Journal of Systems Architecture 61 (2015) 520–530

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Self-adaptive corner detection on MPSoC through resource-aware programming

Johny Paul [a,*], Benjamin Oechslein [b], Christoph Erhardt [b], Jens Schedel [b], Manfred Kröhnert [c], Daniel Lohmann [b], Walter Stechele [a], Tamim Asfour [c], Wolfgang Schröder-Preikschat [b]

[a] Technical University of Munich, Germany
[b] Friedrich-Alexander University Erlangen-Nuremberg, Germany
[c] Karlsruhe Institute of Technology, Germany

ARTICLE INFO

Article history:
Received 9 October 2014
Received in revised form 5 May 2015
Accepted 1 July 2015
Available online 26 July 2015

Keywords:
Corner detection
Resource-aware programming
Invasive Computing
Self-adaptive algorithms
Computer vision

ABSTRACT

Multiprocessor system-on-chip (MPSoC) designs offer a lot of computational power assembled in a compact design. In mobile robotic applications, they offer the chance to replace several dedicated computing boards by a single processor, which typically leads to a significant acceleration of the computer-vision algorithms employed. This enables robots to perform more complex tasks at lower power budgets, less cooling overhead and, ultimately, smaller physical dimensions.

However, the presence of shared resources and dynamically varying load situations leads to low throughput and quality for corner detection; an algorithm very widely used in computer-vision. The contemporary operating systems from the domain have not been designed for the management of highly parallel but shared computing resources.

In this paper, we evaluate *resource-aware programming* as a means to overcome these issues. Our work is based on *Invasive Computing*, a MPSoC hardware and operating-system design for resource-aware programming. We evaluate this system with real-world algorithms, like Harris and Shi–Tomasi corner detectors. Our results indicate that resource-aware programming can lead to significant improvements in the behavior of these detectors, with up to 22 percent improvement in throughput and up to 20 percent improvement in accuracy.

1. Introduction

Humanoid robots are required to handle various tasks like vision, motion planning, speech recognition, or speech synthesis. Usually, the workload is spread across multiple industrial CPU boards. For example, on the ARMAR-III robot [1], data from the sensors flows into the respective processing system, each dedicated to a different task, like computer vision, low-level control, high-level control or speech processing, as shown in Fig. 1. Other examples with a similar architecture include the humanoid robot Asimo [2], HRP-4C [3] and the Hand Arm System from DLR [4]. Such a static multi-computer mapping scheme is employed to reduce the interference between the separate parts of the robot programming, so that algorithms in each step can produce consistent results regardless of the behavior of the other parts of the system. The use of multiple CPU boards results in high power consumption and low interconnect bandwidth, and occupies a large amount of space in the robot.

The use of MPSoC hardware can mitigate some of the above mentioned problems on account of their immense computational power assembled in a compact design. This means the various applications that, on a conventional robot, run on dedicated CPU boards can now be combined and executed on a single-chip. On the one hand, this results in higher communication bandwidth and lower latency [5] between the single processing units, lower power consumption and, thus, reduced cooling costs. On the other hand, mapping various tasks onto a unified MPSoC automatically results in sharing of physical resources, like processing elements (PEs) and interconnect/memory bandwidth, between the once separated tasks.

For example, the humanoid robot ARMAR-III uses a Harris corner detector [6] as the first stage in the object-recognition and -tracking algorithm. It has to operate on the real-time video stream produced by the cameras of the robot. Hence, it has to maintain a steady throughput and good response times to ensure high-grade

* Corresponding author.
E-mail address: johny.paul@tum.de (J. Paul).

http://dx.doi.org/10.1016/j.sysarc.2015.07.011
1383-7621/

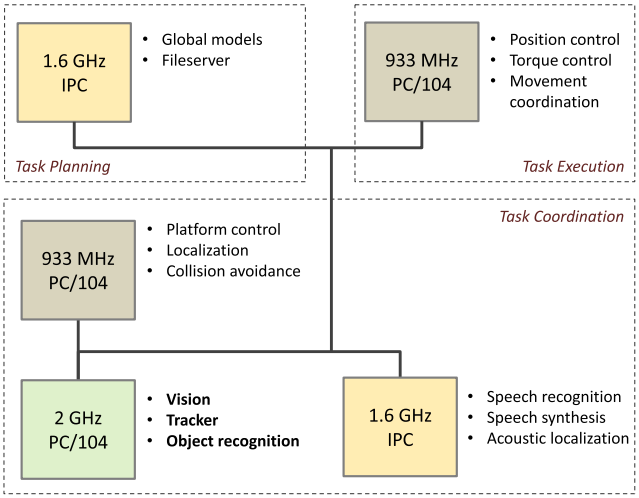

Fig. 1. Compute units on the ARMAR-III robot.

results, because any deterioration in the quality of the detected corners will negatively affect the remaining stages of the object-recognition algorithm. When consolidating the different tasks of the robot shown in Fig. 1 from their respective dedicated compute resources onto a MPSoC hardware, initial measurements showed that the results of the corner-detection algorithm are severely impaired if there are fewer resources available than necessary to process the real-time video stream. This leads to fewer detected corners, and up to 22 percent of the frames subsequently have to be dropped.

In this paper, we present a resource-aware computing paradigm called *Invasive Computing* to mitigate these negative effects of resource sharing. In our new model, the operating system can influence the internal decisions of the application based on dynamic load distribution on the MPSoC. This means that applications gain the ability to adapt to available resources by changing their own workload. For workload adaptations at runtime, we extend the pruning technique shown in [7] and combine it with our new resource-aware programming model. Our evaluation performed on a FPGA prototype shows that these extensions help the corner-detection application to significantly improve its throughput and accuracy.

The remainder of this paper is structured as follows. Section 2 describes problems in detail that arise when running corner-detection algorithms while sharing resources of the underlying machine. Section 3 presents the approach of Invasive Computing. In Section 4 we give a short introduction to the operating system OctoPOS with support for resource awareness. This is followed by the description of our resource-aware corner-detection algorithm in Section 5, together with the measures employed to convert the conventional algorithm to a resource-aware model and the workload distribution scheme based on available resources. Section 6 describes our evaluation platform, including measurements of the operating-system overhead. Section 7 presents the results of our evaluation, showing the benefits of the resource-aware model, while Section 8 discusses related work in the field of resource-aware programming. Finally, Section 9 gives our conclusions as well as future research directions.

2. Problem analysis

To analyze the problems arising from resource sharing, we examined the behavior of the conventional Harris [6] and Shi–Tomasi [8] corner detectors on a MPSoC with a total of 16 PEs. A sequence of 200 VGA frames (640 × 480 grayscale) was processed by the application. To evaluate the impact of other applications running concurrently on the MPSoC, we used applications like audio processing or motor control. These applications create dynamically changing load on the processor based on what the robot is doing at the current point in time. For instance, the speech-recognition application is activated whenever the user speaks to the robot and the motor control is activated when the robot has to move or grasp a recognized object. A conventional operating-system (OS) scheduler schedules the threads of each application considering the overall system load. As a result, the resources available to each application may vary over time. Fig. 2 shows how many PEs were allocated by the runtime system for corner detection.

The corner-detection algorithm is programmed to run at ten frames per second (fps) and needs a minimum of nine PEs. However, at various points in time, fewer PEs were offered by the runtime system, leading to frame drops as the algorithm is running on real-time video input. Frame drops reduce the overall accuracy of the corner-detection algorithm. In order to evaluate the impact of frame drops on the accuracy of detected corners, we use the metrics named *precision* and *recall* as proposed in [9]. The value of recall measures the number of correct matches out of the total number of possible matches, and the value of precision measures the number of correct matches out of all matches returned by the algorithm. The variation in accuracy values due to sharing of available resources is shown in Fig. 3, with the accuracy values computed using (1):

$$Ac(n) = \frac{\sum_{i=1}^{n} Pr(n) + \sum_{i=1}^{n} Re(n)}{2n} \quad (1)$$

Ac represents the average accuracy for *n* frames, *Pr* represents the precision and *Re* represents the recall values. The value of accuracy at any point is the average of the precision and recall until that point in time.

