Resource-Aware Harris Corner Detection based on Adaptive Pruning *

Johny Paul[1], Walter Stechele[1]
Manfred Kröhnert[2], Tamim Asfour[2]
Benjamin Oechslein[3], Christoph Erhardt[3], Jens Schedel[3],
Daniel Lohmann[3], and Wolfgang Schröder-Preikschat[3]

[1] Technical University of Munich, Germany
[2] Karlsruhe Institute of Technology, Germany
[3] Friedrich-Alexander University Erlangen-Nuremberg, Germany
{johny.paul, walter.stechele}@tum.de
{manfred.kroehnert, asfour}@kit.edu
{oechslein, erhardt, schedel,
lohmann, wosch}@cs.fau.de

Abstract. Corner-detection techniques are being widely used in computer vision – for example in object recognition to find suitable candidate points for feature registration and matching. Most computer-vision applications have to operate on real-time video sequences, hence maintaining a consistent throughput and high accuracy are important constrains that ensure high-quality object recognition. A high throughput can be achieved by exploiting the inherent parallelism within the algorithm on massively parallel architectures like many-core processors. However, accelerating such algorithms on many-core CPUs offers several challenges as the achieved speedup depends on the instantaneous load on the processing elements. In this work, we present a new resource-aware Harris corner-detection algorithm for many-core processors. The novel algorithm can adapt itself to the dynamically varying load on a many-core processor to process the frame within a predefined time interval. The results show a 19% improvement in throughput and an 18% improvement in accuracy.

Keywords: Harris corner detection, resource-aware programming, invasive computing, adaptive pruning

1 Introduction

Corner detection is used within computer-vision algorithms like motion detection, image registration, video tracking, feature descriptors for object recognition etc. to infer the contents of an image. Several corner detectors exist today in the literature and comparative evaluations have shown that the Harris [9] corner detectors achieve some of the best results. Recent evaluations in real-time applications such as video tracking [7], visual SLAM [8] and robotic navigation [19] have demonstrated that the

* This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89)

preferred way to detect features in a scene is the use of a Harris detector in combination with more complex feature descriptors. Harris detectors are also used in the humanoid robot ARMAR-III [2] for recognizing and tracking textured objects [3]. A humanoid robot like ARMAR has to handle various tasks like vision, motion planning, speech recognition, etc. with the workload spread across multiple industrial PCs. The data from various sensors flows into the processing system, each dedicated for a different task like computer vision, motion control, speech processing, etc. Similarly, the humanoid robot Asimo uses two PCs, a control and planning processor plus an additional digital signal processor (DSP) for sound processing [18]. Two processor boards were also used in the humanoid robots HRP-2 [11] and HRP-4C [12]. The Hand Arm System from DLR [10] has three layers of computing hierarchy consisting of COTS PCs for control applications, an auxiliary Linux workstation for user interfaces and a composition layer constituted by FPGAs for hardware-accelerated tasks.
The use of multiple PCs results in high power consumption, low interconnect bandwidth and occupies a large amount of space on the robot. The use of many-core processors can mitigate some of the above mentioned problems on account of their immense computational power assembled in a compact design. However, the available resources on a many-core chip (processing elements (PEs), memories, interconnects, etc.) have to be shared among various applications running concurrently, which leads to unpredictable execution time or frame drops for vision applications. Our work focuses on analyzing the effect of sharing resources on a conventional Harris detector and propose a new resource-aware Harris detector to resolve the issues. Evaluations shows that the newly proposed Harris detector is capable of adapting to varying load conditions on the many-core processor and delivers better results in terms of throughput, accuracy and latency. This work also describes how to distribute the workload on the massively parallel PEs for best results, avoiding frame drops, even under varying load conditions.
This paper is organized as follows. Section 2 describes the state-of-the-art algorithms used for corner detection and different schemes using for accelerating the algorithm. Section 3 provides a brief overview of the conventional Harris detector and describes some of the challenges with implementing a conventional Harris detector on many-core processors. Section 4 starts with a brief description of various pruning techniques to accelerate corner detection. This is followed by the description of the resource-aware corner detector using an enhanced pruning technique. Section 5 provides an overview of the many-core system used for evaluation and Section 6 describes the implementation and results, followed by Section 7, which concludes the paper.

