Invasive Computing for Robotic Vision

Johny Paul and Walter Stechele
Institute for Integrated Systems
Technical University of Munich
Germany
{Johny.Paul, Walter.Stechele}@tum.de

M. Kröhnert, T. Asfour and R. Dillmann
Institute for Anthropmatics
Karlsruhe Institute of Technology
Germany
{Kroehnert, Asfour, Ruediger.Dillmann}@kit.edu

Abstract— Most robotic vision algorithms are computationally intensive and operate on millions of pixels of real-time video sequences. But they offer a high degree of parallelism that can be exploited through parallel computing techniques like Invasive Computing. But the conventional way of multi-processing alone (with static resource allocation) is not sufficient enough to handle a scenario like robotic maneuver, where processing elements have to be shared between various applications and the computing requirements of such applications may not be known entirely at compile-time. Such static mapping schemes leads to inefficient utilization of resources. At the same time it is difficult to dynamically control and distribute resources among different applications running on a single chip, achieving high resource utilization under high-performance constraints. Invasive Computing obtains more importance under such circumstances, where it offers resource awareness to the application programs so that they can adapt themselves to the changing conditions, at run-time. In this paper we demonstrate the resource aware and self-organizing behavior of invasive applications using three widely used applications from the area of robotic vision - Optical Flow, Object Recognition and Disparity Map Computation. The applications can dynamically acquire and release hardware resources, considering the level of parallelism available in the algorithm and time-varying load.

I. Introduction

Multicore Processor System-on-chip (MPSoC) is gaining increasing interest in embedded computing, due to their enhanced performance capabilities. Some of the MPSoCs available today are Tileras 64-core processor [2], Ambric (with 336, 32-bit RISC processors) [17], an 80-core Intel prototype processor [21], Xetal-II [2] (a SIMD processor with 320 processing elements). Significant reduction in processing time could be achieved using such MPSoC architectures. For example a real-time optical flow implementation on Ambric is described in [11], with the algorithm statically mapped to the Processing Elements (PE). However, this static implementation has a rigid structure and it fails to operate if the predefined number of PEs are not available at an instance of time. It also does not explain how another application can run in parallel on the same MPSoC. Hence this is not flexible enough to handle dynamically changing applications, e.g. robotics.

Although all the above mentioned architectures offer the capability of exploiting different levels of parallelism, they all perform static resource allocation and hence the degree of parallelism for the applications has to be defined at compile-time. Early work on multicore programming and multicore compilers, including scheduling algorithm for multiprogramming in hard real-time [16], parallelizing programs for multiprocessors [19], and static scheduling algorithms for multiprocessors [14], has shown that static scheduling is not sufficient for compute intensive applications. Machine learning for multicore resource management has been investigated in [3], with results being applied in the Milepost compiler [9]. However, thread distribution requires off-line training, which seems not applicable for dynamically changing applications. In [6] management of shared resources in MPSoC has been investigated, i.e. shared caches, off-chip memory, taking into account bandwidth and power budget. However, a Neural Network has been applied, which requires off-line training which seems not feasible for advanced MPSoC in dynamically changing applications.

In contrast to the aforementioned approaches, Invasive Computing was introduced [20]. In brief, Invasive Computing is a methodology to manage and control the parallel execution of various applications on an MPSoC with many CPUs by giving the power to manage resources (i.e. computing, communication and memory resources) to the applications themselves and thus allow the running programs to manage and coordinate the resources by themselves in a decentralized manner. Through Invasive Computing, a given application program gets the ability to explore and dynamically spread its computations to neighboring processors in a phase called invasion, then to execute portions of its code in parallel based on the available cores. Afterwards, once the program terminates or enters a new phase with less parallelism, the program may enter a retreat phase, deallocating resources and resuming execution sequentially on a single core. This approach enables applications to self-explore the degree of parallelism available and to exploit dynamic resource requirements while avoiding fully centralized control of execution. Resource conflicts during invasion will be handled by an agent system, which is a topic for further research.

