Potentials and Challenges for Multi-Core Processors in Robotic Applications *

Andreas Herkersdorf, Johny Paul, Ravi Kumar Pujari, Walter Stechele, Stefan Wallentowitz, Thomas Wild, Aurang Zaib

Technische Universität München, Institute for Integrated Systems, D–80290 München
{herkersdorf, johny.paul, ravi.kumar, walter.stechele, stefan.wallentowitz, thomas.wild, aurang.zaib}@tum.de

Abstract. Multi-core processors have shown to be superior to single-core with respect to performance and power efficiency. However, multi-core imposes additional challenges on system complexity and application programming. This paper reviews benefits and challenges of multi-core processors in embedded real-time applications like humanoid robotics. Selected approaches towards enabling multi-core processors are shown, covering multiple hardware/software abstraction levels, including isolation of individual applications, differentiated quality-of-service support, thread mapping, and resource-aware programming.

Keywords: multi-core processors, robotic control systems, embedded systems, hardware assist, non-functional requirements, virtualization, OS event handling and threading, resource-awareness, algorithm morphing

1 Introduction

Multi-core processors are a key enabling technology for tackling the important challenges our society will face in the upcoming decades. Safe and sustainable mobility, comprehensive healthcare, efficient energy management and the development of a secure digital society pose immense demands on embedded information and communication technology (ICT) platforms, which hardly can be satisfied without multi-core technology. All leading processor vendors ARM, Freescale, IBM, Intel, MIPS, Nvidia, TI - pursue a strategy towards multi-core architectures. Multi-core processors are superior to their single-core ancestors with respect to processing performance and power efficiency, as they can execute multiple tasks concurrently on less complex, lower frequency but parallel processor cores.

On the other hand, multi-core technology makes industry and academia face entirely new challenges with respect to system complexity. The efficient utilization of parallel processing resources currently relies predominantly on the individual skills of the programmers. In the field of embedded and cyber physical

* This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) and the Priority Program "Dependable Embedded Systems" (SPP 1500).

systems, multi-core processors must adhere to much stronger real-time, power efficiency, reliability, safety and security requirements than found with standard desktop machines. Furthermore, multi-core test and debugging tools are often missing along with methods for the universal modeling, design and validation of multi-core based systems.

This paper argues that the use of multi-core technology in embedded systems applications in general, and in robotics as a demanding example of such systems in particular, has many advantages but also requires an enablement across all hardware and software abstraction layers in order to be effective.

2 Robot Control System

This section will present the challenges involved in multi-core programming in the context of humanoid robots. Todays humanoid robots share similar hardware and control architectures consisting of one or more industrial PCs equipped with single- or dual-core processors. For e.g. the architecture of the humanoid robot Asimo uses two PCs, a control and planning processor plus an additional digital signal processor (DSP) for sound processing [13]. Similarly, two processor boards (one for realtime-control, one for non-realtime tasks) were used on the humanoid robots HRP-2 [7] and HRP-4C [8]. The Hand Arm System from DLR [6] has 3 layers of computing hierarchy. The top layer consists of COTS PCs running real-time OS for control applications. The real-time hosts are augmented by auxiliary Linux workstation for user interfaces Below this, the composition layer constituted by FPGAs for HW accelerated tasks. The ARMAR-III robot consists of numerous sensors like pressure, accelerometers, torque, rotary encoder, image senors for vision, and microphones for audio. The data from the sensors flow into the processing system that consists of five industrial PCs and PC/104 systems built into the robot (Figure 1), each dedicated to different tasks like computer vision, low-level control, high-level control, and speech processing [1]. Using these processors, the humanoid robots can handle different tasks including recognizing objects, motion planning, self navigation, speech recognition and synthesis, etc. Once the a task is assigned and necessary input data is processed, the appropriate command are issued to the motors through low level communication modules (UCoM). The compute units as shown in Figure 1, are stacked inside the torso of the robot, occupying considerable space. The multiple CPU boards and I/O interfaces result in high operating power. The communication bandwidth between PCs is limited to the peak bandwidth offered by the I/O interfaces like Ethernet. The various applications running on the robot posses different resource requirements depending on their objectives and size of data to be processed. Some are continuously running at a specific frequency while others are doing their work asynchronously triggered by external events. For example, the speech processing is triggered when people in close proximity speak to the robot, the vision applications are required only when the robot needs to recognize objects around it, and the motor control is activated when the robot has to move or grasp an object.