2 State of the Art

Several techniques exist today to detect corners in an image. These include Harris corner detection [9], SUSAN [20], FAST [17], etc. Independent of the technique used, corner detection is a compute-intensive task and two main techniques have been used to speed it up. The first approach focuses on algorithmic techniques to reduce the computational complexity, while the second employs hardware accelerators or graphics processing units (GPUs) to accelerate the conventional algorithm. Independent of the technique used, they all pose a challenge to the programmer; how to control the worst-case execution time and avoid frame drops when the resources on the processor are shared across multiple applications. High throughput can be guaranteed using hardware accelerators based on field-programmable gate arrays (FPGAs). However, the flexibility offered by FPGAs is quite low and requires very high effort in terms of design, implementation and verification. On the other hand, GPUs are very powerful and provide significant acceleration over small multi-core processors due to their massively parallel architecture. However, they consume very high power, are less flexible, difficult to debug and require data transfers between processor and the hardware accelerator, which increases the overall latency.

The use of many-core processors can overcome many of the above mentioned hurdles as they offer higher computing power necessary to accelerate the algorithms, while at the same time retaining the simplicity in programming and debugging. Today it is possible to put onto a single chip a large number of general-purpose cores, certainly tens of highly complex cores as on Intel's Single-Chip Cloud Computer [15] or Tilera's 64-core processor [4]. A major challenge associated with todays many-core systems is the question of how to program such systems to make best use of their computing power. In order to address these issues, [14] propose a new resource-aware operating system (ROS) for many-core hardware, with direct 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.

The demand for more stringent (OS-supported) resource awareness was also proposed in [21], 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 many-core system like resource-aware OS, communication interfaces like Network-on-Chip (NoC), and PEs. Programs running on this system get the ability to explore and dynamically spread their computations to neighboring processors and execute portions of code with a high degree of parallelism in parallel based on the availability of resources. Once the program terminates or if the degree of parallelism is expected to be lower again, the program may enter a *retreat* phase. At this point, the resources can be deallocated and execution resumed, for example, sequentially on a single processor. In this work, a resource-aware Harris corner-detection algorithm is evaluated using the Invasive Computing methodology. However, the concepts demonstrated in this work are platform-independent and can be demonstrated on any resource-aware platform including ROS.

3 Harris Corner Detection

This section provides a brief overview of the conventional Harris corner-detection algorithm. The calculation is based on the local auto-correlation function that is approximated by a matrix M over a small window w for each pixel $p(x,y)$:

$$M = \begin{bmatrix} \sum_w W(x) I_x^2 & \sum_w W(x) I_x I_y \\ \sum_w W(x) I_x I_y & \sum_w W(x) I_y^2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{1}$$

where 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 $\lambda_1 \geq \lambda_2$) indicate the type of intensity change in the window w around $p(x,y)$. If both λ_1 and λ_2 are small, $p(x,y)$ is a point in a flat region. If λ_1 is large and λ_2 is small, $p(x,y)$ is an edge point and if both λ_1 and λ_2 are large, $p(x,y)$ represents a corner point. Harris combines the eigenvalues into a single corner measure R as shown in (2) (k is an empirical constant with value 0.04 to 0.06). Once the corner measure is computed for every pixel, a threshold is applied on the corner measures to discard the obvious non-corners.

$$R = \lambda_1 \lambda_2 - k \cdot (\lambda_1 + \lambda_2)^2 = (ac - b^2) - k \cdot (a + c)^2 \tag{2}$$