For the initial few frames, the accuracy stays at the maximum value of 1. However, lack of sufficient resources results in an accuracy drop as shown in Fig. 3. At some instances, the overall accuracy reaches a very low value of 0.73 for the Shi–Tomasi and 0.76 for the Harris detector. The lack of sufficient resources also results in an overshoot in execution time, where, based on the load conditions, the execution time increases by a factor of up to 4.3, as the processing of a single frame now takes 430 ms instead of 100 ms. The number of frame drops during the evaluation period is as high as 22 percent. The overall accuracy drops to 79 and 81 percent for Shi–Tomasi and Harris detectors, respectively. This is a significant loss in accuracy for many real-world applications of corner detection, like object recognition. Object recognition heavily relies on the results of corner detection; the robot may lose track of the object if too many consecutive frames are dropped.

Hence, in order to guarantee a certain accuracy level on a MPSoC, we need a way to cope with dynamically varying workload caused by multiple concurrent applications.

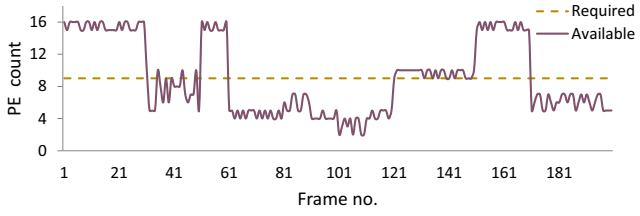

Fig. 2. Resource-allocation profile for corner detection.

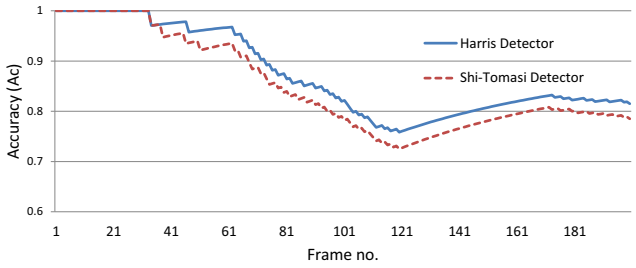

Fig. 3. Variation in accuracy with available resources for conventional Harris and Shi–Tomasi corner detectors running on a MPSoC.

3. Invasive Computing

Invasive Computing [10] is a holistic resource-aware programming paradigm that focuses on leveraging the computing power provided by multi- and many-core systems. It assumes a tiled hardware architecture as shown in Fig. 4, where a tile consists of a number of processing elements sharing a common bus and common memory. Cache coherency is guaranteed only within tile boundaries. Tiles are interconnected through a network-on-chip (NoC) that allows for non-uniform, non-cache-coherent memory access across tile boundaries.

The name Invasive Computing stems from the notion that applications can dynamically *invade* (exclusively acquire) resources according to their needs for computation power, communication bandwidth/latency or memory size. In such a system, applications compete with each other for a share of the hardware resources. In contrast to mere *allocation* of a fixed amount of resources, *invasion* can lead to over- or underfulfillment of the application's requirements, depending on the current state of the system. As a consequence, applications are required to be adaptable, because the result of an invasion may not fully meet their original demand. However, once an application acquires a set of resources, it gains exclusive access to them. This allows the application to tune its algorithms and distribute its workload depending on the amount of resources actually received from the system. The resource guarantees in turn ensure that decisions taken during program tuning and workload distribution remain valid until the application releases the resources back to the system.

Fig. 4 shows an exemplary application flow in an invasive program. First, during the invade phase, the application expresses a request for a set of resources through *hints* given to the runtime system. For example, it can specify that it would like to acquire four processing elements, and give an estimate as to how well it would scale if it were to receive fewer or more processing elements. The system then decides, according to the global system state, which resources to assign to the application, and returns a *claim* describing these resources.

In the next step, the application adapts itself according to the contents of the claim by assorting a so-called *team* of *i*-lets. An *i*-let is a light-weight unit of execution which consists of a snippet of code that can be executed in parallel, and the data on which it operates. In a simple scenario, the application creates at least one *i*-let for each reserved processing element to provide maximal system utilization.

The resources of the claim can then be *infected* with the assorted team, leading to the execution of the team's *i*-lets. Once the execution has finished, the results can be collected and merged. The application may subsequently either reuse and adapt the claim for further computations or *retreat* from it, releasing the associated resources.

The non-traditional programming model of Invasive Computing requires special support from the underlying runtime system and OS to properly implement not only the dynamic resource allocation, but also the resource guarantees given after a claim has been acquired. An operating system that enables resource-aware applications is presented in the following section.

4. Introduction to OctoPOS

OctoPOS [11] is an operating system specifically designed to support the programming model of Invasive Computing. It provides the necessary OS-level primitives and a scalable, efficient, low-overhead execution environment for invasive-parallel applications. The primary *raison d'être* of OctoPOS is the insight that, in practice, the multi-threading model implemented by contemporary operating systems is far from light-weight: Performing a context switch already takes thousands of CPU cycles, and creating a new thread even wastes tens of thousands of cycles. In contrast, the goal of OctoPOS is to build a tailored operating system that cuts down on the costly high level of abstraction provided by traditional operating systems.

OctoPOS is designed for maximum scalability. On the one hand, this means being scalable to systems with hundreds or even thousands of processing elements, on the other hand scalability also has to encompass the application itself. This means that OctoPOS has to be able to execute applications that dynamically create lots of potentially parallel tasks on large MPSoC hardware. We tackle these requirements by employing the following design principles.

4.1. Design principles

On a coarse level, scalability is achieved by composing the overall system of several independent OS instances. With respect to the invasive hardware architecture described in Section 3, each tile of the MPSoC runs its own OctoPOS instance. This approach somewhat resembles the Multikernel presented in [12], but OctoPOS goes one step further and does not rely on global cache coherency between instances. This increases scalability especially for large MPSoCs, where cache coherency is getting increasingly difficult to implement and becomes a bottleneck in the whole system. Instead, the operating-system instances communicate with each other using special operations provided by the network-on-chip that connects the different tiles.

Inside the OS instances, OctoPOS strives to improve scalability by relying solely on non-blocking synchronization for implementing the key components of the application runtime environment. This includes, but is not limited to, all facilities that deal with resource allocation, task creation, execution and their synchronization.

Another constructive measure towards scalability is the tailoring of operating-system functionality to the needs of the application, since providing superfluous features often leads to unnecessary overhead during execution. One example of this is the preemption of running *i*-lets. Invasive applications are structured as a set of claims; inside the claim, the application itself has full control over the hardware, for example the processing elements. Hence, there is no need to temporally isolate *i*-lets running on a processing element through preemption mechanisms (for example, in order to prevent CPU monopolization by an application), as all of these *i*-lets belong to the same application anyway.

4.2. System architecture

The overall system design of OctoPOS is depicted in Fig. 5, exemplarily showing two instances of OctoPOS running on two compute tiles.