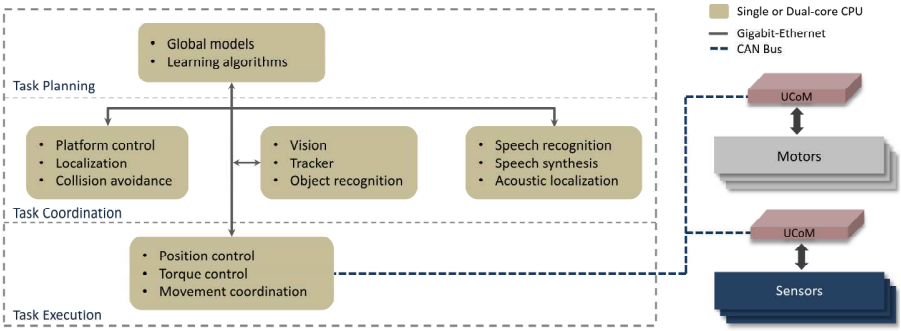
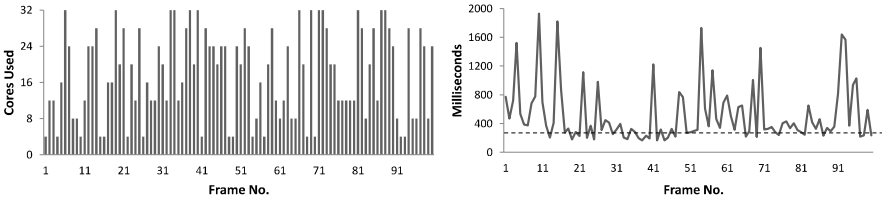


Fig. 1. Various compute nodes and application mapping on Armar-III Robot

In general, these applications create dynamically changing loads on the processor resources based on what the robot is currently doing. A static mapping of functions to processor boards typically results in low overall resource utilization as the applications do not run continuously but the boards were dimensioned based on peak performance requirements. Allowing applications to share processing resources on a multi-core platform with sufficient number of cores could reduce the overall power consumption, improve resource utilization, and offer higher communication bandwidth as all the data exchange is now on-chip. However, the fluctuating resource requirements of different applications over time may also cause shortages of processing memory or interconnect resources which lead to unpredictable execution time, data loss or generally deteriorated overall behavior of the humanoid robot. Such an exposure is demonstrated now with the example of real-time video object recognition and tracking based on a combination of Harris Interest Points and SIFT feature description algorithms as described in [2]. For fast moving objects the search interval has to be reduced so that the object can be tracked accurately. The actual duration of the search depends on the size of the data set to be processed. The data set size depends on the number of objects present in the scene, the structuring of the background, the lighting conditions, etc. The conventional OS scheduler schedules the threads of each application considering the overall system load. Such a situation is depicted in Figure 2(a), where the y-axis represents the number of cores allocated to object tracking application per frame for a total of 100 frames. The resulting execution time is depicted in Figure 2(b). These results indicate significant variations in the number of required processing cores and high jitter in the execution time. As a consequence, standard multi-core processors reveal a highly unpredictable search durations leading to frame drops and tracking errors. The number of frames dropped during the depicted evaluation was as high as **44%**. The robot even lost track of the object if too many consecutive frames were dropped. Moreover, the individual robotic application each operate on large amounts of data and the overall execution times are highly dependent on the latencies to on-chip and off-chip DDR memory.



(a) Dynamic resource usage

(b) Execution time

Fig. 2. NN-Search on multi-core processor

3 Multi-core Enablement

The above example has shown that a multi-core processor may help to overcome the space restrictions, monetary cost and power related challenges of robotic control by integrating a sufficient number of cores on a single device. Nevertheless, the naïve and straightforward mapping of tasks from mixed-criticality robotic applications onto a multi-core processor may result in suboptimal, or even worse system-level performance than with separate processor boards. A major reason for this performance degradation is the increase in resource conflicts and possible priority inversion during accesses to memory or network i/o. Since variations of workload and event occurrence in embedded systems are heavily runtime dependent, it is impossible to consider such aspects during software development at design time. The following sections provide examples for generic enablements at different hardware/software abstraction levels. These enablements are provisioned to mitigate the lack of isolation or to provide differentiated services support. Finally, the advantages of a resource-aware allocation of system resources in a standard multi-core platform are demonstrated.

3.1 I/O Virtualization & NoC traffic management

I/O Virtualization Virtualization is a well established approach to abstract the underlying hardware platform of a host system and allow the execution of different independent guest systems in an isolated manner under the management of a hypervisor or VMM (virtual machine monitor) [3, 4]. The hypervisor makes up a middleware layer, which manages the shared usage of the underlying hardware architecture with its processing, memory and communication resources by the guest operating systems and their processes/threads. Multiple OS domains can thus run on one or more cores of a multi-core CPU with the impression to exclusively use a processor with a certain fraction of the overall compute performance. Virtualized system memory is segmented for individual OS domains, i.e. different memory page tables are used. The interconnect bandwidth of a network on chip (NoC) can dynamically be assigned to different domains or applications (see below). I/O-devices, such as network, audio and video interfaces can be either excursively assigned to domains, or concurrently shared among domains.

