RESOURCE-AWARE DYNAMIC TASK-ALLOCATION IN CLUSTERS OF EMBEDDED SMART CAMERAS BY MOBILE AGENTS

M. Bramberger[1,2], B. Rinner[1], and H. Schwabach[2]

[1] Graz University of Technology , [2] ARC seibersdorf research, AUSTRIA

ABSTRACT

This paper presents a dynamic task allocation method for smart cameras targeting traffic surveillance. Since our target platforms are distributed embedded systems with limited resources, the task allocation has to be light-weight, flexible as well as scalable and has to support real-time requirements. Therefore, surveillance tasks are not allocated to smart cameras directly, but to groups of smart cameras, so called surveillance clusters. We formulate the allocation problem as a distributed constraint satisfaction problem (DCSP) and present a distributed method for finding feasible allocations. Finally, a cost function is used to determine the optimal allocation of tasks. We have realized this dynamic task allocation using heterogeneous, mobile agents which utilize their agencies and our embedded software framework to find the most appropriate mapping of tasks in a distributed manner. The dynamic task allocation has been implemented on our smart cameras (SmartCam) which are comprised of a network processor and several digital signal processors (DSPs) and provide a complex software framework.

1. INTRODUCTION

Surveillance systems currently undergo a dramatic shift. Traditional surveillance systems of the first and second generation have employed analog CCTV cameras, which transferred the analog video data to digital base stations where the video analysis, storage and retrieval takes place. Current semiconductor technology enables surveillance systems to leap forward to third generation systems which employ digital cameras with on-board video compression and communication capabilities. *Smart cameras* (Wolf et. al (9), Bramberger et. al (2)) even go one step further; they not only capture and compress the grabbed video stream, but also perform sophisticated, real-time, on-board video analysis of the captured scene. Smart cameras help in (i) reducing the communication bandwidth between camera and base station, (ii) decentralizing the overall surveillance system and hence to improve the fault tolerance, as well as (iii) realizing more surveillance tasks than with traditional cameras.

Typical tasks to be run in a surveillance system targeting traffic surveillance include MPEG-4 video compression, various video analysis algorithms such as accident detection, wrong-way drivers detection and stationary vehicle detection. Additionally, traffic parameters such as average speed, lane occupancy and vehicle classification are often required. Since computing power is limited we may not allocate all tasks to the cameras.

This paper presents a resource-aware dynamic task allocation system for smart cameras targeting traffic surveillance. Since our target platforms are distributed embedded systems with limited resources, the task allocation has to be light-weight, flexible and scalable as well as has to support real-time requirements. Therefore, surveillance tasks are not allocated to smart cameras directly, but to groups of smart cameras, so called surveillance clusters. We formulate the allocation problem as a distributed constraint satisfaction problem (DCSP) and present a distributed method for finding feasible allocations. Finally, a cost function is used to determine the optimal allocation of tasks. We have realized this dynamic task allocation using heterogeneous, mobile agents which utilize their agencies and our embedded software framework to find the most appropriate mapping of tasks in a distributed manner

The main contributions of this ongoing research project include (i) the distributed agent-based approach for determining all feasible allocations of a DCSP, (ii) the evaluation of a complex cost function considering the limited resources of the embedded platform in detail, (iii) the integration of a mobile agent system in our embedded software framework. The dynamic task allocation has been implemented on our smart cameras (*SmartCam* - Bramberger et. al (3)) which are comprised of a network processor and several digital signal processors (DSPs) and provide a complex software framework.

The remainder of this paper is organized as follows: Section 1.1 sketches related work. Section 2 briefly presents hardware and software of our SmartCam. Section 3 discusses the advantages of grouping smart cameras to surveillance clusters. Section 4 describes the DCSP approach and focuses on finding feasible allocations of tasks and the cost function. Section 5 and 6 present the implementation and the experimental results, respectively. Section 7 concludes the paper with a summary and an outlook on future work.