Corner detection is often employed as the first step in computer-vision applications with real-time video input. Hence, the application has to maintain a steady throughput and good response time to ensure quality results. However, the presence of other high-priority tasks may alter the behavior of the corner-detection algorithm. To evaluate such a dynamically changing situation, we analyzed the behavior of the conventional Harris detector on a many-core processor with 32 PEs. Fig. 1 shows resource-allocation schemes (left) along with the execution-time profiles (right). A video input with 640×480 pixels at 10 frames per second was used, with the test running for 20 seconds. To evaluate the impact of other applications running concurrently on the many-core system, applications like audio processing, motor control, etc. were used. These applications create dynamically changing load on the processor based on what the robot is doing at that point in time. For instance, the speech-recognition application is activated when the user speaks to the robot. The conventional OS scheduler schedules the threads of the applications based on the overall system load. Sharing of available resources resulted in the execution-time profile shown in Fig. 1. It can be seen that the execution time varies from 0 to 430 milliseconds, based on the load condition. A lack of sufficient resources leads to very high processing intervals or frame drops (a processing interval of zero represents a frame drop). The number of frames dropped during this evaluation is as high as **20%** and the worst-case latency increased by 4.3x (100 milliseconds to 430

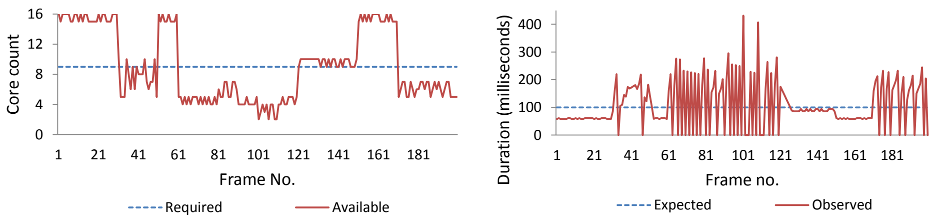

Fig. 1: Variation in processing interval based on available resources

milliseconds). Frame drops reduce the quality of the results and the robot may lose track of the object if too many consecutive frames are dropped. In order to overcome these challenges, we present a modified Harris detector for many-core processors capable of allocating resources based on the current workload. The algorithm can also adapt the workload based on the currently available resources. The following sections demonstrate how the resources are claimed and how the processing interval can be constrained to guarantee consistent throughput and processing intervals.

4 Pruning Techniques

A pruning technique to reduce the computational complexity of the conventional Harris detector is described in [22]. This technique relies on the fact that in most situations, the obvious non-corners constitute a large majority of the image. Hence the Harris detectors incur a lot of redundant computations as they evaluate the entire image for a high corner response. From (2), 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. Hence maximizing $(ac - b^2)$ can select good corners without explicit eigenvalue computation. However, this technique cannot demonstrate a noticeable speedup on platforms with FPU as the pixels are pruned away in the final step, just before eigenvalue computation. The limitations in [22] can be resolved using the multi-stage pruning technique described in [1]. The main difference between these techniques is that the second one can prune away pixels at a very early stage. A corner response (CR) is defined as:

$$CR = min\left(\left|I_x\right|, \left|I_y\right|\right), \tag{3}$$

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. All candidate corners from the previous steps are further assessed by computing the eigenvalues as in the conventional Harris detector (2). 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 Harris detector on a many-core system can be resolved using the pruning techniques described above. In situations where the system is under-utilized, the threshold can be reduced, thereby processing more pixels and achieving a higher accuracy, whereas increasing the threshold can prune away more pixels when the processing system is heavily loaded by other high-priority tasks. Fig. 3 shows the relation between the threshold and the processing

Fig. 2: Snapshot of the video-sequences used for evaluation

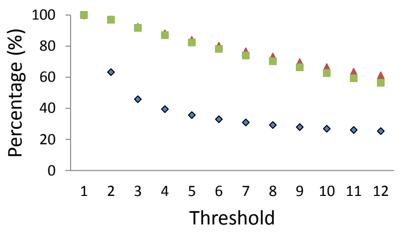

Fig. 3: Multi-stage pruning technique

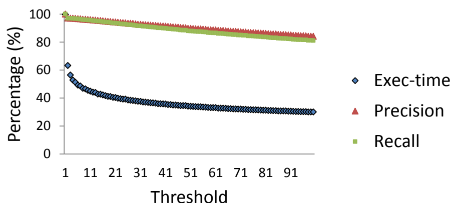

Fig. 4: Resource-aware pruning

interval. The results were captured by applying the pruning technique to six different video sequences whose snapshots are shown in Fig. 2 (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 [13]. 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. From Fig. 3 it can be seen that the application performs well for a threshold below four. However, increasing the threshold further results in a drastic decrease in precision and recall rates (as low as 60% for a threshold of 12). Values beyond this point are not plotted in the graph as the corners with an accuracy below 60% are not suitable for practical use. The second drawback of this algorithm is that it offers only few candidate points for adaptations within an acceptable accuracy range.

