasks, the legacy system is used as the main source of task concurrency and resource sharing information.

A study of bin-packing algorithms for designing distributed real-time systems is presented in [13]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes.

Baruah and Fisher have presented a bin-packing partitioning algorithm (First Fit Decreasing algorithm) in [4] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm assigns each task τ_i to the first processor, P_k for which both of following conditions, under the *Earliest Deadline First* (EDF) scheduling hold:

$$D_i - \sum_{\tau_j \in P_k} DBF^*(\tau_j, D_i) \geq C_i$$

and

$$1 - \sum_{\tau_j \in P_k} u_j \geq u_i$$

where C_i, D_i and T_i specify worst-case execution time (WCET), deadline and period of task τ_i respectively, $u_i = \frac{C_i}{T_i}$, and

$$DBF^*(\tau_i, t) = \begin{cases} 0, & if\ t < D_i \\ C_i + u_i \times (t - D_i), & \text{otherwise} \end{cases}$$

The algorithm, however, assumes that tasks are independent while in practice tasks share resources and therefore blocking time overheads must be considered while schedulability of tasks assigned to the a core is checked. Our algorithm not only considers resource sharing when distributing tasks but it tries to reduce blocking times along with other costs. On the other hand their algorithm works under the EDF scheduling protocol while most of legacy real-time systems use fixed priority scheduling policies. Our proposed algorithm works under fixed priority scheduling protocols as well as other policies.

1.2. Multi-Core Platforms

A multi-core processor is a combination of two or more independent cores on a single chip. They are connected to a single shared memory via a shared bus. The cores typically have independent L1 caches and share an on-chip L2 cache. Figure 1 depicts an example of the architecture.

There are two approaches for scheduling sporadic and periodic task systems on multi-core systems [2, 4, 5, 7] which are inherited from multiprocessor systems; *global* and *partitioned* scheduling.

Under global scheduling, e.g., *Global Earliest Deadline First* (G-EDF), tasks are scheduled by a single scheduler based on their priorities and each task can be executed on any core. A single global queue is used for storing jobs. A task as well as a job can be preempted on a core and resumed on another core (migration of tasks among cores is permitted).

Under partitioned scheduling tasks are statically assigned to cores and tasks within each core are scheduled by uniprocessor scheduling protocols, e.g., *Rate Monotonic* (RM) and EDF. Each core is associated with a separate ready queue for scheduling task jobs.

However there are systems in which some tasks cannot migrate among cores while other tasks can migrate. For such systems neither of global or partitioned scheduling methods can be used. A two-level hybrid scheduling [7] which is a mix of global and partitioned scheduling methods is used for those systems.

Partitioned scheduling protocols have been used more often, as they are more predictable. However, finding an optimal partitioning of tasks on the cores is known to be

NP-hard. Thus heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been studied to find a near-optimal partitioning [2, 5].

While in practice tasks share resources, many of scheduling protocols for multiprocessors (multi-cores) assume independent tasks. However, synchronization which is not less important than scheduling has received less attention.

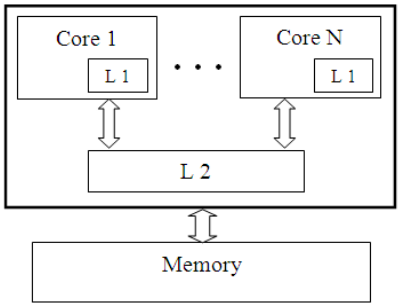

Figure1: Multi-core architecture

Most legacy systems use Fixed Priority Scheduling (FP) protocols. To our knowledge the only synchronization protocol under fixed priority scheduling, for multiprocessor platforms is *Multiprocessor Priority Ceiling Protocol* (MPCP) which was proposed by Rajkumar in [16]. Thus the protocol is suitable for legacy systems when migrating to multi-cores. Our algorithm assumes that MPCP is used for lock-based synchronization. Hence, we will discuss this protocol in more details in Section 3.

The rest of the paper is as follows: we present the task and platform model in Section 2, describe the MPCP in Section 3. We present the migration framework and the partitioning algorithm in Sections 4 and 5 respectively. In Section 6 we use our algorithm to reduce blocking time overheads under MPCP.

