A Framework for Real-Time Systems Migration to Multi-Cores*

Farhang Nemati, Johan Kraft and Thomas Nolte
Mälardalen Real-Time Research Centre
Mälardalen University, Box 883, 72123, Sweden
{farhang.nemati, johan.kraft, thomas.nolte}@mdh.se

Abstract

Power consumption and thermal problems limit a further increase of speed in single-core processors. Processor architects are therefore moving toward multi-core processors. 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 single-core processor 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. In this paper we investigate challenges of migrating complex legacy real-time systems to multi-core architectures. We propose a partitioning algorithm to prepare the migration. Partitioning groups task and maps them to the different cores on the multi-core processor, increasing system performance while ensuring correctness. We have run experiments that compare outputs of the algorithm to the outputs of an exhaustive search. Based on a cost function, the algorithm produces systems very close to optimal partitioning with respect to the cost function.

1. Introduction

Traditionally, computing performance has been improved through increasing clock frequency of processors. However, higher clock frequency results in higher power consumption [16]. Due to the problems with power consumption and related thermal problems, processor architects are moving toward multi-core designs. Multi-core is today the dominating technology for desktop computing.

The performance achieved by multi-core architectures was previously only provided by High Performance Computing (HPC) systems. The HPC programmers are required to have a deep understanding of the hardware architecture in order to adjust the program explicitly for that hardware. This is not a suitable approach in embedded systems development, due to requirements on productivity, portability, maintainability, and short time to market.

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. The migration should maximize the performance without compromising correctness and quality attributes such as maintainability, and portability. 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.

Multithreading and multi-core architecture concepts are discussed in [16]. The author discusses parallelism, its software impacts and tuning performance on multi-core platforms. Migrating legacy systems to multi-core processors is discussed in [8]. Advantages and disadvantages of different target architectures of multi-core processors are compared.

For real-time systems, correctness does not only depend on functional correctness, but also on temporal

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

correctness. The temporal behavior of a real-time system, e.g., worst-case response time, generally depends on the underlying hardware. Thus, in order to migrate a legacy system, a higher level of abstraction as well as environmental (platform dependent) properties of the system should be provided. This means that two perspectives of the system should be considered, a platform independent view, which focuses on design entities (functional behavior), and a platform dependent view, which provides a mapping between design entities and processor cores, which allows developers to best utilize the target platform.

In this paper we present a migration framework 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 constraints of the new tasks/functionalities are added to the constraints of tasks extracted from a legacy system. The framework identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, which impact multi-core migration. 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 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 [9]. 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. The author presents an operational semantics of core PLEX for both single-processor architecture as well as multi-threaded shared-memory architecture.

A related work to ours is presented in [14] 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 and does not mention migration of existing legacy systems with single-core assumptions.

Another approach similar to partitioning is presented by Gerber *et al* in [7] for task slicing as a compiler optimization technique to enhance the schedulability of tasks. The authors present a static method that uses an annotation language and task slicing. This work however targets single-core processors.

Liu *et al* [10] 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 [13] 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 [12]. 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 in [4] have presented a bin-packing partitioning algorithm (First Fit Decreasing algorithm) 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 EDF scheduling) hold:

$$\mathrm{d_i} - \sum_{\tau_j \in P_k} \mathrm{DBF}^*(\tau_j, \mathrm{d_i}) \geq \mathrm{e_i}$$

and,

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

where $\mathrm{u_i} = \frac{\mathrm{e_i}}{\mathrm{T_i}}$, and

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

The algorithm, however, assumes that tasks are independent. For tasks that share resource locks, research is needed to extend the schedulability conditions to include maximum blocking time for tasks.

2. Multi-Core Architectures

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 [4, 5] which are inherited from multiprocessor systems; global and partitioned scheduling. In global scheduling tasks are scheduled by a single scheduler based on their priorities and each task can be executed on any core. A task can be preempted on a core and resumed on another core, i.e., 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. Partitioned scheduling protocols have been used more, 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].