In order to overcome the challenges with the multi-stage pruning technique, we present an enhanced pruning technique with better flexibility and higher accuracy compared to the conventional pruning technique presented in Section 4. Our algorithm uses a new threshold model as described in (4), where product of vertical and horizontal difference in pixel intensities is used and the candidates with low CR values are pruned away.

$$CR = (|I_x \cdot I_y|) \tag{4}$$

This new model results in significantly more selection points as shown in Fig. 4, offering a higher flexibility to the resource-aware algorithm whenever adaptations are necessary. In addition to this, there is a significant improvement in both precision and recall rates. For example, precision is improved from 84.0% to 90.8% and recall is improved from 82.3% to 89.2% for the same speedup value of 35%. The new model shows a consistent improvement in precision and recall for the entire range and higher values are obtained as the threshold is increased. A more detailed analysis on the video sequences shows that the effects of pruning vary based on the scene. For example, 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. Fig. 5 shows the relation between speedup and accuracy for all six video sequences. This means that 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, etc. and therefore the resources have to be allocated on a frame-to-frame basis, based on the scene captured. A resource-aware many-core platform meeting the above requirements, is presented in

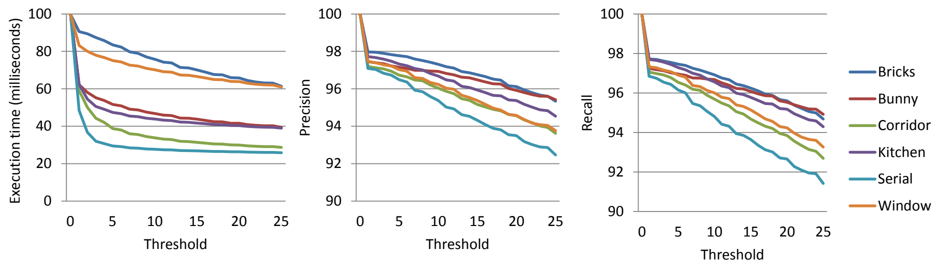

Fig. 5: Effects of pruning on processing time, precision and accuracy

Section 5 with emphasis on how to allocate and release resources in real-time based on application requirements.

5 Evaluation Platform

As described in Section 2, our work focuses on exploring the benefits of resource-aware Harris corner detection. Therefore, we implemented our algorithms on top of OctoPOS [16], a resource-aware operating system for Invasive Computing. OctoPOS shares the same view with ROS [14] as far as application-directed resource management of many-core processors is concerned. Also, both approaches resort to an event-based kernel architecture and largely benefit from asynchronous and non-blocking system calls. The main difference, however, is in the execution model of OctoPOS that was specifically designed to support invasive-parallel applications.

5.1 System Programming Interface

At the OctoPOS interface, resource-aware programming maps to three fundamental system calls: `invade()`, `infect()` and `retreat()`. The typical usage of these calls in the course of an application programm are depicted in Fig. 6. First, the application's