The *resource manager* implements the basic invasive functions *invade* and *retreat* that are used to create, manage and destroy

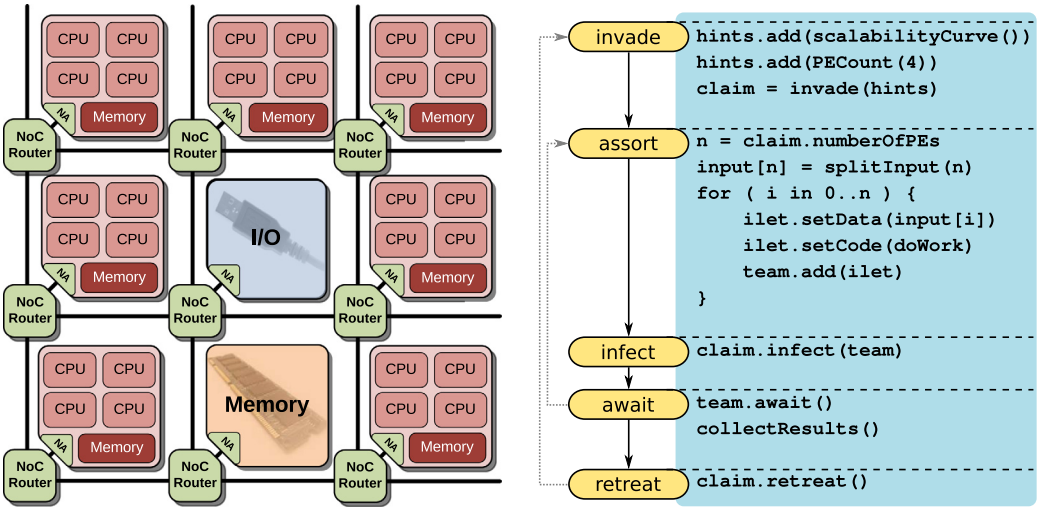

Fig. 4. Tiled hardware architecture (left) and application flow in an invasive program (right).

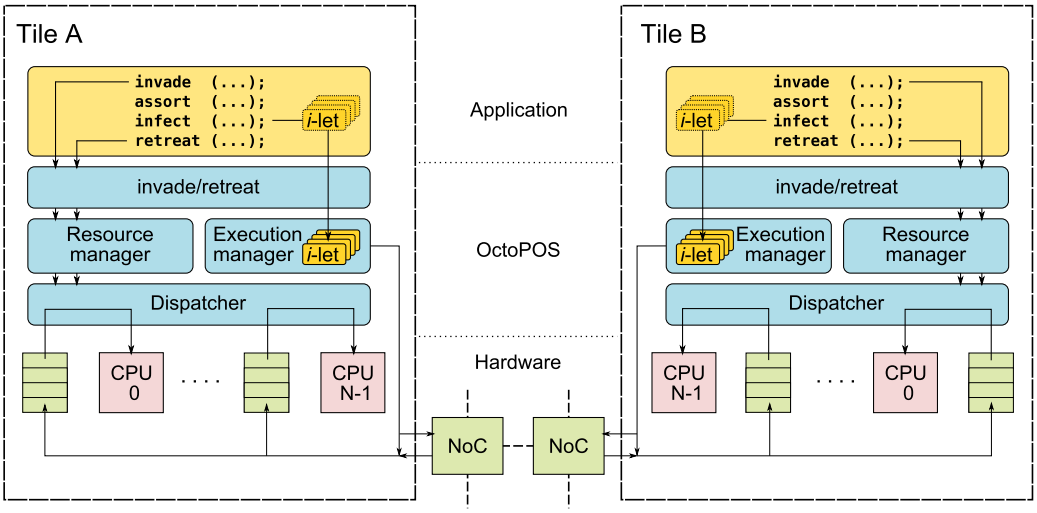

Fig. 5. Architectural overview of the invasive hardware/software stack.

claims containing processing elements on local or remote tiles. The claims and the actual user program supplied to the *execution manager* as a team of *i*-lets are then used as an input to *infect* to start execution. To provide an efficient execution environment, *i*-lets are regarded as mostly run-to-completion functions that can run in parallel. They are represented as a pair of pointers, one for the function to be executed and one for the input data. Due to its run-to-completion semantics, an *i*-let leaves no runtime state on the stack after it has terminated. The *i*-let descriptors are stored in processor-local *i*-let-queues and can be executed by the *dispatcher* like normal functions one after another on the same execution context, without the need for an expensive context switch in between. Only in the exceptional case that an *i*-let executes a blocking OS function (i.e., does not run to completion) must a context switch be performed and a new execution context has to be provided for the following *i*-lets.

To cross tile boundaries, the network-on-chip provides dedicated operations which were co-designed with the operating system. These operations allow OctoPOS to start *i*-lets on a different OctoPOS instance residing on another tile, or to copy a block of data from one tile to another. Applications can use these services via specific system calls to communicate across cache-coherency boundaries.

In summary, OctoPOS is a scalable operating system that provides an efficient execution environment for applications following the paradigm of Invasive Computing. OctoPOS was used as a basis for the implementation of the resource-aware corner-detection algorithms described and evaluated in the following sections.

5. Resource-aware corner detectors

Corner detection is an approach used within computer-vision systems to extract certain kinds of features and infer the contents of an image. Application scenarios include motion detection, image registration, video tracking, image mosaicing, panorama stitching, 3D modeling, and recognizing textured objects. Several corner detectors exist today in literature and comparative evaluations have shown that the Harris [6] and Shi–Tomasi [8] detectors achieve some of the best results. Also, a wide range of real-time applications [13–16] have used Harris and Shi–Tomasi detectors.

5.1. Basics of corner detection

This section provides a brief overview of the Harris and Shi–Tomasi corner detectors. To detect corners, both algorithms test

each pixel in the image by considering how similar a patch centered on the pixel is to nearby, largely overlapping patches. If the pixel is in a region of uniform intensity, then the nearby patches will look similar. If the pixel is on an edge, nearby patches in a direction perpendicular to the edge will look different, but parallel to the edge will result only in a small change. If the pixel is on a feature with variation in all directions, then none of the nearby patches will look similar. The calculation is based on the local auto-correlation function that is approximated by matrix M over a small window w for each pixel $p(x,y)$:

$$M = \begin{bmatrix} \sum_w W(x) I_x^2 & \sum_w W(x) I_x I_y \\ \sum_w W(x) I_x I_y & \sum_w W(x) I_y^2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad (2)$$

I_x and I_y are horizontal and vertical intensity gradients, respectively, and $W(x)$ is an averaging filter that can be a box or a Gaussian filter. The eigenvalues λ_1 and λ_2 (where $\lambda_1 \geqslant \lambda_2$) indicate the type of intensity change in the window w around $p(x,y)$:

1. If both λ_1 and λ_2 are small, $p(x,y)$ is a point in a flat region.
2. If λ_1 is large and λ_2 is small, $p(x,y)$ is an edge point.
3. If both λ_1 and λ_2 are large, $p(x,y)$ represents a corner point.

Shi–Tomasi directly computes the smaller eigenvalue λ_2 as its corner measure C as shown in (3):

$$C = \lambda_2 = \frac{(a+c) - \sqrt{(a-c)^2 + 4b^2}}{2} \quad (3)$$

Harris combines the eigenvalues into a single corner measure R as shown in (4), which avoids the explicit computation of eigenvalues (k is an empirical constant with value 0.04–0.06).

$$R = \lambda_1\lambda_2 - k \cdot (\lambda_1 + \lambda_2)^2 = (ac - b^2) - k \cdot (a+c)^2 \quad (4)$$

Once the corner measure is computed for every pixel, a threshold is applied on the corner measures to discard the obvious non-corners. The rest of the pixels are then ranked in the descending order of the corner measure. After applying the non-maximal suppression, the pixels with the highest corner measures are then selected as corners.