In a conventional, hypervisor-based communication virtualization scenario, general purpose processor cores are still in the communication data path. For multi-core architectures with many independent OS domains of mixed criticality like in robotics, a purely software based and central management of shared resources via hypervisor is not scalable. Therefore, hardware assistance within the shared resources to offload the hypervisor is considered as an efficient approach to avoid bottlenecks and to increase error resilience. In the following, such an offload is described using the example of I/O virtualization of an Ethernet interface in a tiled multi-core architecture for embedded applications with mixed criticality (see Figure 3).

Objective is to achieve a balance between high data throughput and QoS combined with realtime [11]. The major part of the hardware offload is realized in the Virtualized Interface Network Controller (VNIC), which is accessible from the different domains (i.e. compute tiles) via the Virtualized Network Adaptor (VNA).

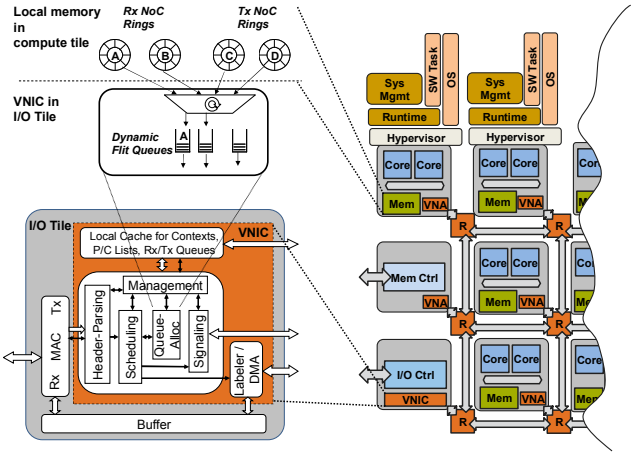
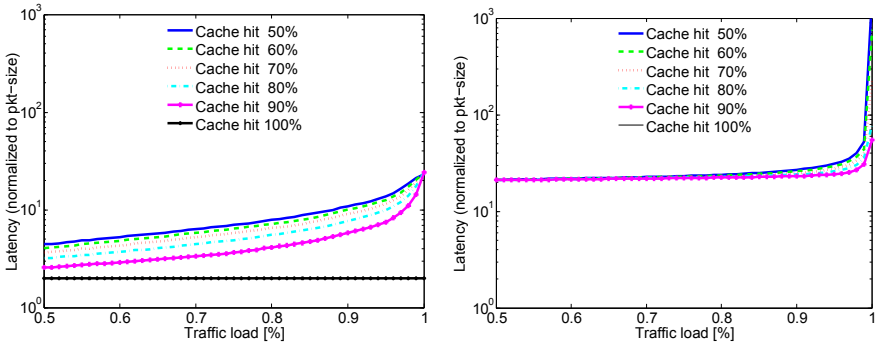


Fig. 3. Virtualized I/O interface in a tiled multi-core architecture

VNIC relies on a tailored set of finite state machines (FSMs) specifically designed for the support of realtime constraints and service levels of domains. It transfers for each configured virtual I/O interface incoming/outgoing Ethernet packets autonomously without involvement of a hypervisor to/from the receive/-transmit buffers in the memory associated to the corresponding domain (within a compute tile or in the external memory). To meet realtime requirements and yet maintain scalability, the context of virtual interfaces (interface configuration, buffer descriptor) belonging to realtime domains is permanently stored within VNIC. For domains requiring only best effort communication, this information is held in system memory and only a limited number of entries are cached in VNIC for active connections, which can dynamically be replaced on demand.

The approach was validated with a MATLAB SimEvents model capturing the different treatment of realtime and best effort [5]. Simulations were done for an internet traffic mix (IMIX) with varying loads and cache hit rates of context for best effort traffic. The realtime traffic was assumed to be 20% of the overall load.



(a) Latency for realtime packets

(b) Latency for best effort packets with non-cached context

Fig. 4. VNIC simulation results [5]

The results in Figure 4 show that the transfer latencies in the VNIC for high priority and best effort traffic are bounded for loads up to 90%. VNIC is capable to support realtime applications like interactive multimedia (Figure 4(a)) and also the latency of traffic with uncached context is very insensitive to the overall load (Figure 4(b)). However, it exhibits a base delay for fetching the context from main memory. Under heavy traffic loads (beyond 95%) the system behaves as expected from an input queuing system. The simulations also show that blocking context fetch requests over the shared NoC adapter introduce additional latency, which is however insignificant with cache miss rates of up to 50%.

In general, with this approach the area requirement of I/O virtualization in the hardware is reduced and the constraints of realtime domains can be met. In case a domain (or task) with an associated virtual I/O interface is moved to a different compute tile, e.g. for reasons of error resilience, VNIC can be reconfigured to seamlessly switch over the associated NoC connection to the new compute tile [12].

NoC traffic management NoCs are distributed communication systems and hence are more scalable for multi-/many-core architectures as compared to centralized bus-based systems. However, one drawback of NoCs is that the transfer latency between different partner tiles and towards shared memory and I/O tiles varies due to the naturally different hop counts. Therefore, strategies are required

which mitigate the latency variance to meet the communication requirements of individual applications while ensuring efficient NoC utilization.