1.1 Related Work

Agents-based load distribution has been an active research in the last decade. An overview of agent standards and available platforms is presented in

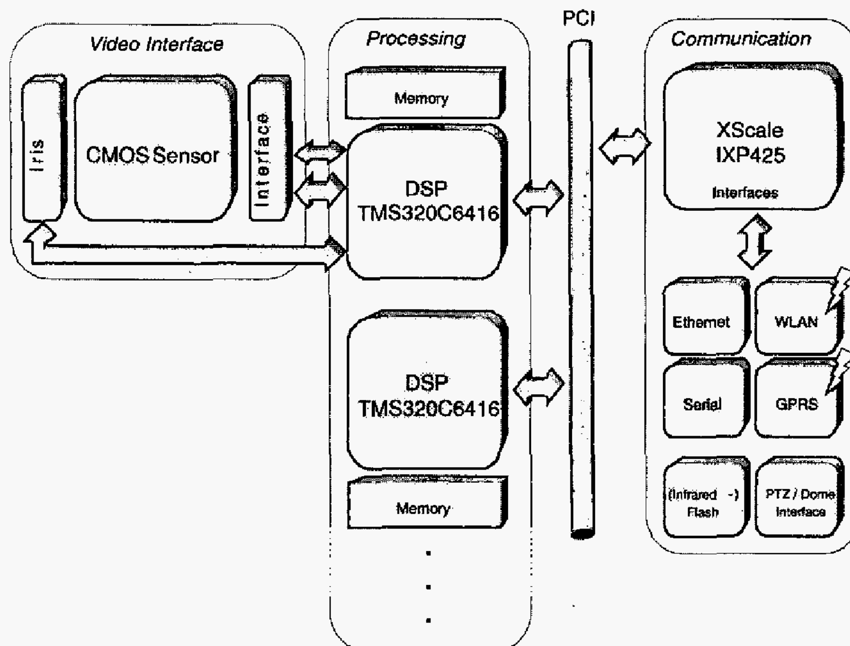

Figure 1: The hardware architecture of the smart camera

Perdikeas at. al (6). WYA (While You're Away - Suri et. al (8)) is based on the NOMADS mobile agent system. It provides dynamic load balancing by mobile agents, which utilize a central coordinator to move between workstations. Qin et al (7) identified the amount I/O-traffic as an important issue for dynamic load balancing beside CPU-load and memory. Chow and Kwok (4) introduced the affinity of an agent to its machine and used a credit-based scheme to determine the agents to be migrated by using a central host station. Agents used in surveillance systems were proposed by the MODEST consortium (Abreu et. al (1)), where mobile agents were used to track vehicles using PC based platforms.

Distributed constraint satisfaction problems have been analyzed by Yokoo et. al. (12) for variable-split CSPs. The presented asynchronous backtracking algorithm used autonomous agents, however, the high communication effort between the agents is impractical for real-time systems.

2. THE SMART CAMERA

Smart cameras are the core components of a 3rd generation surveillance system. These cameras perform video sensing, high-level video analysis and compression and transfer the compressed data as well as the results of the video analysis to a central monitoring station. The video analysis tasks implemented in the cameras clearly depend on the overall surveillance application and may include accident detection, vehicle tracking and the computation of traffic statistics. Most of these tasks, however, require a very high computing performance on the cameras.

2.1 Hardware Architecture

Our smart camera has been designed as a low-power, high-performance embedded system. As depicted in figure 1, the smart camera consists of three main units. (1) The sensing unit, (2) the processing unit, and (3) the communication unit.

A high-dynamic, monochrome CMOS image sensor is the heart of the sensing unit. It delivers images with VGA resolution at 25 frames per second via a FIFO memory to the processing unit. Real-time video analysis and compression is performed at the processing unit which is equipped with up to four digital signal processors (DSPs) TMS320C6415 from Texas Instruments. The DSPs deliver an aggregate computing performance of almost 20 GIPS while keeping the power consumption low. The DSPs are coupled via a local PCI bus which serves also as connection to the network processor (Intel XScale IXP425) in the communication unit. The communication unit provides access to the camera's environment. The communication of the smart camera is basically two-fold. First, the communication unit manages the internal communication between either the DSPs and the DSPs and the network processor. Second, it manages the external communication, which is usually IP-based. The XScale processor is operated by Linux due to large number of available tools and applications available under this operating system. (3) provides a more detailed insight into the hard- and software architecture of the smart camera.