2. Task and Platform Model

We will assume a task set (tasks extracted from legacy system along with new tasks) that consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{c_{i,p,q}\})$ where T_i is the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. The tasks share a set of resources, R which are protected using semaphores. The set of critical sections in which task τ_i requests resources in R is denoted by $\{c_{i,p,q}\}$, where $c_{i,p,q}$ indicates the maximum execution time of the p^{th} critical section of task τ_i in which the task locks any resource $R_q \in R$. Critical sections of tasks can be sequential or properly nested. The deadline of each job is equal to T_i. A job of task τ_i, is specified by J_i .The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

We will also assume that the multi-core platform is composed of *m* identical, unit-capacity processors (cores). The task set is partitioned into *m* partitions $\{P_1, \dots, P_m\}$, and each partition is allocated on one core.

3. The MPCP-multiprocessor priority ceiling protocol

3.1. Definition

The MPCP was proposed by Rajkumar in [16] for synchronizing a set of tasks sharing lock-based resources under partitioned FP scheduling, i.e., RM.

Under MPCP, resources are divided into *local* and *global* resources. Local resources are shared only among tasks from the same processor and global resources are shared by tasks assigned to different processors. The local resources are protected using a uniprocessor synchronization protocol, i.e., priority ceiling protocol (PCP) [17]. A task blocked on a global resource suspends and makes the processor available for the local tasks. A critical section in which a task performs a request for a global resource is called *global critical sections* (*gcs*). Similarly a critical section where a task requests for local resource is *local critical sections* (*lcs*)

The blocking time of a task in addition to local blocking, needs to include *remote blocking* where a task is blocked by tasks (with any priority) executing on other processors (cores). However, the maximum remote blocking time of a job is bounded and is a function of the duration of critical sections of other jobs. This is a consequence of assigning any *gcs* a ceiling greater than priority of any other task, hence a *gcs* can only be blocked by another *gcs* and not by any non-critical section. If ρ_H is the highest priority among all tasks, the ceiling of any global resource R_k will be $\rho_H + 1 + \max\{\rho_i | \tau_i \; requests \; R_k\}$. The priority of a job executing within a *gcs* is the ceiling of the global resource it requests in the *gcs*.

Global critical sections cannot be nested in local critical sections and vice versa. Global resources potentially lead to high blocking times, thus tasks sharing the same resources are preferred to be assigned to the same processor as far as possible. In Section 6, our proposed algorithm attempts to reduce the blocking times by assigning tasks to appropriate processors.

To determine the schedulability of each processor under RM scheduling the following test is performed:

$$\forall k \; 1 \le i \le n, \sum_{k=1}^{i} C_k/T_k + B_i/T_i \le i\left(2^{1/i} - 1\right) \quad (1)$$

where n is the number of tasks assigned to the processor, and B_i is the maximum blocking time of task τ_i which includes remote blocking factors as well as local blocking time.

However this condition is sufficient but not necessary. Thus for schedulability test of tasks the response time analysis may be used to test if the condition (1) is not true for some tasks.

3.2. Blocking times of tasks

Before explaining the blocking factors of blocking time of a job, we have to explain the following terminology:

- n_i^G: The Number of global critical sections of task τ_i.
- $NL_{i,r}$: The number of jobs with priority lower than the priority of J_i executing on processor P_r.
- $\{J'_{i,r}\}$: The set of jobs on processor P_r (other than J_i's processor) with global critical sections having higher priority than global critical sections of jobs that can directly block J_i.
- $NH_{i,r,k}$: The number of global critical sections of job $J_k \in \{J'_{i,r}\}$ having higher priority than a global critical section on processor P_r that can directly block J_i.
- $\{GR_{i,k}\}$: The set of global resources that will be locked by both J_i and J_k.
- $NC_{i,k}$: The number of global critical sections of J_k in which it request a global resource in $\{GR_{i,k}\}$.
- ${\beta_i}^{local}$: The longest local critical section among jobs with a priority lower than job J_i executing on the same processor as J_i which can block J_i.
- ${\beta L_{i,k}}^{global}$: The longest global critical section of job J_k with a priority lower than job J_i executing on a different processor than J_i's processor in which J_k requests a resource in $\{GR_{i,k}\}$.
- ${\beta H_{i,k}}^{global}$: The longest global critical section of job J_k with a priority higher than job J_i executing on a different processor than J_i's processor. In this global critical section, J_k requests a resource in $\{GR_{i,k}\}$.
- ${\beta'}_{i,k}^{global}$: The longest global critical section of job $J_k \in \{J'_{i,r}\}$ having higher priority than a global critical section on processor P_r that can directly block J_i.
- ${\beta_{i,k}}^{lg}$: The longest global critical section of a lower priority job J_k on the J_i's host processor.