In real world scenarios, it is observed that most of the applications exhibit temporal locality in terms of their communication patterns. This behavior leads to end-to-end communication where a source node communicates with a certain sub-set of destination nodes in the architecture. For example, the robotic applications as described in section 2 perform frequent accesses to memory for fetching image data. However, it is difficult to analyze their access patterns before run-time because they are dependent on other parameters such as application mapping, memory hierarchy and background traffic etc. Therefore, a concept was adopted, which manages NoC traffic by observing the communication behavior of applications.

The proposed concept is based on the Network Adapter (NA) design of a tiled architecture in which tiles are connected through a Virtual Channel (VC) based Network-on-Chip. NA can establish virtual connections between source and destination tiles over NoC. Once the virtual connection is established, the communication between source and destination happens in a streaming manner. In addition, packet switching based Best Effort (BE) communication is also supported over NoC.

In the scenarios where the applications communicate with a certain subset of tiles, virtual connections offer following advantages as compared to BE communication by reducing the packet switching overhead:

- Reduced latency for the applications
- Reduced energy consumption for data transmission

Our concept enables to use virtual connections for the applications which show high communication locality to certain destination tiles. In our approach, the communication behavior of applications running on the tile is monitored inside the NA. On the basis of this monitoring information, the NA gains the knowledge about the temporal locality of communication requests. Afterward, NA exploits this information to establish virtual connections to the tiles which are frequently accessed. The tiles which are accessed less frequently are still served with BE communication. In this way, our concept optimizes the network resource utilization according to the communication patterns at run-time [16]. The block diagram of the network adapter with required extensions is shown in Figure 5.

Among the NA common components, the *VC Reservation Table* contains the identifiers (ids) of destination tiles to which a virtual connection exists. *Address Lookup* checks for the existence of a reserved connection in each incoming request from the tile. In both cases, *Packetization* unit is triggered to generate flits/packets for virtual connection/BE traffic respectively. Afterward, the data is placed in the respective FIFOs. The *Scheduler* unit is responsible for scheduling the different FIFOs over the NA output link. Among the introduced hardware extensions, monitoring of communication temporal locality is done by *Communication Monitoring Unit. Communication History Table* sorts the target tiles in the order of highest communication locality. Management of virtual connections is performed by *Virtual Connection Manager.*

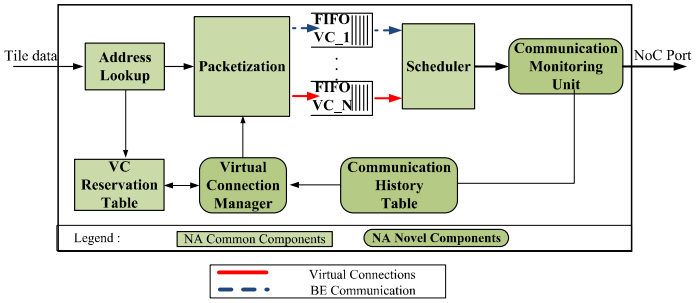


Fig. 5. NA with run-time traffic management support

3.2 Threading

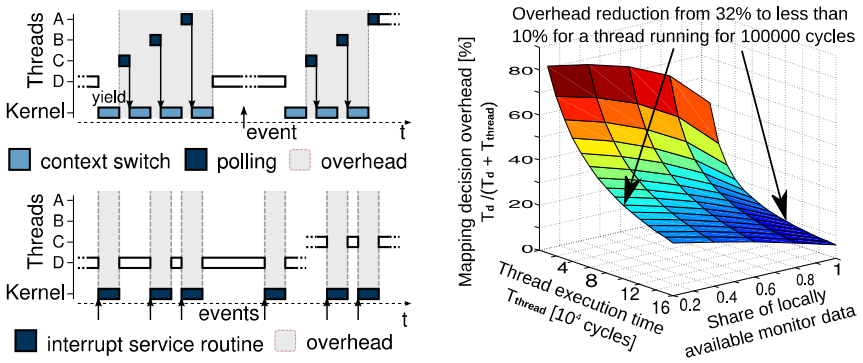
Potential for further improvement of multi-core System-on-Chip by hardware enablements is in multi-threading. Concretely, we address the problems occurring during a) *Event Handling* and b) *Thread Mapping* in a multi-threaded system.

Impact of event handling Traditionally, event completion was signaled either by polling for the completion from the application or by suspending the thread and waking it up by an interrupt service routine. Both ways have disadvantages with respect to the software performance (compare Figure 6(a)). In polling, the number of waiting threads significantly increases the non-productive processing time. When using interrupts, the number of events directly determines the time spent in the interrupt service routine, that is the reason interrupts are commonly used nowadays. Due to the restricted memory sizes and reduced latencies, emerging applications will exchange data more often, so that the events, such as from message passing, DMA or similar, will drastically increase.