In this paper we will investigate principles of Invasive Computing applied on a humanoid robot like ARMAR III [4], who has to deal with various vision algorithms like stereo vision, object recognition, object grasping, obstacle detection, autonomous navigation etc. The load on the robots computing platform may vary from time to time, as various algorithms individually and in combination have to be loaded based on the operation performed by the robot at each instance of time.

II. Invasive Architecture

The Invasive architecture consists of heterogeneous tiles connected by a Network-on-Chip (NoC). Each tile can contain I/O units, loosely coupled RISC CPUs, or massively parallel processor blocks called Tightly Coupled Processor Arrays(TCPA) [12]. TCPAs consists of numerous light weight processing elements(PE). Image processing algorithms with regular loops for pixel processing highly benefit from TCPAs. Loosely coupled processors on the other hand can be used extensively for post-processing of the initial features computed by TCPAs. Additional hardware blocks called CIC(core ilet controller) help the operating system to efficiently map application threads based on the instantaneous system load. Figure 1 shows an architectural overview. More details about the hardware architecture are beyond the scope of this paper.

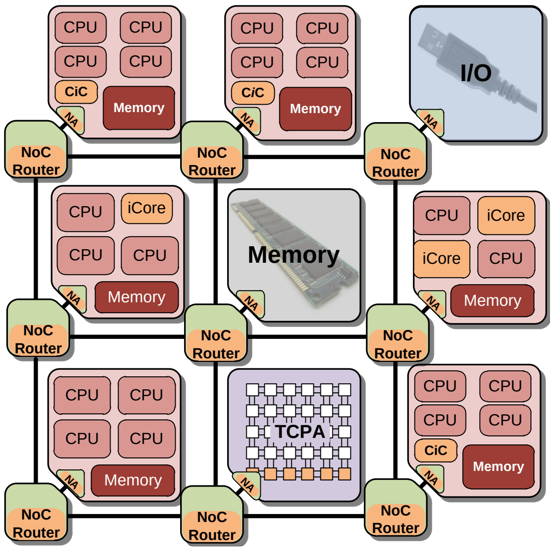

Fig. 1. InvasIC Architecture

III. Resource Aware Programming

In the recent past we have seen a growing number of applications which posses varying computing requirements at runtime. This includes optical flow computation (length of the optical flow vector is proportional to the search area) [8], an object recognition algorithm based on scale invariant SIFT features [5](more the number of features detected, higher the computing power required) and the computation of Another example is the disparity map computation in [18]. This algorithm works on large amount of data (stereo images) and a possible way to speed up such an algorithm is to copy its data to the tile-local memory (TLM) from the relatively slow main memory (DDR). But adapting the algorithm based on the number of available cores at the same time utilize the TLM in an efficient manner is challenging. How to model such applications when the underlying HW cannot scale at run-time, according to the changing requirements of the applications? How can we ensure that the applications get sufficient computing power so that the results are available on time? In this paper we present three case studies based on three different applications with different processing requirements, within the area of robotic vision. Using these case studies we demonstrate how Invasive Computing can solve the aforementioned problems by creating a feedback loop in the system using the resource exploration techniques.

A. Optical Flow on TCPA

Here we demonstrate the benefits of Invasive Computing, using the optical flow implementation in [8] due to its highly parallel implementation on FPGA. Also it posseses all the features described above and is widely used in robotic applications where the flow vector pattern can be used for robot navigation or obstacles detection. The optical flow algorithm is implemented on the TCPA. In order to self-explore resource availability in the neighborhood, a resource exploration controller (invasion controller) is integrated within each processing element (PE). Using these invasion controllers, different regions of available resources can be claimed, explored, and reserved by applications in a fast and distributed manner. The availability of 100s of PEs, along with the dedicated interconnect mechanism between neighboring PEs makes TCPA an idea platform to implement low levels pixel processing applications like optical flow.
The optical flow computation in [8] involves three stages, first of which is image filter to remove ambient noise. The image filtering algorithm involves a sliding 2D window of size $[3\times3]$ or bigger over the complete image. The window is placed, centered at every pixel in the image and each pixel is modified based on its neighboring pixels and the weights as in Equation (1) where $P(x,y)$ is the pixel at location [x,y], $W(x,y)$ is the corresponding weight and w is window size.