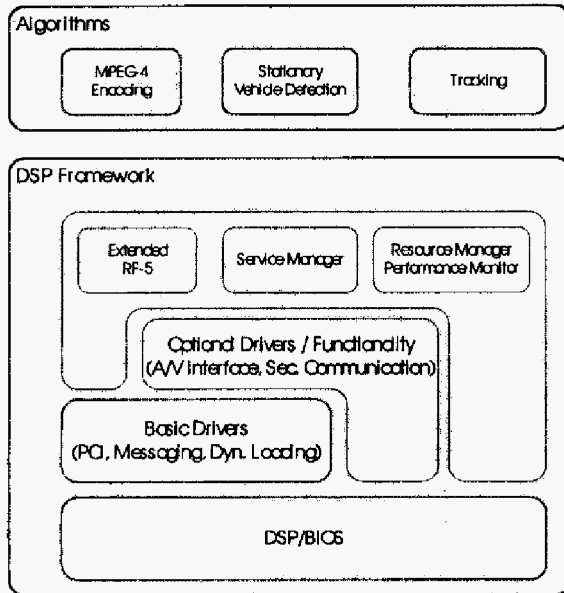

Figure 2: The DSP Framework

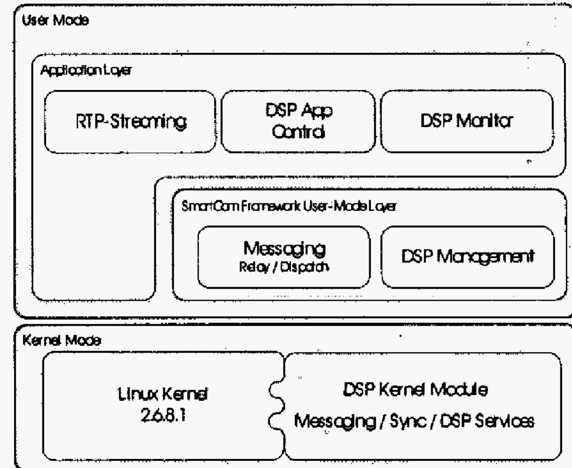

Figure 3: The SmartCam Framework

2.2 Software Architecture

The software architecture of our smart camera is designed for flexibility and reconfigurability. The software architecture consists of several layers which can be grouped into: (1) The DSP framework, which is implemented on the DSPs, and (2) the SmartCam framework, running on the network processor.

DSP Framework The main purpose of the DSP framework (cp. figure 2) is (i) the abstraction of hardware and communication channels, (ii) the support for dynamic loading and unloading of applications, and (iii) the management of on-chip and off-chip resources of the DSP system by utilizing Texas Instruments Reference Framework 5 (see Mullanix et. al (5)).

SmartCam Framework The SmartCam framework (cp. figure 3) serves the following purpose: First, it provides abstraction of the DSPs to ensure platform independence of the agent-system and application layers. Second, the application layer uses the provided communication methods (messaging to the DSPs and IP-based communication to outer world) to exchange information, or work as a relay service, respectively. Finally, the agent-system layer is run on top of Java, whereas the agents are run as a part of the agent platform.

3. SURVEILLANCE ARCHITECTURE

The architecture of the surveillance system consists of a large number of smart cameras deployed alongside highways or in tunnels. Since these smart cameras have limited computational resources, it is not possible to run all required surveillance tasks on a smart camera. Therefore, physically co-located smart cameras are combined into logical groups, so called *surveillance clusters*. Consequently, sets of surveillance tasks (e.g. accident detection, vehicle counting, and vehicle classification) are then allocated to surveillance clusters. This is feasible, since events, observed by co-located cameras are causally associated with each other. Therefore, it is not important on which smart camera a surveillance task is allocated, as long as these surveillance clusters do not span a too large area. However, not all surveillance tasks require small surveillance clusters; classifying and counting of vehicles, for example, may be spread over a larger area, while accident or fire detection tasks are usually allocated to smaller clusters. Therefore, a smart camera may be a member of several surveillance clusters (cp. figure 4). The allocation of surveillance tasks to smart cameras is done dynamically during runtime by the presented task allocation system (see section 4), which is distributed over all smart cameras.