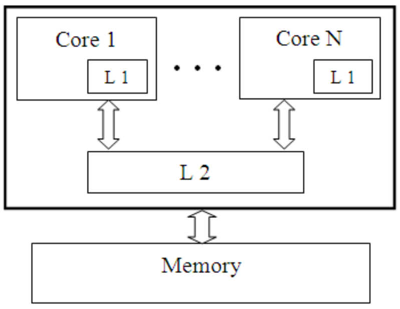


Figure1: Multi-core architecture

3. Migration Framework

A successful migration of a real-time system to a multi-core architecture should, besides correctness of the system functionality and timing behavior, take advantage of the performance offered by the multi-core architecture. 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_3	τ_4	τ_5
τ_1	-	34	18	12	0
τ_2	34	-	0	64	6
τ_3	18	0	-	2	321
τ_4	12	64	2	-	19
τ_5	0	6	321	19	-

(a)

	τ_1	τ_2	τ_3	τ_4	τ_5
R_1	-	-	124	-	24
R_2	164	-	85	-	-
R_3	-	46	-	96	-
R_4	-	-	15	-	32

(b)

Figure 2: Task preferences and resource sharing constraints

The cost function 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). Three types of task constraints and preferences are defined as follows:

1. *Resource sharing constraints*:

The tasks that share resource locks, in the case that only single-core resource sharing protocols have to be used, should belong to the same partition. Figure 2.b depicts resources that are locked by tasks. A value (in milliseconds) shows the maximum time that a task blocks a resource.

2. *Task constraints*:

Timing attributes, e.g. deadline, worst-case execution time (WCET).

3. *Task preferences*:

A preference category for the task set is represented as a matrix (Figure 2.a). A cost given to a pair of tasks, τ_i and τ_j is denoted by v_{ij} 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 very small value, and for two tasks that are highly recommended to belong to the same partition the cost is the highest 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. Coefficients values depend on the *partitioning strategies* (Section 3.1).

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.

Example 1: 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.

Example 2: Let assume that the Priority Ceiling Protocol (PCP) for uniproccessors [15] has been used for synchronization of tasks sharing resources. Suppose P_x and P_y are respectively the highest and the lowest priorities of tasks sharing a resource guarded by semaphore S_i. Any task, τ_i with priority higher than P_y and less than P_x may be blocked by lower priority tasks even if it does not share the resource. In this case it is better that τ_i is assigned to a different processor to decrease the blocking times. Thus in a preference matrix (for reducing blocking times) the cost values between τ_i and each of those tasks should be a high value. The cost value will be increased as the priority of τ_i gets closer to P_x.

3.1. Partitioning Strategies

Depending on the nature of a system the strategy of partitioning may differ and result in different partitioning. 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.

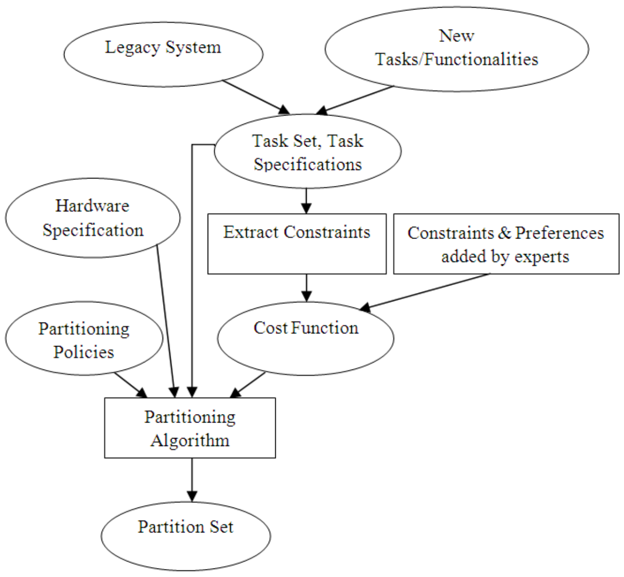


Figure 3: A framework for partitioning

3.2. Task and Platform Model

We will assume a task set (tasks extracted from legacy system as well as new tasks) that consists of n sporadic tasks $\{\tau_1, \ldots, \tau_n\}$, characterized by worst-case execution times $\{C_1, \ldots, C_n\}$, blocking times $\{B_1, \ldots, B_n\}$, and minimum inter-arrival times $\{T_1, \ldots, T_n\}$ that are equal to their deadlines. The utilization of task $\boldsymbol{\tau_i}$ is defined as $u_i = C_i/T_i$.