Overhead in thread mapping Another major issue in a multi-core environment is the mapping of tasks to the available cores. For this the operating system generally probes hardware monitors for the current availability and operational status of processing, communication and memory resources. A typical example code of such a decision logic would be:

```
IF((avg_temp(Core[i]) < 50 C) AND
    (max_load(Core[i]) < 20 %))
        select Core[i]
ELSE IF((max_hops(Core[j]) < 5 hops) OR
    (energy_budget(Core[j]) > 20 Wh))
        ...
```

This probing and decision making done by the software layers accounts for a considerable amount of overhead in the task/thread mapping. In [10], we analytically modeled this decision process for the case of mapping threads to cores in immediate neighborhood tiles on a single hop distance over NoC. Estimates



(a) Event signaling scalability issues [15]

(b) Reduction in thread assignment decision overhead on prefetching the monitor data of cores from neighborhood [10]

Fig. 6. Scalability issues for event handling and thread mapping on multi-core platforms

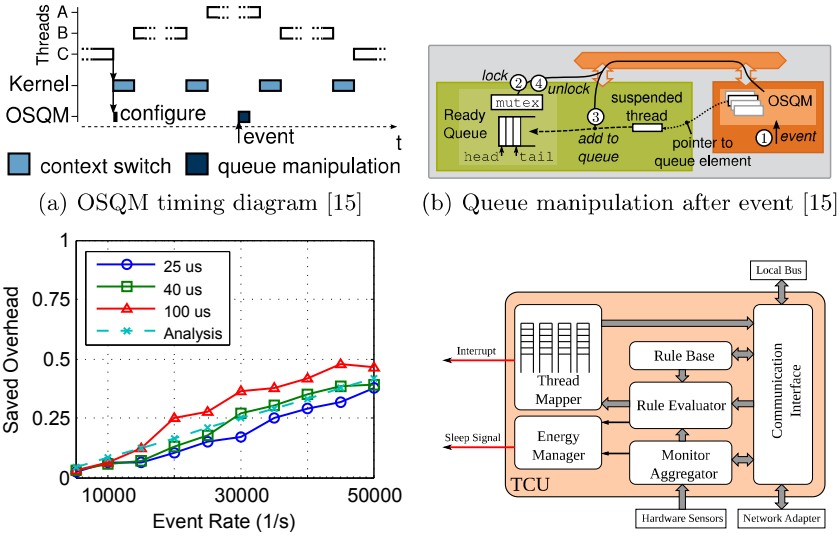
from the model show this overhead is above 30% as depicted in Figure 6(b) for threads with 100 Kcylces runtime. In the following sections we address these two issues by means of hardware enablements for a multi-core platform.

OSQM In [15], we present a technique to improve the impact of event handling in multithreaded systems. Having a look at the general implementation of the parts of an interrupt service routing in multithreaded systems two fundamental parts are identified: On the one hand there are the context switches of the threads and the interrupt service routine, which are most costly. On the other hand, there is the real task of the interrupt service routine, that is essentially to append the waiting thread to the queue of runnable threads (ready queue). Approaches to reduce the impact of interrupts on application processing include coalescing interrupts or adaptively change between polling or interrupts. Other approaches go beyond that and implement significant parts of an operating system in hardware, such as the scheduler.

The central idea of *hardware-based operating system queue manipulation (HW-OSQM)* is to offload the operating system queue manipulation to hardware. On the occurence of an event, the thread waiting for the event needs to be appended to the queue of runnable threads. The basic principle (Figure 7(a)) and the operations of appending a thread to the ready queue are depicted in Figure 7(b). The operation itself is relatively short and only requires a few bus operations. In case the bus cannot perform these operations atomically, the access to the data structure needs to guarded by a mutex lock. This scheduler ready queue is implemented in software as a linked list or queues.

The impact of HW-OSQM is compared to polling and interrupts in a RISC-based platform with a minimalistic runtime system. For both polling and inter-

rupting, HW-OSQM can save the entire non-user processing time and can be glued to any conventional operating systems. The saved overhead in processing compared to interrupts is depicted in Figure 7(c). OSQM can also be integrated with hardware thread queues, as the TCU described subsequently.



(a) OSQM timing diagram [15] (b) Queue manipulation after event [15]

(c) Improvement of OSQM over IRQ [15] (d) Thread Control Unit[10]

Fig. 7. Improvement of OSQM and Thread Control Unit

Thread Control Unit To minimize the delay due to overhead in thread mapping, we propose in [10] to implement
i) global, coarse level thread mapping decisions in software for flexibility and scalability reasons.
ii) finer, localized up-to-date monitors based thread mapping in hardware.