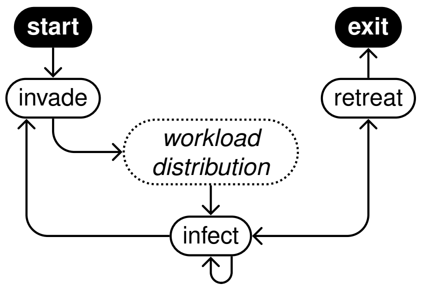

Fig. 6: Structure of an invasive program

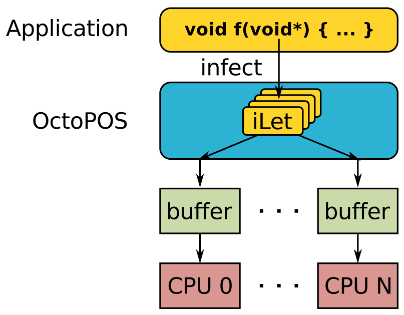

Fig. 7: Execution model of applications in OctoPOS

resource demand has to be expressed to the system. We call this the *invade* phase. It yields a set of resources in the form of a *claim*, the central data structure in the system for representing the resources associated with an application (processors, memory, etc.). Depending on the structure of the *claim*, the application has to distribute its workload accordingly. For example, it can tune its algorithms towards the number of processors present in the claim. Actual computation is then performed using the *infect* call. After execution finishes, another computation phase can be started on the same set of resources, resources can be released using *retreat*, or additional resources can be acquired using *invade*. The basic concept of Invasive Computing states that an application dynamically expands and shrinks its set of resources at runtime according to its own demand and that it can react to undersupply situations where not enough resources are available. Hence, depending on the current system state, the resulting claim may or may not fulfill the demands specified before. On the other side, once an application gets a claim, it gains full control over the associated resources. This guarantee on the acquired resources enables the application to balance its workload according to the dynamic runtime state of the system. Assumptions made during workload distribution before the *infect* phase hold until the application itself changes its resource allocation following the *infect* phase.

The main building blocks of applications in OctoPOS are so-called *i*-lets: Fragments of a program potentially executed in parallel with mostly run-to-completion semantics. These are represented by function and data pointers and thus are very lightweight entities. An *i*-let is like a Cilk procedure [5], but allows for the blocking of its executing thread by creating a “featherweight” continuation when actually releasing a PE. An application can create an arbitrary number of *i*-lets to be executed – potentially in parallel – using the *infect* system call. As depicted in Fig. 7, OctoPOS forwards *i*-lets to processor-local buffer queues for execution. Overall, this leads to an efficient implementation of *i*-let creation and dispatching. Moreover, with a tiled hardware architecture as described in Section 5.2, the buffering scheme is a possible candidate for hardware acceleration: To execute *i*-lets on distant tiles without obstructing the processors in the tile, the buffers can be maintained in hardware and accessed directly through the NoC. This leads to a very scalable system architecture especially suitable for many-core systems.

5.2 Hardware Architecture

Our target many-core processor consists of 9 tiles interconnected by a NoC. Each compute tiles consists of 4 cores interconnected by a local bus and some fast, on-chip tile-local memory, with a total of 32 cores (LEON3, a SPARC V8 design by Gaisler [6]) spread across 8 tiles. The 9th tile is a memory and I/O tile encompassing a DDR-III memory controller and Ethernet, UART, etc. for data exchange and debugging. Each core has a dedicated L1 cache while all the cores within a tile share a common L2 cache for accesses that go beyond the tile boundary to the external DDR-III memory. L1 caches are write-through and L2 is a write-back cache. Cache coherency is only maintained within the tile boundary to eliminate a possible scalability bottleneck when scaling to higher core counts. Therefore, data consistency has to be handled by the programmer through proper programming techniques that are built on top of hardware features to provide consistent data access and exchange between the different cache-coherency domains. This scheme is somewhat similar to the Intel SCC.

6 Implementation and Results

The first step in the resource-aware Harris detector is to allocate sufficient resources to perform corner detection within the interval specified by the user. The number of PEs required is calculated based on the input-image resolution, the processing interval, the nature of the scene, etc. The analysis starts with the generation of a differential image, where each pixel is computed using (4). 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, $limit$ is the maximum possible value generated by (4) 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. The number of PEs (N_{pe}) is calculated using (5).