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

3.3. Cost Function

Considering q task preference matrices, the cost function for a partition is formulated based on the task preferences and utilizations. Let $\mathrm{M_l(v_{ij})}$ 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 (where $1 \leq l \leq q$). 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:

$$cost(P_k) = u_k{}^{\alpha_u} (\textstyle\sum_{l=1}^{q} \left(E_l \sum_{\substack{\tau_i \in P_k \\ \tau_j \in P_k}} \frac{M_l(v_{ij})}{2} \right))^{\beta} \quad (1)$$

where, $u_k = (\sum_{\tau_i \in P_k} u_i + max_{\tau_i \in P_k} (B_i/T_i))$,
α_u is the *utilization parameter*, and β is the *preference parameter*.

The utilization parameter, α_u where $0 \leq \alpha_u$ indicates the importance of task utilizations in the cost function, and β indicates the importance of preference matrices in the cost function. By setting the utilization parameter to 0 ($\alpha_u = 0$), the cost function will only depend on the preference matrices. By setting the preference parameter to 0 ($\beta = 0$) the cost function will only depend on utilization factor which will help the cores be evenly utilized.

```
// The task set {$\tau_1, \ldots, \tau_n$}, is transformed into the macrotask set {$\gamma_1, \ldots, \gamma_z$}, where $z \leq n$. The macrotaskset
   is to be assigned into m partitions, {$P_1, \ldots, P_m$}, which will be allocated on $m$ identical cores.
1  for $k \leftarrow 1$ to $m$    // $k$ ranges over the partitions in any order
2       $P_k \leftarrow \emptyset$
3  end for
4  for $i \leftarrow 1$ to $z$    // $i$ ranges over the macrotasks
5        order partitions by non-decreasing value of cost function (5) assuming $\gamma_i$ is assigned to them
6        for $k \leftarrow 1$ to $m$    // $k$ ranges over the ordered partitions
7              if $\gamma_i$ satisfies conditions (2) on patition $P_k$ then
8                  assign $\gamma_i$ to $P_k$
9              end if
10       end for
11 end for
12 if all macrotasks are assigned to partitions then
13      partitioning succeeded
14      goto line 17
15 end if
16 partitioning failed
17 end
```