This balanced software-hardware partitioned approach leads to an overall reduction of the overhead to less than 10% as depicted in Figure 6(b). Of course this reduction comes at a cost of enabling the multi-core platform with dedicated hardware blocks called thread control units. TCUs, shown in Figure 7(d), are provisioned within each tile to take care of the finer, localized thread assignment to the cores. Using the TCU the functional partitioning of thread assignment is done as follows:

- The OS identifies a target tile by using the status information of the neighborhood region readily available in the TCU's monitor aggregator block.
- TCU pro-actively fetches, aggregates and distributes these abstracted monitor data such as temperature, power consumption, NoC link load, core uti-

lization and ready queue fill levels etc, thereby hiding the latencies in monitor data collection.
- The thread mapper consists of an array of HW FIFO's per core as ready queues. Threads either coming from NoC or triggered by HW-OSQM on event completion are appended atomically to one of these FIFOs. Thread mapper selects the FIFO based on to up-to-date monitor values much faster and without any involvement of OS. The assignment decisions though can still be influenced by OS for power, performance optimization by means of rules set in the rulebase. Monitor data is locally stored and evaluated within the TCU, thus supporting a higher thread assignment rate.
- An energy manager collects selected, energy-related monitor data (e.g. load-/temperature values) from the cores. It influences the mapping decisions of the thread mapper to trigger thread migration and also generates sleep signals to power gate the cores on temperature or power overshoots or when a core is not in use.

Variations of HW-OSQM and TCU FIFO management, such as support for different priority levels, other scheduler implementations and urgent events [15], can also be built at the cost of addition hardware design complexity. The current prototype implementation for OSQM and TCU on FPGA has low area demands (less than 4% of a core) and is easy to integrate with any existing RISC based multi-core platforms.

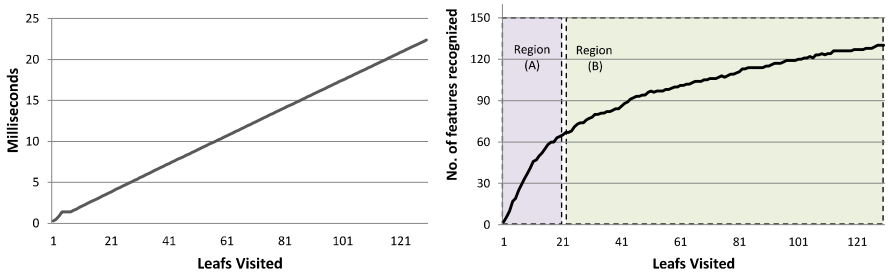
3.3 Resource Aware Programming

Section 2 described various challenges involved in using multi-core platforms in complex environments like humanoid robots. The sharing of resources leads to unpredictable execution time that affects the overall behavior of the robot. However, the execution time can be equalized by adding more cores to the application when higher computing power is required and vice versa. In other words, the application program and the runtime-system have to work together so that a certain output quality can be guaranteed. This can be achieved by exposing the application requirements to the runtime-system so that the runtime-system can allocate the resources accordingly. In order to address these issues [9] proposed a new resource-aware operating system (ROS) for large multi-core HW, with direct support for parallel applications and a scalable kernel. In that work, resources such as cores and memory are explicitly granted to the applications and revoked. The kernel exposes information about a process's current resource allocation and the system's utilization, and allows the application programs to make requests based on this information. The demand for more stringent resource awareness was also proposed in [14], put forward by a new programming methodology called *Invasive Computing*. The main idea and novelty of Invasive Computing is that it extends resource-aware programming support to various layers in the multi-core system like resource-aware OS, communication interfaces like NoCs, etc. This research also focuses on policies for resource allocation and when and how to revoke resources from a process. This remaining of this

section will demonstrate the benefits of resource aware programming using the NN-search on kd-trees.

For the NN-search algorithm, the number of kd-tree leaves visited during the search process determines the overall quality of the search process. Visiting more leaf nodes during the search leads to a higher execution time. The search duration per SIFT feature can be calculated from Figure 8(a). The values were captured by running the NN-search application on a single core, using a library of input images covering various situations encountered by the robot. Moreover, the relation between quality (no. of features recognized) and leaf nodes is shown in Figure 8(a). The quality of detection falls rapidly when the number of leaf nodes is reduced below 20 and increases linearly in the range between 20 and 120. At a further higher leaf count, the quality does not improve significantly as all the possible features are already recognized. In the conventional algorithm used on CPUs, the number of leaf nodes visited is set statically such that the search process delivers results with sufficient quality for the specific application scenario. In order to avoid frame drops and to improve the tracking accuracy, the conventional NN-search can be modified to process the SIFT features based on their quality. Once the deadline is hit, the algorithm drops the remaining low-quality features and move on to the next frame. This technique is relatively simple, easy to implement and works on any single-core or multi-core platform. The algorithm would perform well in scenarios where the scene contains only the object to be recognized and all the detected features belong to the same object. The results deteriorate when there are more objects in the frame and also in scenes with cluttered background; this is because some of the high-quality SIFT features may belong to other objects or to the background. Therefore, the features dropped by the algorithm may belong to the target object, leaving it undetected.