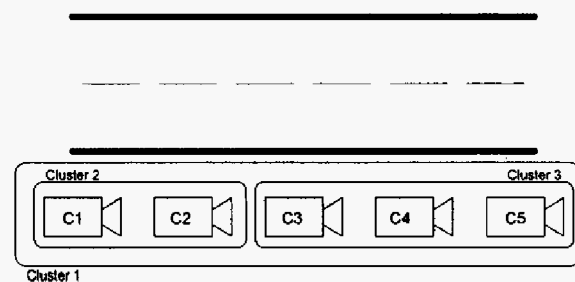

Figure 4: A tunnel surveillance scenario with three surveillance clusters

In contrast to the presented surveillance tasks, which may be allocated to any smart camera in the surveillance cluster, there are also several surveillance tasks which are required to run on a specific smart camera. These tasks possess an affinity to a scene or a camera, respectively. Tracking algorithms, for example, require to be run on a specific camera, which observes the tracked object. These scene-affine tasks are also

contained in a surveillance cluster, however, the task allocation system should allocate these tasks to the required smart cameras. Note that scene-affine tasks might be allocated to a different camera. In this case, however, the communication load would significantly be increased because raw images would have to be transferred from one camera to another.

4. TASK ALLOCATION

The goal of the task allocation system is find a mapping of tasks to smart cameras which satisfies all requirements and is optimal with respect to a specified metric, i.e. some cost function. Since the surveillance tasks have firm real-time requirements, the task allocation system has to take care that no deadlines are missed due to an overload of a camera or a cameras subsystem, respectively.

The re-allocation of tasks may be necessary due to events, raised by hardware or software: (1) Hardware events usually originate from changed resource availability due to added or removed cameras, hardware breakdown, or re-usability of recovered resources. (2) Software events are caused by changes to resource requirements due to changes in the task set of the surveillance cluster, or because of changes in the quality-of-service level (QoS) of tasks, i.e., due to detected events in the observed scene.

The allocation of tasks to smart cameras is done in two steps. (1) In the first step, all feasible allocations of tasks to smart cameras (allocations where no real-time requirements are violated) are determined. (2) In the second step, the optimal allocation of tasks is chosen by using a cost function.

4.1 Find Feasible Allocations

The determination of feasible allocations of tasks to smart cameras is a distributed constraint satisfaction problem (CSP) (12). CSPs are defined by a set of n variables $T=\{T_1,...,T_n\}$, which hold values of a finite domain D, and a set of k constraints $C=\{C_1,...,C_k\}$. In our case, the surveillance tasks to be allocated to the cameras of the surveillance cluster are the variables, and the values they hold is the identifier of the allocated smart camera $D=\{1,...,m\}$. Therefore, $T_i=d$ indicates, that task i is currently allocated to smart camera d.

Equation 1 determines all permutations with repetitions A, denoted by $\prod_R$, from the m cameras and n tasks. The subsets of A are allocations of tasks to cameras, which have to be checked for feasibility. The constraints (cp. eq. 4 are formulated using two functions which determine the resource requirements and availability of the tasks and the cameras. *req(Res, i)* determines the requirements of task i concerning resource *Res*, while *avail(Res, d)* determines the availability of the resource *Res* on smart camera d.

$$A=\prod_R(m,n)=A\{A_1,...,A_p\};p=m^n \tag{1}$$

$$A_j=\{a_i,...,a_n\}\mid a_i\in D \tag{2}$$

$$R=\{CPU,MEM,DMA\} \tag{3}$$

$$C=\{\forall A_j\in A:\forall d\in D:\forall r\in R:\forall a_i\in A_j: \forall a_i=d:(\sum_i req(r,i))<avail(r,d)\} \tag{4}$$

Equations 1 to 4 define a feasible allocation, i.e., any allocation A which does not violate any resource constraint C. Note that in our implementation we also considered the DSPs memory hierarchy and several parameters of the DMA subsystem.