Figure 4: Partitioning algorithm

4. Partitioning Algorithm

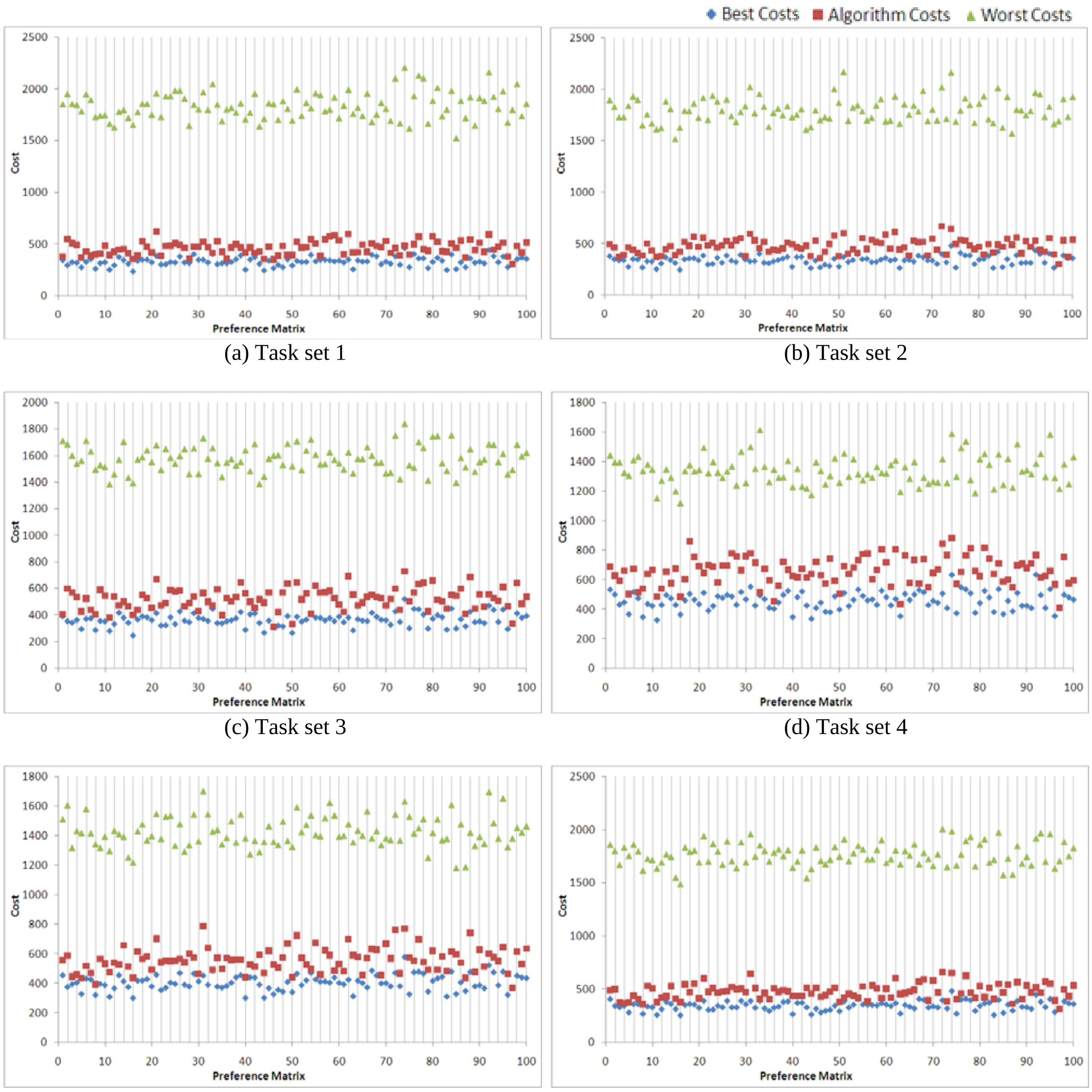
In this section we present an extension to the First-Fit bin-packing algorithm for partitioning sporadic task systems, similar to the algorithm presented in [4]. The major goal of bin-packing algorithms is minimizing the number of needed bins (cores). However our aim is to increase performance while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

Most legacy systems use Fixed Priority Scheduling (FPS). Thus, we assume that the uniprocessor scheduling and resource sharing protocols used in the system are FPS with Rate Monotonic (RM) priority assignment and PCP respectively. Thus the following schedulability test from [15] is used for schedulability analysis of any partition, P_k:

$$\sum_{\tau_i \in P_k} u_i + max_{\tau_i \in P_k}(B_i/T_i) \le n(2^{1/n} - 1) \qquad (2)$$

However the algorithm is not limited to FPS and PCP, and the schedulability test can be extended to other scheduling and resource sharing protocols, e.g., for Earliest Deadline First (EDF) using Stack Resource Protocol (SRP) [3], the following schedulability test from [3] can be used:

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



(a) Task set 1
(b) Task set 2
(c) Task set 3
(d) Task set 4
(e) Task set 5
(f) Task set 6

Figure 5: Results of the partitioning algorithm for 100 different preference matrices

In order to satisfy resource sharing constraints (Section 3), the tasks should be replaced by tasks called *macrotasks* [11]. A macrotask is a set of tasks that directly or indirectly share resource locks, e.g., in Figure 2.b tasks τ_3 and τ_5 share a resource (R_1), τ_3 shares another resource (R_2) with τ_1, these tasks (τ_l, τ_3, τ_5) make a macro task. A task that does not share any resource with other tasks make a macrotask too. The tasks $\{\boldsymbol{\tau}_1, \dots , \boldsymbol{\tau}_n\}$ are transformed into macrotasks $\{\gamma_1, \dots , \gamma_z\}$ where $z \leq n$.