The above mentioned issues can be addressed through a resource-aware programming model as described below. The first step, in the resource-aware model is to allocate sufficient resources to perform a parallel NN-search. The number of



(a) Variation of execution time vs. leaf nodes visited

(b) Search quality vs. leaf nodes visited

Fig. 8. NN-Search algorithm

cores requested by the algorithm is based on the number of SIFT features to be processed, the size of the kd-tree and the available search interval. The number of SIFT features varies from frame to frame based on the nature and number of objects present in the frame, the nature of the background, etc. The size of the kd-tree is decided by the texture pattern on the object to be recognized and tracked. The search interval or the frame rate is decided by the context where the NN-search is employed. Equation (1) represents this relation and can be used to compute the number of cores (N_{cores}) required to perform the NN-search on any frame within the specified interval T_{search}. N_{fp} is the number of SIFT features to be processed and T_{fp} is the search duration per SIFT feature, a function of the number of leaf nodes visited, as described in Figure 8(a). The initial resource estimate is based on the fixed leaf count (N_{leaf_best}), which results in an optimum match count for the particular object under consideration. Every additional thread created by the NN-search algorithm also creates an additional load on the external memory and shared communication interfaces, limiting the scalability. In order to improve the accuracy of the resource-estimation model, an application-specific efficiency factor or scalability information $\eta(N_{cores})$ has been added.

$$N_{cores} \geq \frac{N_{fp} \times T_{fp}(N_{leaf\_best})}{T_{search} \times \eta(N_{cores})} \tag{1}$$

Using the new model, the application raises a request to allocate cores (N_{cores}), which is then processed by the operating system. Considering the current system load, the OS makes a final decision on the number of cores to be allocated to the NN-search algorithm. The core count may vary from zero (if the system is too heavily loaded and no further resources can be allocated at that point in time) to the total number of cores requested (provided that there exists a sufficient number of idle cores in the system and the current power mode offers sufficient power budget to enable the selected cores). This means that under numerous circumstances the application may end up with fewer cores and has to adapt itself to the limited resources offered by the runtime system. In constrained scenarios as explained above, the application has to re-balance the workload in order to complete the NN-search within the search interval specified by T_{search}. This is done by recalculating the number of leaf-nodes to be visited during the NN-search such that the required frame rate can be achieved. However, it can be seen from Figure 8(b) that the quality drops significantly if the number of leaf nodes calculated is too low (between 0 and 20, for the particular object used in our evaluation). This region is marked as region(A) in the figure. This issue can be resolved by preventing the leaf count from falling into region(a), as shown in Figure 8(b). In this case, the NN-search will drop a few low-quality SIFT features to complete the search within the predefined search interval.

In order to demonstrate the difference, the resource-aware NN-search algorithm is compared with the conventional search techniques. A set of 100 different scenes was used for evaluation, where each frame contains the object to be recognized and localized along with few other objects and changing backgrounds. The position of the objects and their distance from the robot were varied from frame

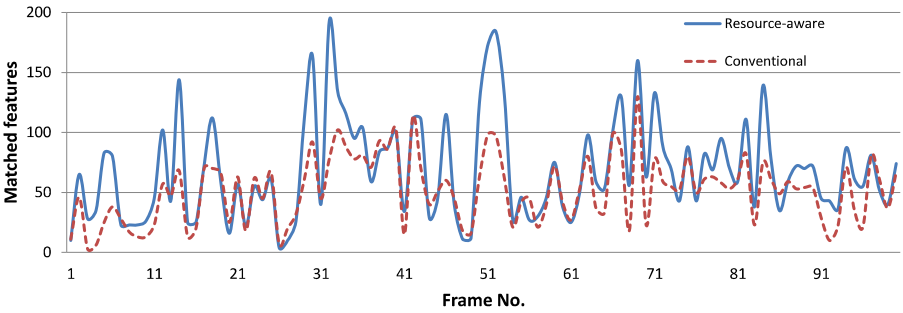


Fig. 9. Comparison between resource-aware and conventional NN-search

to frame to cover different possible scenarios. It is clear from Figure 9 that the resource-aware NN-search algorithm outperforms the conventional algorithm using the same amount of resources. This is because the resource-aware model is capable of adapting the search algorithm based on the available resources compared to the conventional algorithm with fixed thresholds. However, the resource-aware algorithm results in the same number of matched features as the conventional algorithm in some frames. This is because there were a sufficient number of idle cores and the runtime system allocated sufficient resources to meet the computing requirements of the conventional algorithm and hence the conventional algorithm did not drop any SIFT feature. On the contrary, when a frame contains large number of SIFT features and the processing system is heavily loaded by other applications, the conventional algorithm dropped too many SIFT features, thereby resulting in a low overall detection rate (matched features). This result points to the ability of the resource-aware application to adapt itself to changing load conditions and generate better results in tightly constrained situations. Our experiments with similar robotic applications shows considerable improvements in the quality of the overall results and we believe that adding such resource-aware mechanisms into the existing applications can lead to improved results and better response times for other embedded applications as well.