The maximum blocking time B_i of task τ_i is a summation of five blocking factors:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} + B_{i,5}$$

where:

1. $B_{i,1} = n_i^G {\beta_i}^{local}$ each time job J_i is blocked on a global resource and suspends the local lower priority jobs may execute and lock local resources and block J_i when it resumes.
2. $B_{i,2} = n_i^G {\beta L_{i,k}}^{global}$ when a job J_i is blocked on a global resource which is locked by a lower priority job executing on another processor.
3. $B_{i,3} = \sum_{\substack{\rho_i \le \rho_k \text{ and} \\ J_k \text{is not on } J_i\text{'s processor}}} NC_{i,k} \lceil T_i/T_k \rceil {\beta H_{i,k}}^{global}$ when higher priority jobs on processors other than J_i's processor block J_i.
4. $B_{i,4} = \sum_{\substack{J_k \in \{J'_{i,r}\} \text{ and} \\ P_r \neq J_i\text{'s processor}}} NH_{i,r,k} \lceil T_i/T_k \rceil {\beta'}_{i,k}^{global}$ when the *gcs*'s of lower priority jobs on processor P_r (different from J_i's processor) are preempted by higher priority *gcs*'s of $J_k \in \{J'_{i,r}\}$.
5. $B_{i,5} = \sum_{\substack{\rho_k \le \rho_i \text{ and} \\ J_k \text{ on } J_i\text{'s processor}}} \min(n_i^G + 1, n_k^G) {\beta_{i,k}}^{lg}$ when J_i is blocked on global resources and suspends a local job J_k can execute and enter a global section which can preempt J_i when it executes in non-*gcs* sections.

4. Migration Framework

We propose an algorithm that groups tasks into partitions and allocates each partition to a core. At each step when the algorithm assigns a task to a partition the following requirements should be satisfied:

1. Schedulability of the partition is guaranteed.
2. The cost of assigning the task to the partition is minimized.

	τ_1	τ_2	...	τ_i	...	τ_n
τ_1	-	34	...	16	...	0
τ_2	34	-	...	8	...	6
...	...	...	...	...	...	...
τ_j	13	32	...	v_{ij}	...	57
...	...	...	...	...	...	...
τ_n	0	6	...	11	...	-

Figure 2: Task preferences constraints

We derive a cost function that calculates the cost value based on a set of task constraints and preferences which should be extracted from the system as well as those offered by the system experts (Figure 3). Task constraints and preferences are defined in next Section.

4.1. Constraints and preferences

The partitioning algorithm uses the cost function to efficiently distribute tasks among partitions. The cost function is based on following constraints and preferences:

1. *Resource sharing constraints*:

These constraints indicate the critical sections of, and the resources accessed by each task.

2. *Task constraints*:

Specify timing attributes, e.g., deadline, worst-case execution time (WCET). Those constraints together with resource sharing constraints are used to check the schedulability of each partition.

3. *Task preferences*:

A preference category for the task set is represented as a matrix. Figure 2 shows an example of such constraints. A cost given to a pair of tasks, τ_i and τ_j is denoted by v_{ij} and indicates the cost when they are assigned to the same partition, i.e., if two tasks are completely independent and can execute in parallel the cost is set to a large value, and for two tasks that are highly recommended to belong to the same partition the cost is set to a very small value. Each matrix, M_k, represents an aspect of preferences (e.g. communication costs) and has a coefficient E_k which represents the importance of the preference category. Coefficient values depend on the *partitioning strategies* (Section 4.2).