Preference matrices with tasks (Section 3) are transformed into preference matrices with macrotasks. The cost for any pair of macrotasks, γ_p and γ_q, when they are allocated to the same partition, is denoted by $\hat{v}_{pq}$ and is calculated by Equation (5):

$$\hat{v}_{ij} = \sum_{\substack{\tau_i \in \gamma_p \\ \tau_j \in \gamma_q}} v_{ij} \qquad (4)$$

The cost function of any partition in (1) is transformed into cost function (5):

$$cost(P_k) = {u_k}^{\alpha_u}\left(\sum_{l=1}^{q}\left(E_l \sum_{\substack{\gamma_i \in P_k \\ \gamma_j \in P_k}} \frac{\hat{M}_l(\hat{v}_{ij})}{2}\right)\right)^{\beta} \qquad (5)$$

where, $u_k = \sum_{\tau_i \in P_k} u_i + max_{\tau_i \in P_k}(B_i/T_i)$ and $\hat{M}_l$ denotes the transformed matrix of M_l preference matrix and $\hat{M}_l(\hat{v}_{ij})$ denotes the cost of macrotasks γ_i and γ_j being assigned to the same partition in transformed preference matrix $\hat{M}_l$.

5. Experiments

In this section we present an evaluation of our algorithm. In the implementation of the algorithm we investigate two aspects. Firstly we compare the outputs with the results of an exhaustive search for all feasible allocations (schedulable system). Secondly we investigate the effect of the utilization parameter on evenly distribution of tasks in partitions.

In our experiments, to decrease the time of performing an exhaustive search, the number of tasks in each task set is restricted to 12 and the number of partitions (cores) is set to 3. The experiment is run for 10 different task sets that were generated randomly. For each task set the macrotasks were extracted from the tasks and transformation of preference matrices was performed. The experiment runs 100 times for each task set, each time with two randomly generated preference matrices. The coefficient of the preference matrix is set to 1 ($E = 1$). To investigate the effect of the utilization parameter α_u on partitioning, four different parameters were tested, i.e., $\alpha_u = 1$, $\alpha_u = 2$, $\alpha_u = 4$, and $\alpha_u = 6$. The preference parameter β, is set to 1 ($\beta = 1$) for all experiments.

The experiments were implemented under the uniprocessor RMS scheduling protocol and PCP synchronization protocol. However, as it was explained in Section 4, the schedulability test can easily be replaced if other scheduling protocols, e.g., EDF, or synchronization protocols, e.g., SRP, are used.

The results of comparing with exhaustive search for six different task sets are depicted in Figure 5. The diagrams show the outputs of the algorithm together with the best and the worst feasible cases. The best and the worst case results are achieved by exhaustive search based on the cost function. The results are emphasized in Table 2 where average of best, algorithm, and worst costs are shown for each task set. The diagrams in Figure 5 and average costs in Table 2 show that the outputs of the algorithm are generally very close to the best feasible cases with respect to the cost function.

Table 2: Average values of the best, the worst, and algorithm costs

	Average		
	Best Cost	Algorithm Cost	Worst Cost
Task Set 1	330,50	461,40	1 835,36
Task Set 2	338,80	478,90	1 796,43
Task Set 3	365,94	520,32	1 575,06
Task Set 4	457,83	657,55	1 339,41
Task Set 5	406,34	556,49	1 424,45
Task Set 6	342,83	475,38	1 764,15
Task Set 7	350,37	501,23	1 771,85
Task Set 8	347,74	485,20	1 772,53
Task Set 9	367,55	503,33	1 551,46
Task Set 10	389,95	576,40	1 545,87

The results also show that the number of systems that (according to the cost value) are better than the algorithm results is very low. Figure 6 compares the total number of better systems with the total number of feasible systems from the experiments of the first task set.

For one task set, the results of utilization of the partitions with four different utilization parameters, $\alpha_u = 1$, $\alpha_u = 2$, $\alpha_u = 4$, and $\alpha_u = 6$ are depicted in Figure 7. The diagrams show that increasing the parameter α_u leads to a more even distribution of utilization among partitions (cores). On the other hand, by increasing α_u, utilization results for each partition are converged, which means that the effect of preference matrices on the cost function decreases.