5.2. Extension to a resource-aware model

Independent of the algorithm used, corner detection is a compute-intensive step. Two different approaches have been used to increase the throughput of corner detection. The first approach focuses on hardware accelerators or graphics processing units (GPUs) to accelerate the conventional algorithm while the second employs algorithmic techniques to reduce the computational complexity. A high throughput can be guaranteed using hardware accelerators based on field-programmable gate arrays (FPGAs). However, FPGAs offer rather little flexibility and require very high effort in terms of design, implementation and verification. GPUs, on the other hand, are very powerful and provide significant acceleration over small multi-core processors, but they consume very high power and require data transfers between the processor and the hardware accelerator, which increases overall latency.

Hence, a more suitable approach is algorithmic optimization, which includes a pruning technique based on gradient magnitude that selects pixels with a high gradient magnitude as corner candidates for the Shi–Tomasi and Harris algorithms [7]. Another pruning technique to reduce the computational complexity of the conventional Harris detector is described in [17]. This technique relies on the fact that in most situations, the obvious non-corners constitute a large majority of the image. Hence the corner detectors incur a lot of redundant computations as they evaluate the entire image for a high corner response.

In this work, we extend the threshold-based pruning technique in [7] to a resource-aware pruning technique in order to control the workload for the corner detection based on available resources on the MPSoC. In the first step, the algorithm considers the corner point as the junction of two or more edge lines. One important property of such a point is that it has a high gradient change in more than one direction. A corner response (CR) function can be defined from the above property as:

$$CR = (|I_x| \cdot |I_y|) \quad (5)$$

where I_x and I_y are the horizontal and vertical pixel-intensity derivatives. If CR is greater than a predefined gradient threshold, the pixel is a corner candidate and should be retained for processing in the subsequent steps. This technique ensures that the non-corner pixels are removed prior to more intensive processing. From (4), R is most influenced by the term $(ac - b^2)$ as the two $(a+c)$ terms cancel out. For a good corner, R needs to be a large value. Therefore, maximizing $(ac - b^2)$ can select good corners without explicit eigenvalue computation. Hence, the new algorithm successively reduces the number of candidate corners at every step to minimize the computational effort. In the next step, an intermediate corner non-maxima suppression is applied to reduce the effect of false corners caused by multiple responses of the gradient operator. All candidate corners from the previous steps are further assessed through computing the eigenvalues as in the conventional Harris (4) or Shi–Tomasi detector (3). Finally, a non-maxima suppression is applied to suppress the corners that are close to each other. Some of the challenges posed by the conventional corner detector on a MPSoC can be resolved using the pruning technique described above.

The main idea and novelty of the resource-aware corner detector is that the threshold for pruning is based on the available resources. This means that the new detector can control the workload by changing the threshold. In situations where the system is under-utilized, the threshold can be reduced, whereby more pixels will be processed and a higher accuracy will be achieved. On the other hand, increasing the threshold can prune away more pixels when the processing system is heavily loaded by other high-priority tasks.

Fig. 7 shows the relation between the threshold and the processing interval (pixels processed). The results were captured by applying the pruning technique on six different video sequences as shown in Fig. 6 (each video sequence consists of 200 frames). In order to evaluate the impact of pruning on the accuracy of detected corners, we use the metrics named *precision* and *recall* as proposed in [9]. Results indicate that the effects of pruning vary based on the scene. For instance, the speedup achieved (using the same threshold) is low for cluttered scenes like *Bricks* while the majority of the pixels can be pruned away for scenes with plain backgrounds (e.g., the moving cereal box). Fig. 7 also shows the relation between threshold and accuracy (average of precision and recall rates) for all six video sequences, plotted independently of each other. Hence, the amount of computing resources required to perform the corner detection will vary from one scene to another based on the nature of the foreground, background, and so on. Therefore, the resources have to be allocated on a frame-to-frame basis, based on the scene captured. Section 5.3 explains the technique used to estimate resource demands based on application requirements, how to allocate and release these resources at runtime, and how to adapt to available resources at runtime.

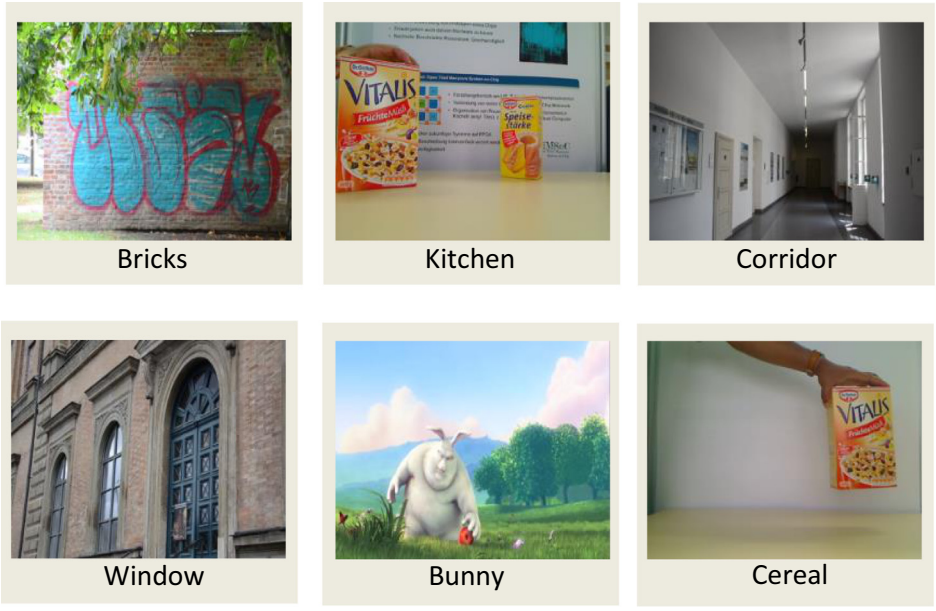

Fig. 6. Snapshots of the video sequences used for evaluation.

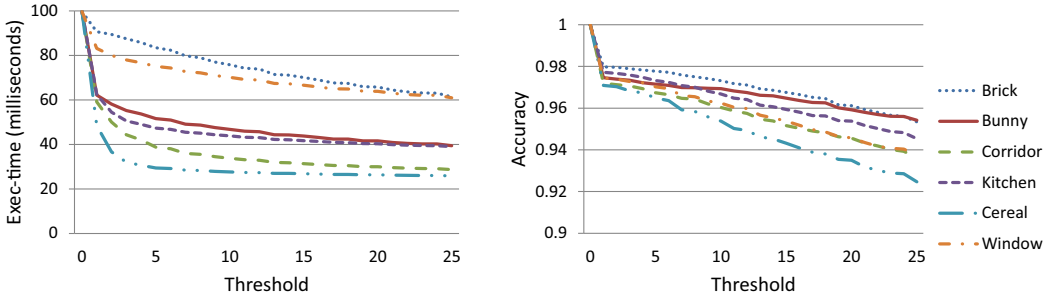

Fig. 7. Effects of pruning on execution time and accuracy for Harris detector.

5.3. Resource allocation and workload distribution

The first step within the algorithm is to allocate sufficient resources to perform corner detection within the interval specified by the user. As shown in Fig. 7, a threshold around 4 offers a significant speedup without much loss in accuracy. This region is interesting for most mobile platforms as the number of computations and hence power consumption can be reduced by operating in this range.

Choosing a threshold of 4 means that the computing resources (PEs) needed by the algorithm will vary over time. Fig. 8 shows the PEs required for Harris corner detection with a pruning threshold of 4. As the percentage of pixels that can be pruned depends on the nature of the scene, each scene type requires a different set of PEs. In some cases the PE count varies within the scene type due to minor changes within the video sequences. For example, as the object in the *Kitchen* scene moves towards the camera, the scene becomes more cluttered, the algorithm cannot prune away as many pixels as in the previous frame and hence to meet the deadline of 10 fps, the application has to *invade* more PEs.