Extracting preference matrices is not easy and for complex systems it may require a lot of engineering skills and system knowledge. Hence, the extraction complexity may differ for different matrices. For example Suppose in a system, tasks share large amounts of data, hence increasing cache hits is important. The values in the related matrix could be a function of amount of shared data between task pairs.

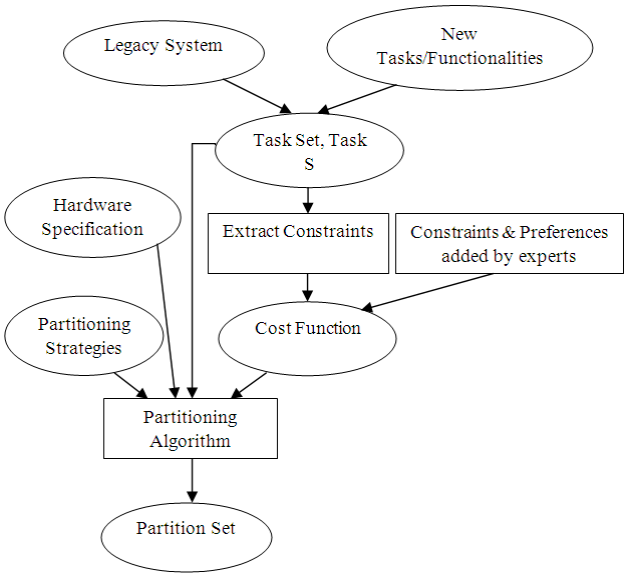

Figure 3: A framework for partitioning

4.2. Partitioning Strategies

Depending on the nature of a system the strategy of partitioning may differ and result in different partitions. A strategy indicates how tasks are grouped together and based on that the coefficient parameters are given to different preference matrices. For example in a system that processes large amounts of data it is important that the tasks that share data heavily are assigned to the same partition to increase cache hits. On the other hand for a system in which tasks share small amounts of data or are independent, it is important that the tasks are assigned to different partitions to increase parallelism.

The partitioning strategy in Section 6 represents extracting a matrix from resource sharing constraints which is used by partitioning algorithm. The partitioning strategy in Section 6 is to reduce blocking times under MPCP.

4.3. Cost Function

Considering z task preference matrices, the cost function for a partition is formulated based on the task preferences. Let $M_l(v_{ij})$ denote the cost of task τ_i and τ_j being assigned to the same partition in preference matrix M_l with coefficient value E_l. For any partition P_k (where $1 \leq k \leq m$ and m is the total number of partitions/cores), $cost(P_k)$ denotes the total cost of the partition:

$$cost(P_k) = {u_k}^{\alpha} \sum_{l=1}^{z} \left(E_l \sum_{\substack{\tau_i \in P_k \\ \tau_j \in P_k}} \frac{M_l(v_{ij})}{2} \right) \qquad (2)$$

where, $u_k = \sum_{\tau_i \in P_k} u_i$, and α is the *utilization parameter*.

The utilization parameter, α, where $\alpha = 0$ *or* $\alpha = 1$, indicates the importance of task utilizations in the cost function. By setting the utilization parameter to 0 ($\alpha = 0$), the cost function will only depend on the preference matrices. On the other hand by setting $\alpha = 1$ the cost function will also depend on utilization factor of the partition which will increase evenly distribution of tasks among partitions. The total cost of the system is the summation of costs of all partitions.

5. Partitioning Algorithm

Now we present an extension to the First-Fit bin-packing algorithm for partitioning sporadic task systems, similar to the algorithm presented in [4]. The major goal of bin-packing algorithms is minimizing the number of needed bins (cores). However our aim is to increase performance while guaranteeing correctness. Thus, we extend the bin-packing algorithm with task preferences (cost function) as well as resource sharing constraints.