4 Conclusion

Multi-core processors are superior to their single-core ancestors with respect to processing performance and power efficiency, as they can execute multiple tasks concurrently on less complex, but parallel processor cores operating at lower voltage and frequency levels. Conflicting accesses to shared components and lack of isolation for concurrent applications with different safety, security or quality of service requirements can degrade system performance. Moreover, integrity of functional and non-functional behavior of applications might be compromised in an unacceptable way. Leaving the responsibility for coping with these artifacts with the application programmers easily overwhelms their capabilities and ca-

pacities because of the overall system complexity. This paper brought forward four examples of generic enablement of multi-core technology, which can be exploited by a wide range of embedded systems applications. We neither argue that the presented examples provide an exhaustive solution for all conceivable problems, nor are they the only way to approach these challenges. They are rather meant as an initial step into multi-core enablement and, thus, may serve as candidates for being adapted – in one form or the other and to different degrees – in future multi-core architectures.

Acknowledgment

The authors would like to thank T. Asfour and M. Kröhnert from KIT for the detailed information on ARMAR robot control structure and applications, J. Becker and J. Heisswolf for NoC architecture, W. Schröder-Preikschat, D. Lohmann, C. Erhardt, B. Oechslein and J. Schedel from FAU for the collaboration on hardware/software OS assists within the DFG SPP/TRR 89 program "Invasive Computing" and H. Rauchfuss from TUM, J. Henkel and T. Ebi from KIT for contributions and cooperation within VirTherm-3D project of DFG SPP 1500 program "Dependable Embedded Systems".

References

1. T. Asfour, K. Regenstein, P. Azad, J. Schroder, et al. ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pages 169 –175, Dec 2006.
2. P. Azad, T. Asfour, and R. Dillmann. Combining harris interest points and the sift descriptor for fast scale-invariant object recognition. In *Intelligent Robots and Systems, 2009. IROS 2009.* IEEE, 2009.
3. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.
4. G. Heiser. Hypervisors for consumer electronics. In *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, CCNC'09, pages 614–618, Piscataway, NJ, USA, 2009. IEEE Press.
5. A. Herkersdorf, H. Rauchfuss, T. Wild, et al. Multicore enablement for automotive cyber physical systems. *it - Information Technology*, 54(6):280–287, 2012.
6. S. Jorg, M. Nickl, A. Nothhelfer, et al. The computing and communication architecture of the dlr hand arm system. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 1055–1062. IEEE, 2011.
7. K. Kaneko, F. Kanehiro, S. Kajita, H. Hirukawa, et al. Humanoid robot HRP-2. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 2, pages 1083 – 1090, may 2004.
8. K. Kaneko, F. Kanehiro, M. Morisawa, K. Miura, S. Nakaoka, and S. Kajita. Cybernetic human HRP-4C. In *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, pages 7 –14, Dec 2009.
9. K. Klues, B. Rhoden, Y. Zhu, A. Waterman, and E. Brewer. Processes and resource management in a scalable many-core os. *HotPar10, Berkeley, CA*, 2010.

10. R. K. Pujari, T. Wild, A. Herkersdorf, B. Vogel, and J. Henkel. Hardware assisted thread assignment for risc based mpsocs in invasive computing. In *Integrated Circuits (ISIC), 2011 13th International Symposium on*, pages 106–109, 2011.
11. H. Rauchfuss, T. Wild, and A. Herkersdorf. A network interface card architecture for i/o virtualization in embedded systems. In *Proceedings of the 2nd conference on I/O virtualization*, WIOV'10, pages 2–2, Berkeley, CA, USA, 2010.
12. H. Rauchfuss, T. Wild, and A. Herkersdorf. Enhanced reliability in tiled manycore architectures through transparent task relocation. In G. Mühl, J. Richling, and A. Herkersdorf, editors, *ARCS Workshops*, volume 200, pages 263–274, 2012.
13. Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, et al. The intelligent ASIMO: system overview and integration. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2478 – 2483, 2002.
14. J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive Computing: An Overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and ToolIntegration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.
15. S. Wallentowitz, T. Wild, and A. Herkersdorf. Hw-osqm: Reducing the impact of event signaling by hardware-based operating system queue manipulation. In H. Kubtov, C. Hochberger, M. Dank, and B. Sick, editors, *Architecture of Computing Systems ARCS 2013*, volume 7767 of *Lecture Notes in Computer Science*, pages 280–291. Springer Berlin Heidelberg, 2013.
16. A. Zaib, J. Heisswolf, A. Weichslgartner, T. Wild, J. Teich, J. Becker, , and A. Herkersdorf. Auto-gs: Self-optimization of noc traffic through hardware managed virtual connections. In *Proceedings of the 2013 16th Euromicro Conference on Digital System Design. DSD 13. to appear*, Santander, Spain, Sep 4-6 2013. DSD.