Towards Migrating Legacy Real-Time Systems to Multi-Core Platforms

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 the single-core processors to be faster. Processor architects are therefore moving toward multi-core processors. Developers of embedded real-time systems however hesitates a shift to multi-core processors, especially for 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 can not 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 componentization and partitioning to prepare the migration. Componentization groups logically related tasks into components (or subsystems). This provides an abstraction layer from a scheduling perspective, which facilitates migration. Partitioning maps tasks to the different cores on the multi-core processor, maximizing system performance while ensuring correctness.

1. Introduction

Traditionally, computing performance has been improved through increasing clock frequency of processors. However, higher clock frequency needs more transistors and higher input voltage which results in higher power consumption [14]. 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 schedule them to maximize the utilization of the processors. 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 needs 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 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. To benefit from multi-core processors, they therefore 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 [3]. 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.

For real-time systems, correctness does not only depend on functional correctness, but also on temporal correctness. The temporal behavior of a real-time system, e.g. worst case response time, generally depends on the underlying hardware. Thus, 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 dependant view, which provides a mapping between design entities and processor cores, which allows developers to best utilize the target platform. In this work in progress paper we investigate *componentization* and

partitioning of a legacy system, which provides a high level design view and low level platform view, respectively. These concepts are explained in Section 3.

This paper proposes a framework for migration of legacy real-time systems to multi-core platforms, based on componentization and partitioning. The framework identifies dependencies between tasks and other behavior which impact multi-core migration. The result is a set of task models, which provides a multi-core architecture for the migrated system.

1.1. Related Work

Multithreading and multi-core architecture concepts are discussed in [14]. The author argues parallelism, its software impacts and tuning performance on multi-core platforms. Migrating legacy systems to multi-core processors is discussed in [6]. Advantages and disadvantages of different target architectures of multi-core processors are compared.

Lindhult in [8] presents the parallelization of sequential languages 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 similar work to ours is presented in [13] 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 share data, to improve the performance; we call this activity in our work as partitioning. The paper presents 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.

Related to the componentization concept we present in this paper is a case study at ABB presented in [10] which shows the experiences learned by redesigning of the architecture of ABB's existing control system, letting its IO and communication components be developed by different development centers around the world. It was decided to split the components into generic and non generic parts; the generic parts are hardware independent, shared by all hardware, and the implementations for specific hardware are in the non-generic parts. However, the case study presents the componentization approach as a way to ease redeveloping different parts of the system in different development centers and it doesn't target migrating to another platform, e.g. multi-core.

Comella-Dorda *et al* [2] point to extracting components and their relationships from a legacy system to get a higher level of abstraction of the system. This will help in understanding of the functionality of the system as an initial to white-box modernization. The paper targets information systems, whereas non-functional properties are not discussed.

We have found and studied several works similar to the partitioning part of our framework. Partitioning of embedded system for multi-core is discussed in [5]. However, the goal of partitioning in this paper is security issues of embedded systems.

Another approach similar to partitioning is presented by Gerber *et al* in [4] 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.

A study of bin-packing algorithms for designing distributed real-time systems is presented in [11]. 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 to minimize the number of bins (processors).

Liu *et al* [9] present a heuristic algorithm for allocating tasks in multi-core based massively paralleled 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 [12] is of relevance to our work. Similar to our framework, in the grey-box task model the abstraction level is at task level and it emphasizes on performance of the processors as well as on 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 (similar to partitioning in our approach), and (4) resource estimators, high level timing estimators and interface refinement. However, in our approach, besides task level, a higher level of abstraction is introduced with components as a result of componentization which reduces complexity and increases maintainability and portability of the legacy system.

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 shared bus. The cores typically have independent L1 caches and share an on-chip L2 cache. Figure 1 depicts an example of such architecture.

Two forms of multi-core architectures are inherited from multiprocessors with discrete CPUs. *Symmetric Multiprocessing* (SMP) in which a common OS manages all cores, allowing tasks to be transferred between cores to balance utilization. In the *Asymmetric Multiprocessing* (AMP), each core is managed by either a separate or the same operating system. Tasks are thereby assigned to

specific cores. It is relatively easy to migrate legacy software to AMP as each independent application can be assigned to a core, which appears as a single-core processor. Unlike SMP, AMP is not efficient because if a core gets too busy, its work cannot be transferred to other cores to balance the utilization between the cores. There is also a form of Multiprocessing known as *Bound Multiprocessing* (BMP), which combines the advantages of SMP and AMP. A single OS is used, which allows utilization balancing, but tasks can also be locked to individual cores.

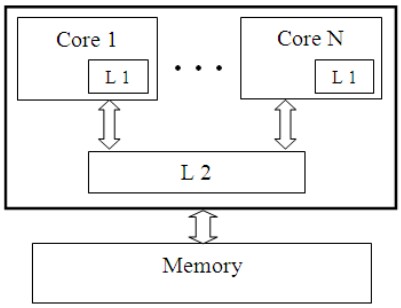

Figure1: Multi-core Architecture

3. Migration Framework

Migration of a complex real-time system to a multi-core architecture requires system models that capture a higher abstraction view (*Application models*) as well as platform dependant view (*Context models*). Application models give a description of the system which will preserve semantics in spite of the platform on which it is deployed, while context models facilitate the system to efficiently utilize the target platform. These two views of system are called *Opacity* and *Visibility* respectively in [1]. Opacity gives an abstraction of underlying hardware while visibility exploits the key elements of the hardware. These two views of the system may be conflicting, thus a tradeoff between implementation efficiency and achieving quality attributes may be needed.