The algorithm assumes that tasks are ordered non-increasingly based on their *weights*. The weight of a task τ_i, denoted by w_i indicates the importance of the task according to the partitioning strategy. For example in the partitioning strategy for reducing inter-core communication, the weight of a task may be the total number of messages it sends or receives during its execution time. Figure 4 depicts the pseudo-code for the partitioning algorithm.

```
// The task set {τ1, ... , τn} is to be assigned into m partitions, {P1, ... , Pm}, which will
// be allocated on m identical cores.
1  order the task set {τ1, ... , τn} based on their weights;
2  for each partition Pk    // m partitions
3        empty Pk;
4  end for
5  for i ← 1 to n    // n is th enumber of tasks
6        pick the task τi from the top of the ordered list;
7        order partitions by ascending order in cost increment assuming τi is assigned to them;
8        for j ← 1 to m    // i ranges over the ordered partitions
9              calculate blocking times of all tasks in all partitions according to MPCP;
10             if all tasks in all partitions satisfy condition (1) then
11                   assign τi to Pk;
12             end if
10       end for
13 end for
14 if all tasks are assigned to partitions then
15       partitioning succeeded;
16       goto line 19
17 end if
18 partitioning failed;
19 end
```

Figure 4: Partitioning algorithm

The schedulability test (1) (Section 3) is used for schedulability analysis of any partition, P_k. At each step that the algorithm assigns a task to a partition, P_k, the schedulability test should be performed for all other partitions as well, since the remote blocking term of any task in any partition may be affected.

The algorithm is not limited to FPS and MPCP, and the schedulability test can be extended to other scheduling and resource sharing protocols, e.g., for Partitioned Earliest Deadline First (P-EDF) using the Multiprocessor Stack-based Resource sharing Protocol (MSRP) [8], the following schedulability test from [3] may be used:

$$\sum C_i/T_i + \max_{\tau_i}(B_i/T_i) \leq 1 \qquad (3)$$

6. Reduce blocking times under MPCP

6.1. Partitioning strategy

In this section we present a partitioning strategy that targets reducing the blocking times under MPCP. We will use our algorithm to assign tasks to partitions according to the partitioning strategy.

Considering the blocking factors of tasks under MPCP, tasks with more and longer global critical sections lead to more blocking times. This is also shown by experiments presented in [8]. The goal is to (i) decrease the global critical sections by assigning the tasks sharing resources to the same partition as far as possible, (ii) decrease the ratio and time of holding global resources by assigning the tasks that request the resources more often and hold them longer to the same partition as long as possible.

The algorithm (Section 5) assumes that the tasks are ordered according to their weights. Since the partitioning strategy is to reduce blocking times, the tasks that may cause higher blocking times should get higher weights. Thus the weight of task τ_i should be a function of the number of its critical sections as well as the length of its largest critical sections:

$$w_i = \sum_{q=1}^{|R|}(n_{\forall cs_p}\{c_{i,p,q}\} \times m_{\forall cs_p}\{c_{i,p,q}\})/T_i \qquad (4)$$

where $n\{c_{i,p,q}\}$ is the number of critical sections in which τ_i requests resource R_q, $m\{c_{i,p,q}\}$ denotes the largest critical section of τ_i requesting R_q, and $|R|$ is the total number of resources in R.

The tasks will be ordered based on their weights and each time the algorithm attempts to assign a task to a partition it will pick the first task (with the highest weight).

Now we will derive a preference matrix which will contain the pair costs (v_{ij}) for each task pair τ_i and τ_j (Section 4.3). First, for any resource R_q we derive an individual matrix in which the cost of pair τ_i and τ_j denoted as $v_{ij,q}$ will be a function of the number of critical sections as well as the length of largest critical sections of tasks τ_i and τ_j:

$$v_{ij,q} = -n\{c_{i,p,q}\} \times m\{c_{i,p,q}\} \times n\{c_{j,k,q}\} \times m\{c_{j,k,q}\} + 1 \qquad (5)$$

As the number and the maximum length of critical sections of task pairs increases the cost of assigning them to the same partition should decrease. This is why that first term of the cost in (5) has a negative form. If two tasks do not share resource R_q the first term of $v_{ij,q}$ will be *0*, hence $v_{ij,q} = 1$ which means if they are assigned to the same partition the cost of the partition should be increased. This is logical because regarding R_q they are independent and are not recommended to be assigned to the same partition.