$$N_{pe} \geq \frac{n \cdot T_{prn} + P_{pix}(th) \cdot T_{hcd}}{T_{exe} \cdot \eta(N_{pe})} \tag{5}$$

where n is the total number of pixels, T_{prn} is the processing time per pixel until the generation of integral histogram, P_{pix} is the number of pixels to be processed as computed by the pruning algorithm (a function of the threshold value th), T_{hcd} is the time to compute R for pixels with CR above threshold, T_{exe} is the processing interval and $\eta(N_{pe})$ represents the algorithm's efficiency as a function of degree-of-parallelism or available resources (N_{pe}). Including an efficiency factor is important as every additional *i*-let created by the algorithm also creates additional load on the external memory and shared communication interfaces, limiting the overall scalability. For the best results, the threshold value th can be set to zero so that the algorithm will attempt to process all pixels in the image. It should be noted that T_{prn} and T_{hcd} may vary based on the actual implementation and processor architecture. Hence these values are estimated by

Algorithm 1 Differential image

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

Algorithm 2 Integral histogram

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

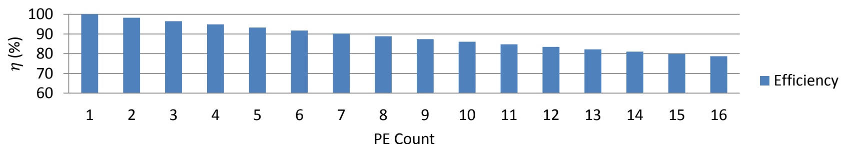

Fig. 8: Efficiency map for Harris detector on target hardware

profiling the application on the target platform. Fig. 8 shows the change in efficiency against degree of parallelism, on the target HW. It can be seen clearly that when the number of *i*-lets is doubled from 1 to 2, the execution time does not halve, but is reduced by a factor of 1.96, which means an efficiency of 98%. In the next step, an invade request (for N_{pe}) is raised and the OS makes a final decision on the number of PEs considering the current system load. 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 has to adapt to the limited resources offered by the runtime system by increasing the threshold value th until the condition in (6) is satisfied.

$$P_{pix}(th) \leq \frac{N_{pe} \cdot T_{exe} \cdot \eta(N_{pe}) - n \cdot T_{prn}}{T_{hcd}} \tag{6}$$

The new workload is processed by the allocated PEs and the resources can be released at the end of the processing interval or a new invade request can be raised if more PEs are required for the next frame due to a change in the scene. The behavior of the new resource-aware Harris detector is depicted in Fig. 9. The resource-allocation scheme remains same as described in Fig. 1. The execution-time profile in Fig. 9 shows that the resource-aware Harris detector can constrain the execution time per frame to the specified value of 100 milliseconds. The use of the conventional algorithm resulted in very high latencies under circumstances where sufficient resources are not available, and dropped frames occasionally. The values of precision and recall drops slightly in the region where the application has to adapt by pruning pixels. However, this helps to avoid overshoot in execution time and eliminate frame drops, so that results are consistently available within the predefined intervals. An overall comparison between the two scenarios is shown in Table 1. The use of the conventional algorithm leads to a very high worst-case execution time(WCET) and frame drops. The precision and recall values are low for the conventional algorithm as a frame drop leads to zero precision and recall for that

	Throughput	WCET	Precision	Recall
Conventional	**81%**	4.31x	0.82	0.81
Resource-aware	100%	1.04x	**0.98**	**0.98**

Table 1: Comparison between conventional and resource-aware Harris detectors

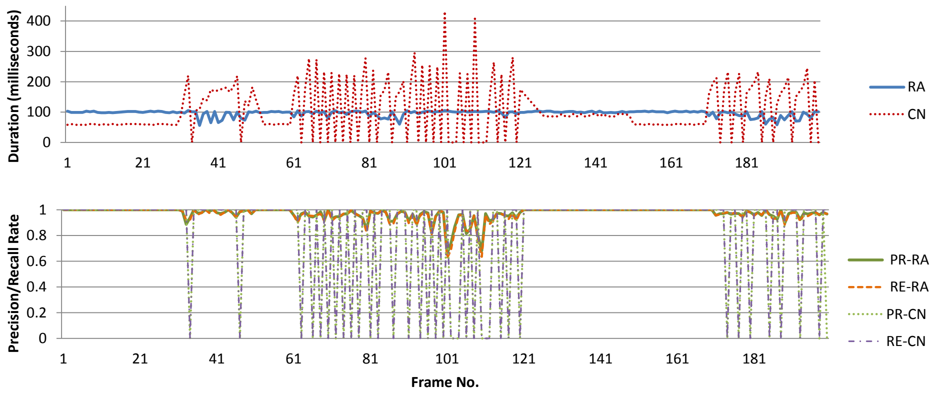

Fig. 9: Comparison between conventional and resource-aware model (RA = resource-aware-model, CN = conventional-model, PR = precision, RE = recall)