However, this simple approach requires a central host. A more scalable solution is to distribute the CSP to several cameras in the surveillance cluster.

The distributed CSP. In order to distribute the determination of feasible allocations to all smart cameras, we split the domain D into m sub-domains $D_1,...,D_m$, each comprised of a single value $D_l=l$. The set of tasks T remains the same, while the complexity of the constraints is reduced slightly (cp. equation 5).

To achieve all possible allocations of tasks, we choose $\forall 0<i\leq n$ tasks out of the set of all tasks $\forall 0<i\leq n: A_i=choose(i,n)$. Every set A_i is comprised of ν allocations $A_i=\{A_1^i,...,A_\nu^i\}$, each of which consisting of i tasks $A_\nu^i=\{a_1,...,a_i\}\mid\forall a_\eta\in T$. All allocations A_ν^i, which meet the constraints (cp. equation 5) for all resources R, are chosen as feasible, partial allocations.

$$C=\{\forall A_i\in A:\forall A_\nu^i\in A_i:\forall a_j\in A_\nu^i: \forall r\in R:(\sum_i req(r,a_j))<avail(r,d)\} \tag{5}$$

Note that the constraints only consider tasks allocated to a single smart camera, so the CSP is split into m sub-CSPs, which can be solved in parallel on the m smart cameras in the surveillance cluster.

Normally, CSPs require that all variables are included in the solutions of the problem, however, in our case this rule would imply that only partial allocations are valid, which include all n tasks of the surveillance cluster. In order to accept task allocations which do not include all tasks, the CSP rules have to be weakened. Even the allocation of no task is a valid allocation, which is reasonable, if a surveillance cluster contains fewer tasks than smart cameras.

Merging of Allocations. Finally, m sets of partial task allocations are available $P=\{P_1,...,P_m\}$; $P_j=\{P_1^j,...,P_\nu^j\}$. In order to find all feasible task allocations these partial allocations have to be merged.

Consequently, valid task allocations

1. do not allocate tasks to more cameras concurrently, and

2. include all surveillance tasks.

The merging of partial allocations can be done in parallel, as long as rule 1 is not violated. However, the final merging process has to obey rule 2 too.

The combination of the partial allocations finally provides a set of f feasible allocations F, from which the most appropriate allocation has to be chosen.

4.2 Find Optimal Allocation

For finding the most appropriate allocation of tasks a cost-scheme is used. This cost-calculation scheme includes five cost classes, namely (1) resource costs - C_R, (2) data-transfer costs - C_T, (3) migration costs - C_M, (4) affinity costs - C_A, and (5) quality-of-service costs - C_{QoS}. These costs are calculated for every task on every smart camera in the surveillance cluster. Finally, the overall costs are accumulated as enlisted in the set of feasible allocations F.

Cost Calculation. To compute the overall cost C_{Tot} of a surveillance task, the cost values of the five cost classes are weighted and added:

$$C_{Tot} = k_R * C_R + k_T * C_T + k_M * C_M + k_A * C_A + k_{QoS} * C_{QoS} \quad (6)$$

Resource Cost. Highly-optimized applications for DSP-based embedded systems typically utilize the memory hierarchy and on-chip resources such as direct memory access and timers to exploit the processors performance. In order to fully exploit the performance capabilities of a DSP, it is necessary to carefully allocate these limited resources to applications. In order to guarantee resources to an application, the DSP framework has to keep track of the status of these DSP resources.

Actually, the system considers the following resources: (1) CPU load, (2) memory usage (including memory hierarchy on DSPs), (3) DMA utilization (including channels, reload-tables, and transfer complete codes), (4) memory bus utilization, (5) communication ports, (6) timers, and (7) interrupts.

$$C_R = \sum_{\forall R_i} C_{R_i} * k_{R_i} \quad (7)$$

where

$$C_{R_i} = \frac{\text{Resource Requirements}}{\text{Resource Availability}} \quad (8)$$

As denoted in equation 8, the resource costs are not only composed by sufficient availability of resources, but they are computed as the ratio of resources required to resources available.