Table 1: The task set to be partitioned

Task	Period	C_i in non-critical sections	$n\{c_{i,p,1}\}$	$m\{c_{i,p,1}\}$	$n\{c_{i,p,2}\}$	$m\{c_{i,p,2}\}$	$n\{c_{i,p,3}\}$	$m\{c_{i,p,3}\}$	$n\{c_{i,p,4}\}$	$m\{c_{i,p,4}\}$	$n\{c_{i,p,5}\}$	$m\{c_{i,p,5}\}$
τ_1	39	4	1	1	0	0	1	1	0	0	0	0
τ_2	41	5	0	0	1	1	1	1	0	0	0	0
τ_3	42	4	0	0	0	0	0	0	1	1	0	0
τ_4	48	3	0	0	1	2	0	0	1	1	0	0
τ_5	52	5	0	0	0	0	1	2	0	0	1	1
τ_6	57	5	0	0	0	0	0	0	1	1	1	1
τ_7	58	6	1	1	0	0	0	0	2	1	0	0
τ_8	63	8	0	0	0	0	0	0	0	0	0	0

The individual matrices for each resource are then used to derive the preference matrix in which v_{ij} (the cost of pair τ_i and τ_j if they are assigned to the same partition) will be as follows:

$$v_{ij} = \sum_{R_q \in R} v_{ij,q} \quad (6)$$

The partitioning algorithm will use the obtained preference matrix for assigning the tasks to partitions.

6.2. Example

In this section we present an example in which our algorithm will attempt to reduce blocking times while partitioning a task set onto different cores of a multi-core processor. The partitioning is performed based on the partitioning strategy in Section 6.1.

In this example we set $\alpha = 0$ in the cost function so that the cost only depends on blocking time costs. We attempt to assign a task set consisting of eight tasks (Table 1) into four partitions which will be assigned onto a processor with four cores. There are five resources, $\{R_1, R_2, R_3, R_4, R_5\}$ which are shared among tasks and are protected by semaphores. The tasks in Table 1 are indexed based on their periods (priority). For each task τ_i, the table contains the period, WCET of non-critical sections, the number of critical sections in which the task request R_q ($n\{c_{i,p,q}\}$) and WCET of the largest critical section for resource R_q ($m\{c_{i,p,q}\}$).

Table 2: The task weights

Task	Weight
τ_4	0,063
τ_5	0,058
τ_1	0,053
τ_7	0,052
τ_2	0,049
τ_6	0,035
τ_3	0,024
τ_8	0

First, the weights of tasks are calculated based on formula (4). Table 2 shows the ordered list of tasks based on the calculated weights. For each resource a matrix was created which contains the costs for each task pairs calculated by formula (5). and the final preference matrix was obtained based on the resource matrices. Table 3 shows the preference matrix which includes the costs for each pair of tasks. Since we only have one preference matrix we set the coefficient of the matrix, E_1, to 1 ($E_1 = 1$).

Table 3: The preference matrix

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8
τ_1	-	4	5	5	3	5	4	5
τ_2	4	-	5	3	3	5	5	5
τ_3	5	5	-	4	5	4	3	5
τ_4	5	3	4	-	5	4	3	5
τ_5	3	3	5	5	-	4	5	5
τ_6	5	5	4	4	4	-	3	5
τ_7	4	5	3	3	5	3	-	5
τ_8	5	5	5	5	5	5	5	-

While partitioning the task set using the bin-packing algorithm without considering the blocking costs does not result in a schedulable system, our algorithm, based on the preference matrix, successfully partitions the task set onto four partitions. Task sets $\{\tau_4, \tau_2\}$, $\{\tau_5, \tau_3\}$, $\{\tau_1, \tau_8\}$, and $\{\tau_7, \tau_6\}$ are assigned to partitions P_1, P_2, P_3, and P_4 respectively. Table 4 shows the five blocking factors and total blocking time for each task in the obtained system.

Table 4: The blocking times of tasks

Task	$B_{i,1}$	$B_{i,2}$	$B_{i,3}$	$B_{i,4}$	$B_{i,5}$	$\boldsymbol{B_i}$
τ_1	0	4	0	0	0	4
τ_2	2	2	2	0	1	7
τ_3	0	1	0	4	4	9
τ_4	0	1	2	4	0	7
τ_5	0	2	4	1	0	7
τ_6	0	0	6	10	3	19
τ_7	0	0	6	6	0	12
τ_8	0	0	0	0	0	0

7. Summary and Future Work

In this paper we have mentioned the major challenges (targeting performance and correctness) of migrating a legacy real-time system to multi-core architectures where

it will execute along with other systems, e.g., how to take advantage of performance offered by multi-core platforms while guaranteeing correctness. We have proposed a framework for migrating legacy real-time systems to multi-core processors, which includes a heuristic algorithm that extends a bin-packing algorithm with a cost function based on preference matrices. Each obtained partition will be mapped on one core.