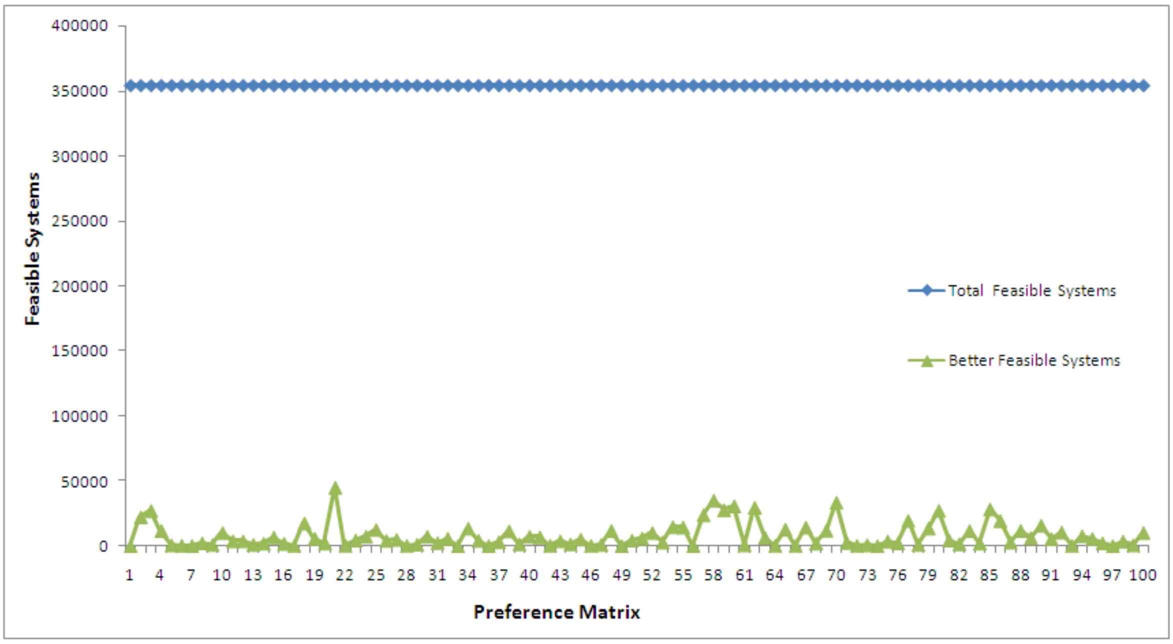
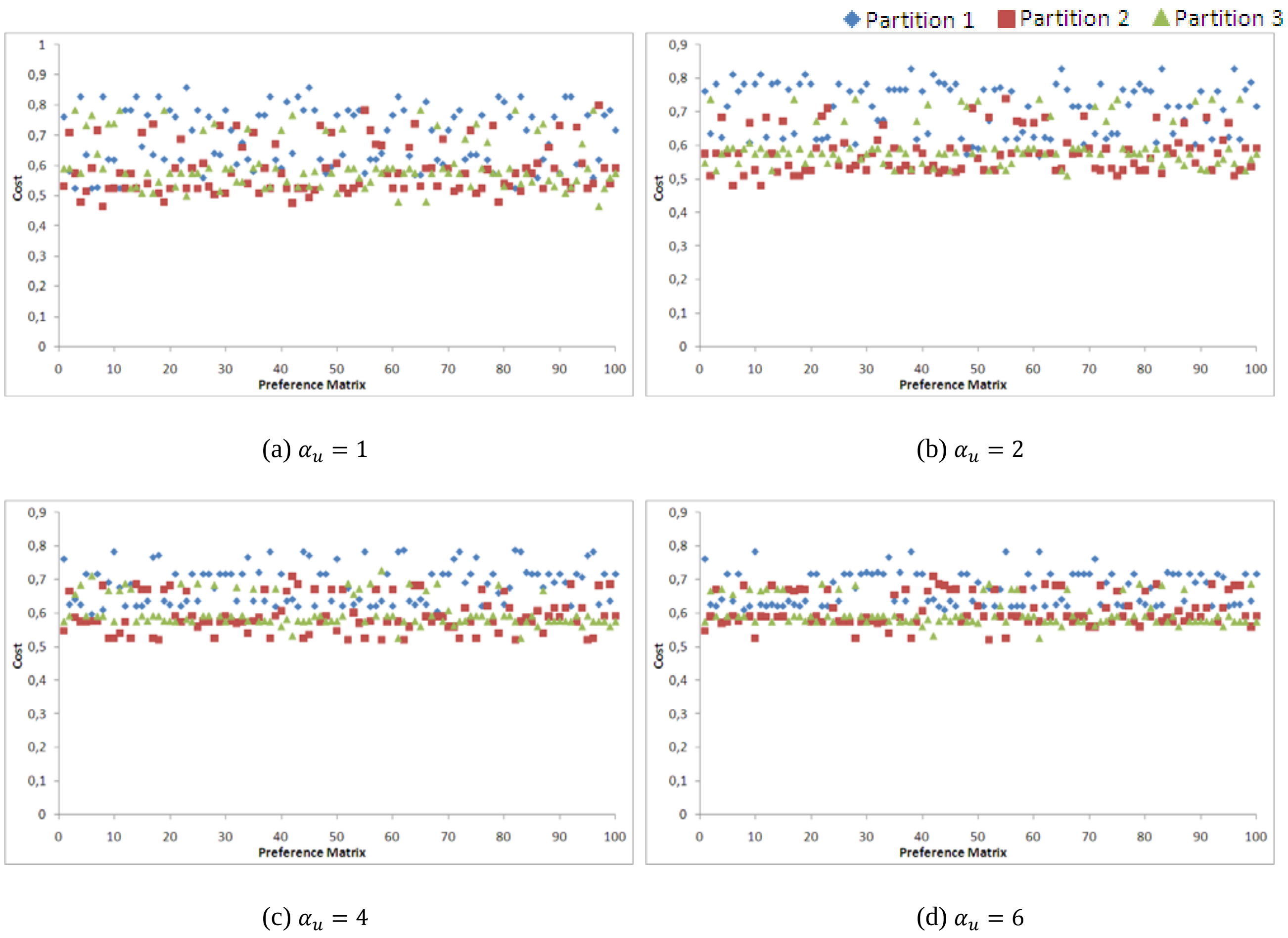