$$P_s(x,y) = \frac{\sum_{x_1=x-w/2}^{x+w/2} \sum_{y_1=y-w/2}^{y+w/2} P(x,y) * W(x,y)}{\sum W(x,y)} \quad (1)$$

Conventional way to increase parallelism/multi-threading for such applications is to split the image horizontally into sub-images and then assign it to threads. The individual results are combined later. But such implementations have two main problems. The results from processing sub-images have to be stored in external memory as there may not be sufficient cache in massively parallel processor arrays (MPPA) like TCPA. The partial results are combined by a separate thread, leading to two-stage processing. The second stage increases memory bandwidth requirements and this is a critical for large MPPAs as available bandwidth does not scale with PE count. A second problem arises as extra rows belonging to the neighboring sub-images have to be read by every thread while processing the pixels along border. This becomes critical for large windows, e.g. for an image with resolution 640×480 pixels, window size of 10×10 and 20 threads, each thread has to read 33% excess data to process its sub-section. All these factors reduces the scalability and performance benefits from using MPPAs. Also the other applications might slow down as they encounter excessive latency during memory access.

A careful inspection will show that for a sliding window

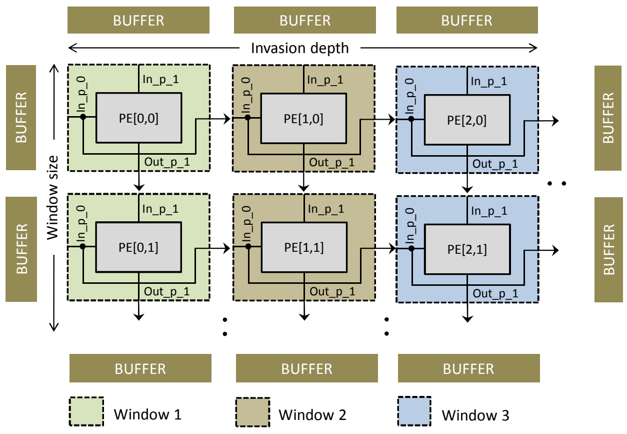

Fig. 2. Image filtering on TCPA

approach each window share its pixels with the neighboring window. Thus if multiple neighboring windows/pixels are processed simultaneously, it can lead to much better utilization of memory BW. To process one pixel for a window size of 3×3, we map the application onto a set of 3 PEs as shown in Figure 2. Each PE processes one row within the window and together they can process one complete window in few iterations. Moreover to exploit the parallelism in MPPA, multiple windows can be processed simultaneously. For example another set of 3 PEs, if available can process the neighboring window by reusing the data read by the first set of PEs, through continuous data exchange over the dedicated interconnect. This helps to reduce the overall execution time without increasing the memory BW requirements. The memory architecture and technique used to fill the pixel data into the I/O buffers in Figure 2 is beyond the scope of this paper and is explained in [15]. Equation 2 can be used to compute clocks/pixel(CPP) value for various configuration during image filtering, where id is $invasion\ depth$ (the number of windows processed simultaneously), C_p is the number of clock to process one row within the window, C_d is the number of clocks to discard pixel which does not belong to the current window and C_{const} is the time to reinitialize the variable after processing one window and move on to the next.

$$CPP = \frac{(C_p * w) + (id - 1)C_d + C_{const}}{id} \quad (2)$$

The second stage is census signature generation where a unique signature is computed for every pixel in a frame based on its neighboring pixels. This stage uses fixed window size as in case of image filtering and hence a mapping scheme similar to image filtering can be adopted. The third stage operates as follows. When two consecutive signature images are ready, the signature of every pixel in frame (t) is compared with signature of pixels in frame $(t+1)$. Since a comparison with every other signature is a computationally intensive task, the search region is limited to a small region in frame $(t+1)$ located around the candidate pixel from frame (t). More details about the optical flow implementation is beyond the scope of this paper and can be obtained from [8].