3.1. Partitioning and Componentization of Legacy Systems

We propose an outline for a framework that generates two types of models from a real-time legacy system; the first type of models will represent a higher level abstraction of the system and are independent of the hardware, and the second type of models, which are hardware dependent, will be used to optimize the performance of the migrated system on a multi-core platform. To obtain these models we see two challenges.

- The first challenge is to divide the system into sets of tasks that are logically related and form a component. This results in a higher level of abstraction which facilitates migration of the system to new platforms. We refer to this activity as componentization.
- The second challenge is to divide the tasks into sets which should execute concurrently, to optimize the performance. We refer this activity as partitioning and each set as a *partition*. The partitioning is orthogonal to the componentization.

Componentization groups the tasks with respect to their logical relation from a design point of view, i.e., in subsystems. Partitioning specifies which tasks can run concurrently and also which tasks run on each core, to improve the performance. The policy that groups tasks into partitions can differ depending on the target platform as well as the nature of the application. Componentization and partitioning may intersect each other as these concepts are orthogonal; a component may include tasks from several partitions, and a partition may include tasks from several components/subsystems (Figure 2).

Both approaches need information in the form of task models of the legacy system, e.g., dependency between tasks. However, legacy systems generally are very complex, developed and maintained by many developers, many of them may no longer be available, and documentation is often not complete or out-dated. Reverse engineering techniques, static and dynamic analysis, are needed to extract required task models, which will contain information about dependency among tasks, and their upper bound resource budgets. These techniques will assist developers in componentization and partitioning. Currently we are investigating existing methods and possibilities to extend them to be suitable for componentization and partitioning of the legacy systems.

3.2. Specification of the Framework

Each task is a member of exactly one component and a member of at most one partition. If a task is not a member of a partition, it is allowed to run on any core. The membership of tasks in components and partitions is depicted in Figure 2, in which a box represents a component, a dashed oval is a partition and a circle represents a task.

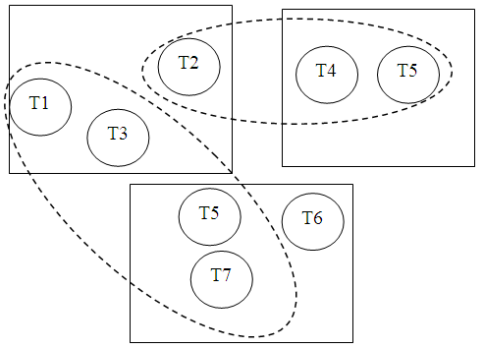

Figure 2: Task Model

To identify the component and partition containing a task we present concepts of Partition Task ID (PTID) and Component Task ID (CTID). The idea is influenced by concept Related Thread ID (RTID) presented in [13].

All tasks that are members of a component will have a common CTID and all tasks that are members of a partition will have a common PTID. While CTIDs will remain unchanged, PTIDs may change if the policy of grouping tasks together is changed or if the hardware is changed, e.g., number of cores are increased. A partitioning policy is used to assign PTIDs to the tasks, which ensures that timing constraints are met. Finally, an interface has to be developed and added to the legacy system to translate the tasks to the operating system based on their PTIDs. Most of the exiting thread library implementations support assigning tasks to specific cores and migrating them [7]. Figure 3 depicts a framework for componentization and partitioning. The boxes show activities and the ovals show data.

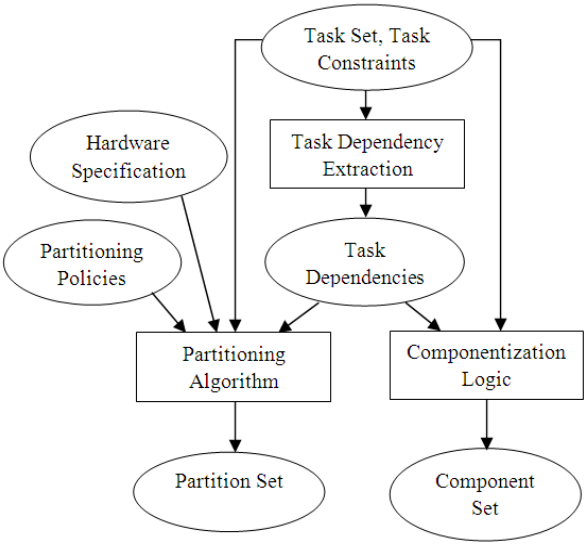

Figure 3: A Framework for Partitioning and Componentization

4. Conclusions and Future Work

In this paper we have proposed a framework for migrating legacy real-time systems to multi-core processors, which includes elements of componentization and partitioning. Componentization will result in a set of components containing logically related tasks which captures a higher level abstraction of the system. Partitioning will result in a set of partitions containing tasks to be run concurrently to maximize process performance.

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 can be used to extract required information from the legacy system, e.g., the use of shared resources. We also work on developing formal task model description languages that suits our framework.

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.

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] S. Comella-Dorda, K. Wallnau, R. C. Seacord, and J. Robert. A Survey of Legacy System Modernization Approaches. *Technical Note, CMU/SEI-2000-TN-003, Carnegie Mellon University,* April 2000.

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

[4] 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.

[5] K. Johnson, and R. Saha. Secure Partitioning for Multi-Core Systems. *QNX Software Systems GmbH & Co. KG*, 2007.

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

[7] B. Lewis, and D.J. Berg. Multithreaded Programming With PThreads. *Prentice Hall*, 1998.

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

[9] 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.

[10] F. Luders, I. Crnkovic, and A. Sjögren. Case study: componentization of an industrial control system. *In Proceedings of 26th International Computer Software and Applications Conference (COMPSAC), pages 67-74*, 2002.

[11] 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.

[12] 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.*

[13] 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)*, May 2007.

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