The resource-aware corner-detection algorithm is structured as follows. The core algorithm consists of five stages as shown in Fig. 9. The first stage is called a resource-estimation stage, where the image is pre-processed (differential-image and integral-histogram computation) to estimate the PEs needed based on the scene type and the user-specified deadline.

The analysis starts with the generation of a differential image, where each pixel is computed using (5). In order to speed up the pruning logic within the algorithm, an integral histogram is computed from the differential image as described in Algorithms 1 and 2, where n is the total number of pixels to be processed, I_{diff} is the differential image, dx and dy are the horizontal and vertical pixel-intensity derivatives, *limit* is the maximum possible value generated by (5) and H is the integral histogram computed from differential image. Once the integral histogram is computed, the values in the bin represent the number of pixels to be processed by the algorithm when the threshold is set to histogram-bin-index.

Algorithm 1. Differential image

1: $i \leftarrow 0$
2: $h \leftarrow 0$
3: **while** $i < n$ **do**
4: $\quad I_{diff}(i) \leftarrow |dx(i) \cdot dy(i)|$
5: $\quad h(I_{diff}(i)) \leftarrow h(I_{diff}(i)) + 1$
6: $\quad i \leftarrow i + 1$
7: **end while**

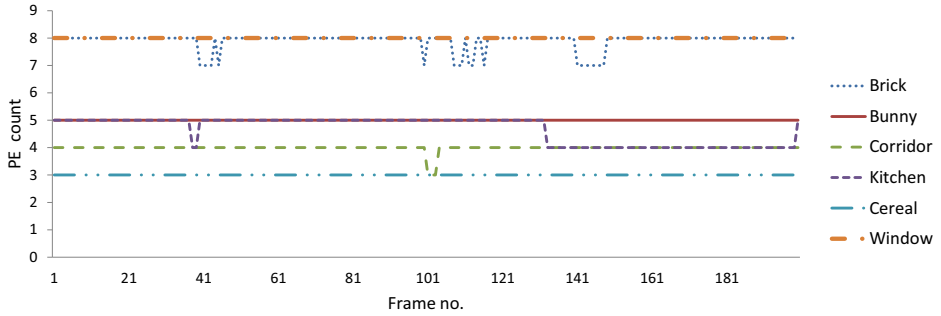

Fig. 8. Resource usage based on scene type (Harris detector with a threshold of 4).

Algorithm 2. Integral histogram

```
1: i ← 0
2: H ← 0
3: while i < limit do
4:    k ← i
5:    while k < limit do
6:       H(i) ← H(i) + h(k)
7:       k ← k + 1
8:    end while
9:    i ← i + 1
10: end while
```

In the next stage, an *invade* request is raised to allocate sufficient PEs using (6),

$$N_{pe} \geqslant \frac{(n \cdot T_{prn}) + (P_{pix}(th) \cdot T_{cd})}{T_{exe} \cdot \eta(N_{pe})} \tag{6}$$

where N_{pe} is the number of PEs needed for the computation, T_{prn} is the processing time per pixel until the end of the pre-processing stage, P_{pix} is the number of pixels to be processed by the upcoming stages as computed by the pruning algorithm as a function of the threshold value th, T_{cd} is the time to compute the final corner measure for pixels with corner response (CR) above threshold and T_{exe} is the processing interval. $\eta(N_{pe})$ represents the algorithm's efficiency as a function of degree-of-parallelism or available resources. This term helps to increase the accuracy of the resource-estimation model, as the execution time for corner detection application does not decrease linearly with increasing degree of parallelism. This is due to the fact that some parts of the algorithm (workload distribution, merging final results, etc.) are performed by a single *i*-let, in a sequential manner. Moreover, additional *i*-lets created by the application also creates additional load on the external memory and shared communication interfaces, limiting the scalability.

Our analysis on the proposed hardware shows an efficiency graph as depicted in Fig. 10. From the graph it can be seen that when the number of *i*-lets is increased from 1 to 2, the execution time does not improve by 2×, instead by 2 × 0.98 (98%), that is, 1.96×. Using this graph, the efficiency factor for various levels of parallelism can be computed. The values shown here are applicable only for the Harris-detector implementation used in this paper and may vary based on how the original algorithm is implemented. For best results, the threshold value *th* can be set to the so that the algorithm will attempt to process all pixels in the image. It should be noted that T_{prn} and T_{cd} may vary depending on the actual implementation and the processor architecture. Hence these values are estimated by profiling the application on the target platform. Once sufficient resources are *invaded*, the workload-distribution

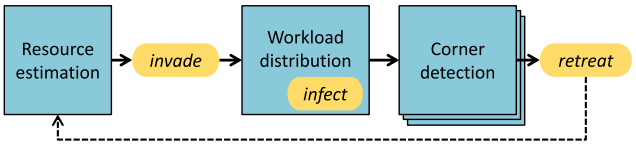

Fig. 9. Flow diagram for the resource-aware corner detector.

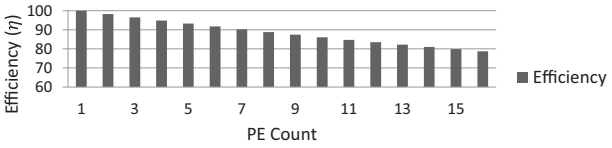

Fig. 10. Efficiency map for resource-aware Harris corner detection on target HW.

stage can split the total workload into N *i*-lets (where $N = N_{pe}$) and move the individual workload data to the target tiles, through DMA operations over the NoC. This is followed by the *infect* operation which send the *i*-lets to perform the corner detection based on the pre-calculated threshold value. The *i*-lets are scheduled on the *invaded* PEs by OctoPOS. Each *i*-let computes the corner measure using (3) or (4), depending on which type of corner detection algorithm is being used. Only those pixels with corner response (CR) above the threshold value are processed by *i*-lets. The results computed by each *i*-let is stored in local memory and copied back to the source tile once the entire workload is processed. Upon completion, the application can release the allocated PEs (*retreat*) and retain execution on a single PE or perform a new *invade* request to allocate PEs for the next frame.

Considering the current system load, OctoPOS makes a final decision on the number of PEs to be allocated to the application. The PE 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 PEs requested (provided that a sufficient number of idle PEs exist in the system and the current power mode offers sufficient power budget to enable the selected PEs). This means that under numerous circumstances the application may end up with fewer PEs and has to adapt itself to the limited resources offered by the runtime system. This is achieved by increasing the threshold value *th* until the condition in (7) is satisfied.

$$P_{pix}(th) \leqslant \frac{N_{pe} \cdot T_{exe} \cdot \eta(N_{pe}) - (n \cdot T_{prn})}{T_{cd}} \tag{7}$$

Fig. 11 shows the effect of increasing the threshold on a sample image (top left corner). The pixels pruned away are represented using black color and the processed pixels in white. As the pruning threshold is increased, more pixels are pruned away and hence

more and more regions in the resultant image turn black. This example also shows that the pruning technique eliminates regions without corners (e.g., wall regions) while the regions with high gradient values are retained for further processing. The modified flow diagram is depicted in Fig. 12, where a new stage (adaptive threshold calculation) is added after the resource-allocation stage. This stage helps the algorithm to adapt to the available resource so that the probability of frame drops can be reduced even when sufficient resources are not available. The results from the resource-aware model are presented in Section 7, where the conventional algorithm is compared against the resource-aware model.