To detect fast motion, a wider search region is necessary. But this also leads to higher computing requirements as more signatures need to be compared. Hence, based on the scenario or application requirements, the seed PE raises a request to invade $[w \times id]$ region of the TCPA array, where w is the window size and id is invasion depth, very similar to image filtering. After infection, each PE processes its corresponding row within the 2D window. An incoming signature value from frame(t) has to be matched with a window of signatures from frame(t+1). In case of unique match, a vector is generated with its endpoints extending from center of the window to the location of the matching signature. In case of multiple match, a vector will not be generated. As one PE processes only a section of a complete window, the results are not final and have to be compared with results of other PEs for multiple match condition. The clocks/pixel values can be calculated using the Equation 3 where C_s is the clocks required to move a result from one PE to its neighbor over the dedicated interconnect. This equation shows that for an increasing w, the processing time can be constrained by varying the value of id.

$$CPP = \frac{(C_p * w) + (w - 1)C_s + C_{const}}{id} \quad (3)$$

In case of insufficient resource the application can decide whether to reduce the window size and hence avoid a frame drop or decide to drop the frame as it may not be possible to deliver the results on time. The resources will be released immediately enabling other applications to use it more efficiently.

Based on simulations, the behavior of Invasive Computing applications is analyzed using clocks/pixel (CPP) value as an index for evaluation. A low CPP value indicates low execution time or low power consumption (at reduced operating frequency). The graph in Figure 3 was plotted using Equations (2) and (3), for a fixed window size. C_{conf} is the time to configure the TCPA interconnect structure and load the executable code into the PEs. The values in Table I were obtained using a cycle accurate simulator published in [13]. Graph in Figure 3 indicates variation of CPP value against invasion depth(id) for image filtering, signature and vector generation stages of optical flow algorithm. The graph clearly shows that the CPP decreases with an increase in id. The execution time almost reduced to half upon moving from an invasion depth of one to two and the acceleration

TABLE I
SIMULATION RESULTS

Stage	C_p	C_s	C_d	C_{const}	C_{conf}
Image Filter	3	-	1	1	42
Census Sign Generation	2	2	-	9	105
Flow Vector Generation	2	2	-	4	101

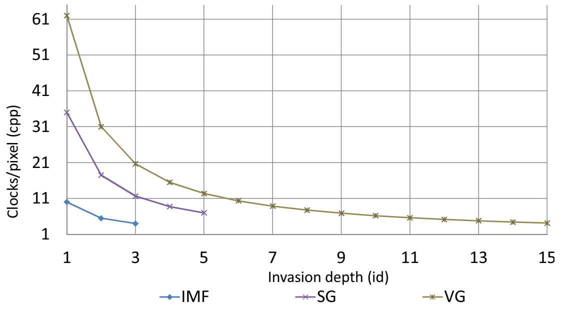

Fig. 3. Clocks/pixel Vs Invasion Depth (IMF: image filtering, SG: signature generation, VG: vector generation)

is consistent over the range of values for *id*. The results prove that our implementation is flexible enough to continue processing based on the available resources, which is not possible with the Ambric implementation in [11].

B. Object Recognition through Invasive Computing

Numerous object recognitions algorithms are available today, one of which is based on Harris features and SIFT descriptors [5]. This paper presents a combination of the Harris corner detector and the SIFT descriptor, which computes features with a high repeatability and very good matching properties in real-time. The objects to be recognized at run time are known beforehand and their features are stored in the database in a kd-tree structure. At runtime, the algorithm looks for matching features in the input frame to detect objects as shown in Figure 4. The left sub-image is the real-time input

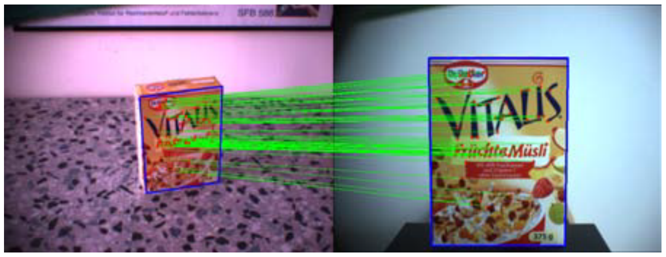

Fig. 4. Harris SIFT object recognition [5]