Since most legacy real-time systems use fixed priority scheduling protocols, we have developed our framework based on MPCP, the only existing synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling. However, this protocol introduces large amounts of blocking time overheads especially when the global resources are relatively long and the access ratio to them is high. As an example we have presented a partitioning strategy and we have obtained preference matrices based on critical sections. The cost function is calculated based on the obtained preference matrix and finally, the algorithm uses the cost function to reduce blocking times.

Our algorithm depends on attributes of tasks, and for legacy systems some information about tasks should be extracted from the existing system. In the future we will study and investigate techniques including reverse engineering methods such as static and dynamic analysis. We will use these methods to extract required information from the legacy system, e.g., information about shared resources, and timing attributes.

A future work will be evaluation of our framework by means of simulation and applying it to a real system. We also plan to study industrial legacy real-time systems and investigate the challenges and possibility of migrating these systems to multi-core architectures. Our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. *University of California at Berkeley, Technical Report No. UCB/EECS-2006-183*, December 2006.

[2] T. Baker. A Comparison of Global and Partitioned EDF Schedulability Test for Multiprocessors. *Technical Report TR-051101,* Department of Computer Science, Florida State University, 2005.

[3] T. Baker. Stack-based Scheduling of Real-time Processes. *J.Real-Time Systems, vol. 3, no. 1, pages 67-99,* March, 1991.

[4] S. Baruah, and N. Fisher. The Partitioned Multiprocessor Scheduling of Sporadic Task Systems. *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05), pages 321 – 329,* December 2005.

[5] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In J. Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models, pages 30.1-30.19.* ChapmanHall/CRC, Boca Raton, Florida, 2004.

[6] R. Craig, and P. N. Leroux. Case Study - Making a Successful Transition to Multi-Core Processors. *QNX Software Systems GmbH & Co.* KG, 2006.

[7] U. Devi. Soft Real-Time Scheduling on Multiprocessors. *PhD thesis*, October 2006, *http://www.cs.unc.edu/˜anderson/diss/devidiss.pdf.*

[8] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. 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.*, *pages 189-198*, May 2003.

[9] P. Leroux, and R. Craig. Migrating Legacy Applications to Multicore Processors. *Military Embedded Systems http://www.mil-embedded.com /pdfs/QNX.Sum06.pdf*, 2006.

[10] J. Lindhult. Operational Semantics for PLEX A Basis for Safe Parallelization. *Licentiate Thesis, No. 85, Mälardalen University*, May 2008.

[11] Y. Liu, X. Zhang, H. Li, and D. Qian. Allocating Tasks in Multi-core Processor based Parallel Systems. *Network and Parallel Computing Workshops, IFIP International Conference, pages 748-753*, September 2007.

[12] J. M. López , J. L. Díaz , and D. F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems, v.28 n.1, pages 39-68,* October 2004.

[13] D. de Niz, and R. Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems. *International Journal of Embedded Systems, Vol. 2, No. 3-4, pages 196-208*, 2006.

[14] A. Prayati, C. Wong, P. Marchal, F. Catthoor, H. de Man, N. Cossement, R. Lauwereins, D. Verkest, and A. Birbas. Task Concurrency Management Experiment for Power-Efficient Speed-Up of Embedded MPEG4 IM1 Player. *International Conference on Parallel Processing Workshops (ICPPW'00), pages 453-460,* 2000.

[15] M. Rajagopalan, B. T. Lewis, and T. A. Anderson. Thread Scheduling for Multi-Core Platforms. *In Proceedings of the 11 th Workshop on Hot Topics in Operating Systems (HotOS'07)*, May 2007.

[16] R. Rajkumar. Synchronization in multiple processor systems. *In Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[17] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time System Synchronization. *IEEE Transactions on Computers, 39(9), pages 1175-1185,* 1990.