Figure 6: Systems better than algorithm results compared to total feasible systems



(a) $\alpha_u = 1$

(b) $\alpha_u = 2$

(c) $\alpha_u = 4$

(d) $\alpha_u = 6$

Figure 7: Utilization of partitions for different α_u values

6. Conclusions and Future Work

In this paper we have discussed major challenges regarding 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 and task utilization values. Partitioning will result in a set of partitions containing tasks and each partition will be mapped on one core. We have developed an experiment where results of our algorithm are compared to the results from performing an exhaustive search. Based on a cost function, the algorithm produces systems very close to optimal.

In the future we will study and investigate more techniques and the possibility of extending them to our framework, including reverse engineering techniques such as static and dynamic analysis. These techniques will be used to extract required information from the legacy system, e.g., the use of shared resources, and timing attributes.

Another plan that we have for the future is to study industrial legacy real-time systems and investigate the challenges and possibility of migrating these systems to multi-core architectures.

In our approach we assign all tasks that share resource locks to reuse uniprocessor synchronization protocols. However in the future we will investigate existing global synchronization protocols and possibly develop new protocols.

We have restricted our framework to partitioned scheduling approach for multi-cores, and we have reused uniprocessor scheduling. In the future we plan to extend the migration framework to global scheduling protocols, e.g., hierarchical scheduling protocols for multi-core architectures.

Acknowledgments

The authors wish to thank Moris Behnam for fruitful discussions and comments on the paper.

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] R. Gerber, and S. Hong. Slicing Real-Time Programs for Enhanced Schedulability. *ACM Transactions on Programming Languages and Systems, Vol.19, No.3, pages 525-555*, May 1997.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] C. Szydlowski. Multithreaded Technology & Multicore Processors. *Dr. Dobb's Journal*, May 2005.