6. Evaluation platform

We implemented and evaluated our adaptive algorithm for corner detection on a version of the invasive hardware platform described in Section 3. It comprises 16 SPARC LEON3 processing elements that are available as part of GRLIB [18], equally distributed over 4 tiles that in turn are connected via a custom-designed network-on-chip [19]. The platform is prototyped on multiple FPGAs (Synopsys CHIPit System [20] consisting of six Xilinx Virtex-5 XC5VLX330 FPGAs) and runs at a frequency of 50 MHz.

Using this platform, we performed a number of micro-benchmarks to determine the system overhead when using the invasive commands that OctoPOS provides. To measure execution times, we used cycle-accurate time-stamp counters present in our hardware prototype. The upper part of Table 1 shows the runtimes of the system calls *invade* and *retreat* when increasing or decreasing a claim by one processing element for both local PEs on the same tile, as well as remote ones on another tile of the system. The lower part shows the execution time of the *infect* function for one *i*-let in a team of twenty *i*-lets, once for a claim residing on the local tile and once for a claim on a remote tile. In addition to the mere execution time of *infect*, we also measured the time it takes for a new *i*-let to get executed on another processing element. This *infect* latency is measured from the start of executing the *infect* function to the beginning of the actual user code of the *i*-let.

As expected, invading a remote PE is more costly than doing the same operation locally. However, retreating from a remote PE is actually cheaper, as the request is just forwarded to the remote tile via the network-on-chip, and is then processed there asynchronously. Starting application code both locally and remotely by employing *infect* is quite fast and, due to special hardware support, the remote *infect* does not suffer from the same penalty as a remote *invade* call does. The same holds for the *infect* latency that stays in the same order of magnitude even when crossing the tile boundary. This means that even short *i*-lets can be spawned efficiently and, thus, applications can make better use of a massively parallel hardware.

Generally, these results show that the implementation of OctoPOS imposes very little overhead on the application. In comparison, the operations for creating and switching traditional threads, for example on Linux, are typically more expensive by multiple orders of magnitude. Hence, resource awareness can be implemented efficiently by means of a specialized system-software layer.

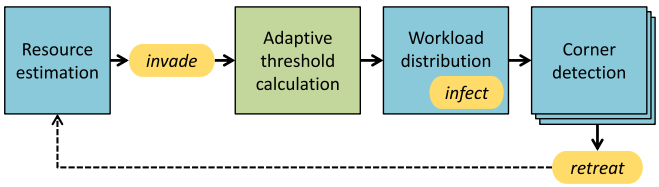

Fig. 12. Flow diagram for threshold estimation based on available resources.

Table 1
Execution overheads for fundamental OctoPOS system calls (All numbers are given in execution cycles).

	Invade PE	Retreat PE
Tile-local	355	471
Cross-tile	2162	235
	Infect PE	Infect latency
Tile-local	86	168
Cross-tile	99	255

7. Results

This section presents the results obtained using the resource-aware corner-detection algorithm and also provides a

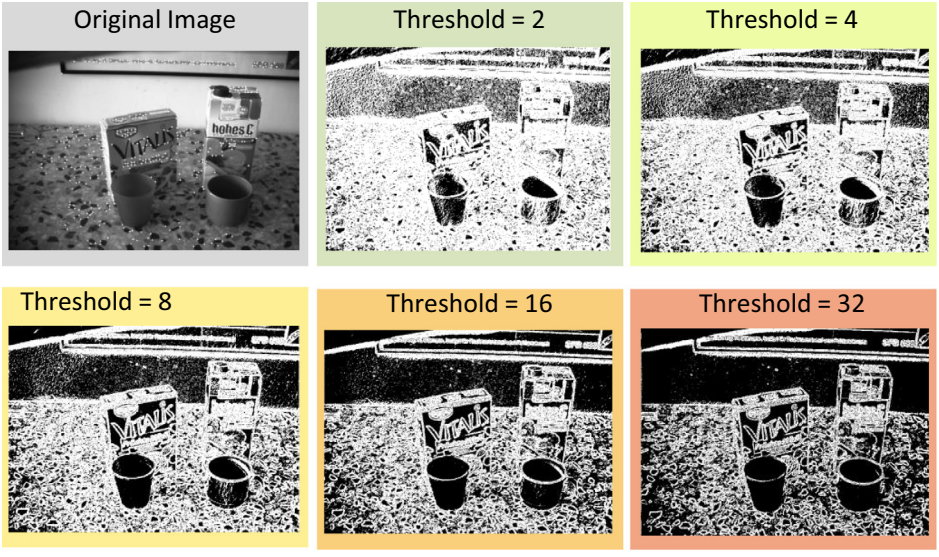

Fig. 11. Image samples demonstrating the pruning effect.

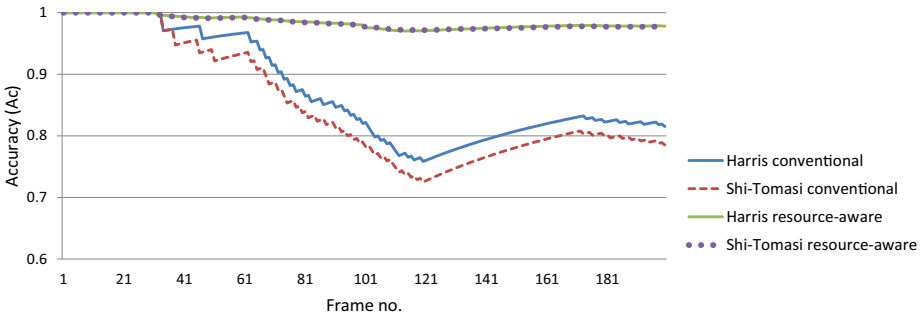

Fig. 13. Comparison between the conventional and resource-aware models.

Table 2
Comparison between conventional and resource-aware corner detectors.

Algorithm	Throughput (%)	Accuracy
Harris conventional	81	0.81
Harris resource-aware	100	0.98
Shi–Tomasi conventional	78	0.78
Shi–Tomasi resource-aware	100	0.98

comparison with the conventional model. The behavior of the new resource-aware corner detector is depicted in Fig. 13. The conventional and the resource-aware models were evaluated on the MPSoC hardware described in Section 6. A fixed resource-allocation scheme was used, as shown in Fig. 2, to ensure a fair comparison. Results were captured for both Harris and Shi–Tomasi detectors and the accuracy (*Ac*) values shown in Fig. 13 were computed using Eq. (1). Our evaluation shows that the adaptations made by the resource-aware model helped to avoid frame drops. No frame was dropped during the evaluation and the accuracy values improved significantly over the conventional approach. The resource-aware and the conventional algorithms resulted in the same accuracy value when sufficient resources are available. In the case of resource shortage, however, the resource-aware model starts to adapt the workload by increasing the pruning threshold. A slight drop in accuracy caused by the pruning of pixels is visible in Fig. 13. However, the overall accuracy has improved significantly over the conventional approach. Moreover, this helps to avoid the overshoot in execution time and thus to eliminate frame drops, so that the results are consistently available within the predefined deadline.

An overall comparison between the two models is available in Table 2, where throughput is represented by the percentage of frames processed and accuracy is measured using *Ac* presented in Eq. (1). In brief, the resource-aware corner detector can operate very well under dynamically changing conditions by adapting the workload and thus avoiding frame drops and improving the detection accuracy.

When comparing the conventional detectors, the overall accuracy values for the Shi–Tomasi detector remain below the Harris detector as the Shi–Tomasi algorithm involves additional computations (additional square-root and division operation in (3)) while the resource-allocation pattern remained unchanged. However, the resource-aware versions of both algorithms result in an almost equal accuracy as the pruning was more effective for the Shi–Tomasi detector due to the higher computation demand after the pruning stage.