from the camera mounted on robot head while the template stored in database is shown on the right. The number of features to be matched is based on the features computed by Harris Corner Detection algorithm. Hence the overall features detected can vary based on the objects present in the frame, their orientations and lighting conditions. Hence the processing time also vary based on the varying conditions. We selected this application to demonstrate how such algorithms benefit from Invasive Computing while running on a set of loosely coupled RISC processors (this algorithm is not suitable for TCPA because of kd-tree based nonlinear search).
In Invasive Computing we try to constrain the processing time under varying load by acquiring and releasing processing elements at runtime. This means the application can request for extra processing elements if the features to be matched is above a threshold. For e.g, the feature matching can be performed on two RISC CPUs simultaneously each performing a kd- tree based search in one-half region of the frame. The partial results can be combined later. Another approach would be to acquire processing elements based on the objects stored in the database and the objects currently detected in the input frame. Most often, in a real-time implementation, an exhaustive search is not required in every frame, as there is a high probability for the previously detected object to exist in the current frame and a low probability for finding a new object in every new frame. Hence the algorithm can be optimized so that an exhaustive search is performed only once in every second, looking only for an already existing object in the rest of the frames. A simple example with 3 objects in the database, while only one is detected in the last frame, the total features to be matched can be reduced to one-third (assuming equal number of features from every object). Additional PEs will be acquired only when necessary and thus releasing them to other applications whenever possible. Moreover the exhaustive search can then be delayed if the PEs are overloaded with workload from other applications. This makes Invasive Computing significantly different from conventional multi-threading or multi tasking setup, where all the available PEs are shared by applications without having any notion about the availability of resource or current load. Our approach can ensure better distribution of application programs and load balancing, as the applications can breathe based on their processing requirements at the same time avoiding overloading of the PEs.

C. Disparity Map on RISC

This section shows how the calculation of a disparity map from a pair of stereo camera images can benefit from resource-aware programming. In contrast to the optical flow implementation, the disparity map algorithm is implemented in the X10 programming language [7] and is designed to run on invasive RISC CPUs. Invasive features are available by using the resource-aware programming methods and functional simulation tools presented in [10]. As the invasive compiler, operating system and hardware are currently in an early stage of development the programs are simulated on a desktop computer as a first step. A first working version of the mentioned components is expected next year.
Both a single-core and a resource-aware multi-core version of the disparity map algorithm were implemented in X10 based on the work published by Faugeras in 1993 [18]. As described in this paper, the algorithm operates on a pair of rectified stereo images. These images can for example be retrieved from the humanoid robot head of ARMAR-IIIa [4]. The output of the algorithm is a disparity map image with pixel colors correlated to the distance of the objects visible in the input images. Smaller values correspond to objects further away and are displayed in a darker gray color. To calculate this correlation value a sliding window is moved across the stereo images to find the best match. The IVT toolkit [1] provides a reference implementation of this algorithm in C++ which is used for

comparisons later on.

The single-core variant of the algorithm is implemented in X10 and matches the IVT implementation and processes the complete image in one large but efficient loop. However, it does not make use of resource-aware concepts and is tailored for architectures similar to those found in the simulation computer or in ARMAR-III. These architectures typically have large amounts of main memory which is accessible through a fast BUS interface. Embedded systems with possibly more cores but less main memory would not be able to execute this algorithm, at least not very efficiently. Due to higher restrictions in embedded domains it is therefore desirable to use resource-aware solutions which can exploit the constraints imposed by the underlying system. In order to show the benefits of resource-aware computing a multi-core capable version of the disparity map algorithm has been developed. It can adapt to different resource situations and is described in more detail below.

To achieve resource-awareness it is necessary for algorithms to react to changing requirements and available resources. As a basis for the following discussion and evaluation we use an architecture description for the simulator. The architecture consists of a main memory and two compute tiles being connected through a NoC. Each tile consists of 4 RISC processors and a block of tile-local memory (TLM). If this architecture is implemented in real hardware, the TLM has faster access times than the main memory.

It is easily recognizable that an algorithm running on one core and accessing only the main memory will have the longest execution time possible. The single-core version of the disparity map algorithm is an example for this category as it does not take advantage of the faster TLM or multiple CPU cores. Therefore it puts a lot of load on the NoC by constantly accessing the main memory.