particular frame. In brief, the resource-aware Harris detector can operate very well under dynamically changing conditions by adapting the workload, avoiding frame drops and regulating the WCET, leading to high precision and recall rates.

7 Conclusion

This paper presented a resource-aware Harris corner detector and demonstrated how to estimate the resources required for corner detection based on the scene, the resolution of the input image and the user-specified time interval. The application is aware of available resources on the many-core processor and can adapt the workload if sufficient resources are not available. The enhanced corner detector can generate results within the specified search interval and avoid frame drops. Our experiments show that incorporating resource awareness into the conventional Harris detector can significantly improve the quality of the algorithm. A detailed evaluation was conducted on an FPGA-based hardware prototype to ensure the validity of the results. The results show up to 19% improvement in throughput and 18% improvement in accuracy as described in Section6. Though the evaluations were conducted using the OS and hardware explained under Section 5, the benefits are expected to be visible on any resource-aware platform including ROS [14]. The resource allocation and release happens once per frame and the additional overhead in execution time is negligible when compared to the time taken by the detector to process millions of pixels in every frame.

References

1. S. Alkaabi and F. Deravi. Candidate pruning for fast corner detection. *Electronics Letters*, 40(1):18–19, 2004.
2. T. Asfour, P. Azad, et al. ARMAR-III: An integrated humanoid platform for sensory-motor control. In *6th IEEE-RAS International Conference on Humanoid Robots*. IEEE, 2006.

3. 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.
4. S. Bell, B. Edwards, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. Digest of Technical Papers*, pages 88–598. IEEE, 2008.
5. R. D. Blumofe, C. F. Joerg, et al. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, 1995.
6. J. Gaisler and E. Catovic. Multi-core processor based on leon3-ft ip core (leon3-ft-mp). In *DASIA 2006-Data Systems in Aerospace*, volume 630, page 76, 2006.
7. S. Gauglitz, T. Höllerer, et al. Evaluation of interest point detectors and feature descriptors for visual tracking. *International journal of computer vision*, 94(3):335–360, 2011.
8. A. Gil, O. M. Mozos, M. Ballesta, and O. Reinoso. A comparative evaluation of interest point detectors and local descriptors for visual slam. *Machine Vision and Applications*, 21(6):905–920, 2010.
9. C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.
10. 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.
11. 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.
12. 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.
13. J. Klippenstein and H. Zhang. Quantitative evaluation of feature extractors for visual slam. In *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, pages 157–164. IEEE, 2007.
14. 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.
15. T. Mattson, M. Riepen, et al. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.
16. B. Oechslein, J. Schedel, J. Henkel, D. Lohmann, W. Schröder-Preikschat, et al. Octopos: A parallel operating system for invasive computing. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, 2011.
17. E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *Computer Vision–ECCV 2006*, pages 430–443. Springer, 2006.
18. 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.
19. A. Schmidt, M. Kraft, et al. An evaluation of image feature detectors and descriptors for robot navigation. In *Computer Vision and Graphics*, pages 251–259. Springer, 2010.
20. S. M. Smith and J. M. Brady. Susana new approach to low level image processing. *International journal of computer vision*, 23(1):45–78, 1997.
21. J. Teich, J. Henkel, A. Herkersdorf, W. Schröder-Preikschat, et al. 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.
22. M. Wu, N. Ramakrishnan, S.-K. Lam, and T. Srikanthan. Low-complexity pruning for accelerating corner detection. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 1684–1687. IEEE, 2012.