Table 3 shows the profiling results for each stage within the resource-aware Harris corner detector. These experiments involved an *invade* request for 8 PEs spread across the 4 tiles of our evaluation platform. As the execution time for corner detection varies based on the actual workload, it is difficult to provide a precise number for this stage. The value provided in Table 3 is for zero

Table 3
Time taken for various stages in the resource-aware corner detection.

Stage	Time (ms)
Resource estimation	12
Resource allocation (*invade*)	0.26
Adaptive threshold calculation	0.8
Workload distribution	2.4
Corner detection	97
Resources de-allocation (*retreat*)	0.01
Total	112.5

threshold, meaning that all pixels were processed. It can be seen that the time spent for resource allocation and release is below 0.5 percent, while the additional logic for resource awareness resulted in approximately 12 percent overhead. However, results computed within this stage can be reused in the corner-detection stage, resulting in a reduced overhead for the corner detection. The time for *infect* is only a small fraction of the workload-distribution stage, as most of the time is consumed by loading and transferring the large image blocks across tiles.

Regarding scalability towards larger architectures, two effects come into play. First, with an increasing number of tiles, communication latency over the NoC rises, as packets have to traverse more hops to reach their destination. For our prototype's $N \times N$ meshed grid network, the maximum number of traversed hops equals $\sqrt{N}$, which means that maximum latency scales better than linearly with an increasing number of tiles. Second, when targeting larger platforms with more tiles and PEs, more messages need to be send both for acquiring and releasing resources via *invade* and *retreat*, and for dispatching *i*-lets via *infect* to do the actual computation. For *invade* and *retreat*, it is necessary to send more messages to address more tiles. However, once the request messages are out, the operations complete in parallel on all affected tiles. The same holds for dispatching *i*-lets via *infect*. Therefore, on the one hand with increasing architecture sizes we incur an additional overhead for initiating *invade*, *infect* and *retreat* operations, but on the other hand we leverage the increased parallel processing power to complete all operations in parallel.

8. Related work

A major challenge associated with future MPSoCs is the question how to program such systems to make best use of their computing power. In order to address this issue, [21] proposes ROS (**R**esource-aware **O**perating **S**ystem), an operating system for MPSoC hardware with support for parallel applications and a scalable kernel. ROS offers a resource-management scheme based on resource provisioning which enables system-wide, efficient accounting and utilization of resources. 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.

Tessellation [22] is another resource-aware MPSoC operating system that provides guaranteed fractions of system resources (such as processors, cache, network or memory bandwidth) to application programs. The status of an application is monitored through performance counters and heartbeats. Application heartbeats enable applications to communicate both their current and target performance. Within Tessellation, a *Resource Allocation Broker* distributes resources to cells while attempting to satisfy competing system-wide goals, such as deadlines, energy efficiency, and throughput.

Both ROS and Tessellation share the same view with OctoPOS as far as application-directed resource management of MPSoC is concerned. However, they do not implement a runtime environment specifically tailored towards large MPSoCs and furthermore are limited to cache-coherent system designs.

The Autonomic Operating System project AcOS [23] enhances commodity operating systems with an autonomic layer that enables self-X properties through adaptive resource allocation. They aim at enabling users to state Service Level Objectives (SLOs) and to automatically tune resource allocations in order to meet these user-specified SLOs, while enforcing system-level constraints, such as maximum processor temperature. Unlike AcOS, OctoPOS is a native operating system rather than an additional software layer.

A further related approach that, however, covers a much wider scope than considered by OctoPOS is SEEC [24]. The goal of SEEC is to reduce the programming effort in multi-core systems. The idea is to (1) have application programmers in charge of specifying goals and progress and (2) enjoin on system programmers the specification of system-level actions that can affect application-level processes. Based on these specifications, the SEEC runtime system then attempts to optimize resource allocations using an *adaptive control system* that learns application and system models on-line. Regarding the constructive approach, SEEC is implemented as a runtime system and a set of libraries for Linux. In this sense, it shares the same approach with AcOS and, thus, differentiates fundamentally from OctoPOS, as it cannot directly control the hardware to provide for the strict guarantees that are needed to separate different applications from each other.

9. Conclusion and future work

With their immense computational power, MPSoCs provide the means to consolidate the whole computing system in mobile robotic applications onto one chip instead of relying on separate physical computing units. With our case study evaluating two widely used corner-detection algorithms, we demonstrated accuracy loss on a conventional MPSoC hardware due to resource sharing and thus, we have shown that running once-separated applications on the same machine poses major problems.

Therefore, we proposed a resource-aware programming model, *Invasive Computing*, to overcome these challenges. It enables the application programmer to tune the workload and data distribution of the application according to the actual resources available in the system at runtime. We have adapted both the Harris and Shi–Tomasi corner detectors to the invasive programming model and extended already existing pruning techniques. We have enabled the corner-detection algorithms to recalculate their workload at runtime, based on the currently available resources on the MPSoC.

These adaptations help the corner-detection application to avoid frame drops and guarantee smooth processing of real-time video inputs. Our evaluation shows that adapting the workload to available resources dynamically at runtime helps to improve the performance, with up to 22 percent improvement in throughput and up to 20 percent improvement in accuracy. Profiling results based on an FPGA prototype indicate that the overhead from resource allocation and release can be kept very low by employing a specialized operating system and suitable hardware support.

Our future work will focus on extending the resource-aware model to more robotic applications, to perform evaluations on the complete chain of algorithms, like object recognition, and also to develop prediction models within the applications so as to foresee the resource requirements. We will also focus on techniques to efficiently allocate other resources like communication bandwidth and on-chip memory, among various applications on the MPSoC. The runtime system will be extended to allow the applications to allocate (temporal) fractions of a PE and to migrate running applications to different tiles if needed. Also, we will make the operating system dynamically reconfigurable, so that it can adapt itself to the requirements of applications at runtime.

References

[1] T. Asfour, K. Regenstein, P. Azad, J. Schroder, et al., ARMAR-III: an integrated humanoid platform for sensory-motor control, in: 2006 6th IEEE-RAS International Conference on Humanoid Robots, 2006, pp. 169–175. http://dx.doi.org/10.1109/ICHR.2006.321380.

[2] Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, et al., The intelligent ASIMO: system overview and integration, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2002, vol. 3, 2002, pp. 2478–2483. http://dx.doi.org/10.1109/IRDS.2002.1041641.

[3] K. Kaneko, F. Kanehiro, M. Morisawa, K. Miura, S. Nakaoka, S. Kajita, Cybernetic human HRP-4C, in: 9th IEEE-RAS International Conference on Humanoid Robots, 2009. Humanoids 2009, 2009, pp. 7–14. http://dx.doi.org/10.1109/ICHR.2009.5379537.

[4] S. Jorg, M. Nickl, A. Nothhelfer, et al., The computing and communication architecture of the DLR hand arm system, in: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2011, pp. 1055–1062.

[5] T. Bjerregaard, S. Mahadevan, A survey of research and practices of network-on-chip, ACM Comput. Surv. (CSUR) 38 (1) (2006) 1.

[6] C. Harris, M. Stephens, A combined corner and edge detector, in: Alvey vision conference, vol. 15, Manchester, UK, 1988, p. 50.

[7] S. Alkaabi, F. Deravi, Candidate pruning for fast corner detection, Electron. Lett. 40 (1) (2004) 18–19.

[8] J. Shi, C. Tomasi, Good features to track, in: Proceedings CVPR'94, 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1994, IEEE, 600, 1994, pp. 593–600.

[9] J. Klippenstein, H. Zhang, Quantitative evaluation of feature extractors for visual slam, in: Fourth Canadian Conference on Computer and Robot Vision, 2007. CRV'07, IEEE, 2007, pp. 157–164.