Key to exploiting the available architecture and its resources is using a distributed approach. In our approach, the computation is split up into more fine-granular work packages. Once the partial results are ready they are merged into the complete result image. As the single-core version already provides a very efficient implementation it is used for processing the work packages. Algorithm 1 shows the complete listing of the different steps needed to calculate the disparity map. First of all, the maximum number of wanted PEs is requested in line 1. After that, the validity of the claimed resources is tested, followed by the most interesting part of the algorithm (lines 3 to 8). These lines show the adaptability to changing resource assignments. For distributing the image data both the count of available processing resources and the amount of available TLM are taken into account. This enables the algorithm to run on systems which are equipped with a small amount of TLM. Faster access times of the TLM are another advantage over constantly accessing the main memory. In the design of the Invasive Computing architecture all components are connected through a NoC which might slow down tasks with intensive I/O behaviour. The load of the NoC is greatly reduced by copying the necessary data into the TLM. In addition to that, the CPUs benefit from a shorter and faster BUS connection. However, the big difference between current algorithms using static resource allocation and resource-aware implementations becomes more obvious with more than one process running in parallel. In the case of static allocation the process is not able to perform its work if one or more resources are occupied. In contrast, the invasive approach can adapt to an ever changing resource situation. If a critical process is for example occupying half of the available resources, the invasive disparity map algorithm is able to adjust its degree of parallelism by only using the remaining free resources. During execution the request for additional cores and memory is only partially fullfilled. In this case the input and output images will either be split into larger chunks getting processed by each available core or into a lot of smaller chunks with each core queueing up several work packages. The version chosen depends on the amount of useable TLM and available processor cores.

Algorithm 1 Invasive disparity map algorithm

```
1:  $claim \leftarrow invade(maxPEs)$
2:  if $claim \neq invalid$ then
3:      $n \leftarrow claim.numberOfPEs()$
4:      $mem \leftarrow claim.localMemory()$
5:      $data[] \leftarrow divideIntoBlocks(imageData, n, mem)$
6:      for all $pe \in claim.PEs()$ do
7:          $pe.localMemory.data \leftarrow data[pe.id]$
8:      end for
9:      $infect(disparityMapFunction())$
10:     $synchronizeparalleloperations$
11:     for all $pe \in claim.PEs()$ do
12:         $result[pe.id] \leftarrow pe.localMemory.data$
13:     end for
14:     $retreat(claim)$
15: end if
```

A PC running Ubuntu Linux 8.04 and X10 (version 2.2.1 using socket communication) on a Core i7 (4 cores with Hyper-Threading @ 2.93GHz) processor is used to perform the evaluation. To achieve the best performance both the X10-runtime and the invasive simulator including the implementations are compiled from scratch with the C++ backend using the following flags `-Doptimize=true -DNO_CHECKS=true`. Measurement of the overall execution time is performed by executing 5 runs of each algorithm with one run consisting of 100 sequential executions of the algorithm on the same set of input images. Both IVT and single-core X10 versions of the disparity map algorithm can not use more than one PE as they are not capable of parallelizing their computation.

Table II lists the evaluation results. The first thing to notice is the IVT implementation being faster than the X10 implementations. Currently the X10 runtime does not implement a very efficient way to access Array data structures and therefore results in performance slowdowns. Additionally, the invasive algorithms are executed in a simulator accounting for parts of the execution time. Moreover, the invasive implementation is using invasive api calls such as $invade(), infect(), retreat()$ to allocate, use, and release resources compared to the single-core X10 implementation. Comparison of the execution times of both algorithms shows that the runtime-overhead of using the invasive api is relatively small. Considering that in the invasive version the image data blocks also need to be copied to and from TLM the overhead is comparably low. Considering the execution time of the invasive version of the algorithm the performance visibly increases with the number of used PEs. The usage of the Core i7 processor supporting Intel's

TABLE II
PERFORMANCE EVALUATION (UNITS: TIME $[ms]$, VARIANCE $[ms^2]$, STANDARD-DEVIATION (SD) $[ms]$)

Algorithm	#PEs	Mean	Min	Max	Variance	SD
IVT	1	38	37	90	9 - 73	3 - 9
X10	1	93	84	143	123 - 273	11 -17
Invasive	1	135	125	236	10 - 197	3 - 14
Invasive	2	138	125	193	8 - 120	3 - 11
Invasive	3	143	104	283	38 - 345	6 - 18
Invasive	4	144	131	283	12 - 353	3 - 19
Invasive	5	120	89	150	20 - 38	4 - 6
Invasive	6	105	98	134	11 - 38	3 - 6
Invasive	7	96	65	148	21 - 58	4 - 8
Invasive	8	90	81	213	15 - 309	3 - 18

HyperThreading technology enables the processor to schedule two threads on the same core simultaneously. This can on one hand boost the overall performance but can also hinder it if simultaneous threads need to use the same processing resources. Regarding the varying standard deviations the HyperThreading can probably be made responsible for this. When running on real hardware this problem is going to be eliminated. Thus the next step is running experiments on real hardware once the necessary components of the Invasive Computing project are ready.

IV. CONCLUSION

In this paper we have investigated a new paradigm of resource-aware computing, called Invasive Computing, on three case studies from Robotic Vision, optical flow on TCPA, disparity map calculation and object tracking both on RISC clusters. Although still work in progress, we could show self-organized adaptation of performance and functionality with respect to availability of computing resources through invasive programming. Multiple applications sharing same computing resources may dynamically scale their performance and functionality depending on the current state of the robot. In future work, we plan to extend our investigations from computing resources towards memory usage and on-chip communication.

V. ACKNOWLEDGMENT

The authors wish to thank Frank Hannig, Vahid Lahri and Sascha Roloff from University of Erlangen-Nürnberg, Germany, for their support in TCPA investigations and the invasive simulator as well as Prof. Gregor Snelting and Andreas Zwinkau for the invasive X10 framework. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89).

REFERENCES

[1] *http://ivt.sourceforge.net*.

[2] A. Abbo, R. Kleihorst, V. Choudhary, et al. Xetal-II: A 107 GOPS, 600 mw massively parallel processor for video scene analysis. *IEEE Journal of Solid-State Circuits*, 43(1):192–201, 2008.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, et al. Using machine learning to focus iterative optimization. 2006.

[4] T. Asfour, K. Regenstein, 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.

[5] 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.

[6] R. Bitirgen, E. Ipek, et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 318–329. IEEE Computer Society, 2008.

[7] P. Charles, C. Grothoff, C. von Praun, et al. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005.

[8] C. Claus, A. Laika, L. Jia, and W. Stechele. High performance FPGA-based optical flow calculation using the census transformation. *IEEE Intelligent Vehicle Symposium*, June 2009.

[9] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, et al. Milepost gcc: machine learning based research compiler. 2008.

[10] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau. Resource-aware programming and simulation of mpsoc architectures through extension of x10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '11. ACM, 2011.

[11] B. Hutchings, B. Nelson, S. West, et al. Optical flow on the ambric massively parallel processor array (MPPA). In *17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009.

[12] D. Kissler, F. Hannig, J. Teich, et al. A highly parameterizable parallel processor array architecture. In *IEEE International Conference on Field Programmable Technology (FPT)*. IEEE, 2006.

[13] A. Kupriyanov, D. Kissler, F. Hannig, and J. Teich. Efficient Event-driven Simulation of Parallel Processor Architectures. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 71–80. ACM Press, 2007.

[14] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[15] V. Lari, F. Hannig, and J. Teich. System integration of tightly-coupled reconfigurable processor arrays and evaluation of buffer size effects on their performance. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 528–534. IEEE, 2009.

[16] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 1973.

[17] B. Nelson, S. West, R. Curtis, et al. Comparing fine-grained performance on the ambric mppa against an fpga. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2009.

[18] O. Faugeras and B. Hotz and H. Mathieu and others. Real-time Correlation-based Stereo : Algorithm, Implementations and Applications. Technical Report 2013, INRIA, 1993.

[19] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 637–646. ACM, 1989.

[20] J. Teich. Invasive Algorithms and Architectures. *it–Information Technology*, 50:5, 2008.

[21] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, et al. An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In *Solid-State Circuits Conference, 2007*, pages 98–589. IEEE, 2007.