[10] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, Gregor Snelting, Invasive Computing: An Overview, in: M. Hübner, J. Becker (Eds.), Multiprocessor System-on-Chip – Hardware Design and Tool Integration, Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[11] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, W. Schröder-Preikschat, OctoPOS: a parallel operating system for invasive computing, in: Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys, 2011.

[12] A. Baumann, P. Birham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, A. Singhania, The multikernel: a new OS architecture for scalable multicore systems, in: Proceedings of the 22nd ACM Symposium on Operating System Principles (SOSP 2009), Big Sky, MT, USA, 2009, pp. 29–44. October 11–14, ACM Press, New York, NY, USA, 2009.

[13] D.J. Mirota, M. Ishii, G.D. Hager, Vision-based navigation in image-guided interventions, Annu. Rev. Biomed. Eng. 13 (2011) 297–319.

[14] S. Ehsan, K.D. McDonald-Maier, On-board vision processing for small UAVs: Time to rethink strategy, in: NASA/ESA Conference on Adaptive Hardware and Systems, 2009. AHS 2009, IEEE, 2009, pp. 75–81.

[15] A. Schmidt, M. Kraft, A. Kasiński, An evaluation of image feature detectors and descriptors for robot navigation, in: Computer Vision and Graphics, Springer, 2010, pp. 251–259.

[16] S. Gauglitz, T. Höllerer, M. Turk, Evaluation of interest point detectors and feature descriptors for visual tracking, Int. J Comput. Vision 94 (3) (2011) 335–360.

[17] M. Wu, N. Ramakrishnan, S.-K. Lam, T. Srikanthan, Low-complexity pruning for accelerating corner detection, in: 2012 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2012, pp. 1684–1687.

[18] J. Gaisler, GRLIB IP Library User's Manual (Version 1.1. 0), Gaisler Research.

[19] J. Heisswolf, R. Knig, M. Kupper, J. Becker, Providing multiple hard latency and throughput guarantees for packet switching networks on chip, Comput. Electr.

Eng. 39 (8) (2013) 2603–2622, http://dx.doi.org/10.1016/j.compeleceng.2013.06.005.
[20] Synopsys, in: CHIPit Platinum Edition – ASIC, ASSP, SoC Verification Platform, 2009.
[21] K. Klues, B. Rhoden, Y. Zhu, A. Waterman, E. Brewer, Processes and resource management in a scalable many-core OS, HotPar10, Berkeley, CA.
[22] J.A. Colmenares, G. Eads, S.A. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D.B. Bartolini, N. Mor, et al., Tessellation: refactoring the OS around explicit resource containers with continuous adaptation, in: DAC, 2013, p. 76.
[23] D.B. Bartolini, R. Cattaneo, G.C. Durelli, M. Maggio, M.D. Santambrogio, F. Sironi, The autonomic operating system research project: achievements and future directions, in: Proceedings of the 50th Annual Design Automation Conference, ACM, 2013, p. 77.
[24] H. Hoffmann, M. Maggio, M.D. Santambrogio, A. Leva, A. Agarwal, Seec: a framework for self-aware management of multicore resources, Tech. Rep. MIT-CSAIL-TR-2011-016, Computer Science and Artificial Intelligence Laboratory, MIT, March 2011.

Johny Paul is a PhD student and research staff at the Institute for Integrated Systems (LIS), Technical University of Munich (TUM), Germany. Johny Paul received his Bachelor Degree in Electronics and Communication Engineering from Mahatma Gandhi University, India and worked at Wipro Technologies in areas of VLSI design for 3 years. Later he received his Master Degree from Nanyang Technological University (NTU), Singapore in Integrated Circuits Design and is currently working at Technical University of Munich, Germany. His research interests include parallel processing, multi-core processor design, robotic vision, driver assistance systems and realtime embedded systems. He is currently pursuing research on the topic "Invasive Computing for Robotic Vision".

Benjamin Oechslein received his diploma degree in computer science from Friedrich-Alexander-University Erlangen-Nuremberg (FAU). Since then he has been working as a doctoral researcher at FAU as a member of the System Software Group. His main research interest lies in the design of system software for massively parallel systems, especially many-core architectures, with a focus on co-design between system software and hardware to lower overheads and increase the scalability of highly parallel application programs.

Christoph Erhardt received his degree (Dipl.-Inf.) in computer science in 2011 from Friedrich-Alexander-University Erlangen-Nuremberg (FAU), Germany. He has since become a member of the System-Software Group at FAU, where he is working as a doctoral researcher. His interests include compiler technologies and operating-system design. He is currently researching mechanisms for dynamic system-software reconfiguration.

Jens Schedel received his diploma degree in computer science from the Friedrich-Alexander University Erlangen-Nuremberg (FAU) in 2010 and is currently a doctoral researcher at the system software group of the FAU. His research interests include operating system design for multi/manycore with a focus on concurreny-aware kernels.

Manfred Kröhnert received the diploma degree in computer science from the Karlsruhe Institute of Technology (KIT) in 2010. He is currently a doctoral researcher at the Karlsruhe Institute of Technology (KIT), where he is a member of the High Performance Humanoid Technologies lab. His research interests include hardware/software architectures for humanoid robots, especially profiling and modeling of behavior and resource demands of robot programs.

Daniel Lohmann is Assistant Professor at the Chair of Distributed Systems and Operating Systems at FAU. He received his diploma (Dipl.-Inform.) from University of Koblenz, Germany in 2002. In 2003 he joined the group of Wolfgang Schröder-Preikschat at FAU, where he received his PhD (Dr.- Ing.) in 2009 and his *venia legendi* (Dr.-Ing. habil.) in 2014. His research activities are centered around the topic of tailorable and configurable system software. Dr. Lohmann is a member of GI, ACM, and IEEE.

Walter Stechele received the Dipl.-Ing. and Dr.-Ing. degrees in electrical engineering from the Technical University of Munich, Germany, in 1983 and 1988, respectively. In 1990 he joined Kontron Elektronik GmbH, a German electronic company, where he was responsible for the ASIC and PCB design department. Since 1993 he is with Technical University of Munich. His interests include visual computing and robotic vision, with focus on Multi Processor System-on-Chip (MPSoC) architectures and design methodology, low power optimization, dynamic reconfiguration of FPGA devices, and applications in automotive and robotics.

Tamim Asfour is full Professor at the Institute for Anthropomatics, Karlsruhe Institute of Technology (KIT). He is chair of Humanoid Robotics Systems and head of the High Performance Humanoid Technologies Lab (H^2T). His current research interest is high performance humanoid robotics. He is developer and leader of the development team of the ARMAR humanoid robot family. He has been active in the field of Humanoid Robotics for the last 14 years resulting in about 150 peer-reviewed publications with focus on engineering complete humanoid robot systems including humanoid mechatronics and mechano-informatics, grasping and dexterous manipulation, action learning from human observation, goal-directed imitation learning, active vision and active touch, whole-body motion planning, system integration, robot software and hardware control architecture. He received his diploma degree in Electrical Engineering in 1994 and his PhD in Computer Science in 2003 from the University of Karlsruhe.

Wolfgang Schröder-Preikschat studied computer science at the TU Berlin, Germany, where he also received his Ph.D. and venia legendi. After spending about ten years as a research associate and director of the system software department at the German National Research Center of Computer Science (GMD FRIST), Berlin, he became a full professor for computer science at the Universities of Potsdam (1995–1997), Magdeburg (1997–2002), and Erlangen-Nuremberg (FAU, since 2002), Germany. He is elected member of the DFG Review Board and serves in the DAAD Selection Committee of the young academics program for computer science. His main research interests are in the domain of real-time embedded distributed/parallel operating systems. Selected memberships in professional associations include ACM, EuroSys, GI/ITG, IEEE, and USENIX.