Note the also the utilization of the memory bus is taken into consideration, since an overload of the memory bus usually results in violating real-time deadlines.

Data-Transfer Cost. We have defined two different types of data transfers: (1) Data entering or leaving the system will usually be transmitted using a network interface, while (2) intermediate data transfers operate internally via PCI bus or directly to memory. To consider the bandwidth of the used interface, the data-transfer cost C_T is calculated as the amount of megabytes (MB) transferred, multiplied by a slowness factor depending on the bandwidth of the connection.

Migration Cost. The migration of tasks between smart cameras is accounted for by two costs, which are scaled and added to receive the migration cost C_M.

$$C_M = k_{MT} * C_{MT} + k_{MI} * C_{MI} \quad (9)$$

1. Migration Transfer Cost - C_{MT} Data transfer costs are raised, since the task including the intermediate results has to be transferred to the new smart camera. These transfer costs are calculated similar to the data-transfer costs as the amount of to be transferred megabytes (MB) is multiplied by the appropriate slowness factor.
2. Migration Idle Cost - C_{MI} In many cases the restart of a video analysis algorithm after migration will require a certain amount of time due to a learning or initialization phase. The migration idle cost considers the time required by the algorithm to become fully functional again. It is set to low values for algorithms with no or little initialization or learning phase like video encoding agents. In contrast, video analysis algorithms which have to setup background models or learn a certain behavior will have a high migration idle cost to assure a high level of usability as a high number of migrations would result in many periods of non-functionality.

Affinity Cost. The distribution of tasks inside a surveillance cluster is controlled by two affinity costs, which are defined by the task designer.

1. **Cluster Affinity.** As the overall load of a surveillance cluster increases, it may not be possible to run all tasks due to insufficient computing resources. The cluster affinity introduces a priority scheme, whereas higher affinity values denote higher priority. Therefore tasks with high cluster affinity values will be allocated to smart cameras more likely.
2. **Scene Affinity.** As mentioned in section 3, agents may require to process video data from a special smart camera. Tracking applications, for example, require to process video frames which contain the tracked object. Such agents feature high scene-affinity values, which increases the probability that the agent will be executed on the appropriate smart camera. However, if the agent is run on another smart camera in the system, the video data from the required scene has to be forwarded to the surveillance task.

Based on these two affinity values, the affinity costs - Cluster Affinity Cost C_{AC} and Scene Affinity Cost C_{AS} - are calculated, whereas the higher cost is selected as the overall affinity cost.

$$C_A = \max(k_{AC} * C_{AC}, k_{AS} * C_{AS}) \qquad (10)$$

As higher affinity values denote lower cost, the affinity costs are normalized to the maximum affinity. If the cluster or scene affinity is zero, it will not be considered in the cost calculation. Therefore, if both affinities are zero, the affinity cost is set to ∞.

$$C_{AC} = \frac{\text{Maximum Affinity}}{\text{Cluster Affinity}} \qquad (11)$$

$$C_{AS} = \frac{\text{Maximum Affinity}}{\text{Scene Affinity}} * k_L \qquad (12)$$

The scaling factor k_L will increase the scene affinity cost be a factor k_{RP} if the agent is not executed on the required smart camera. Otherwise it will be set to *1*. k_{RP} is called *Remote Penalty* and specifies the penalty induced by remote execution of agents featuring scene affinity.

Quality-of-Service Cost. The task allocation system considers all quality-of-service levels, therefore, we have to provide a possibility to choose which QoS-level should be used. It is desired to run all tasks with highest possible quality, therefore this cost adds penalty costs, if the task should be run with lower QoS-levels. These penalty costs and the number of QoS-levels are defined by the task designer.

Since lower QoS-levels lead to lower resource requirements of tasks, the number of possible task allocations will rise. Therefore a high number of QoS-levels yield longer run-times of the task allocation system.

4.3 Reallocation Scenarios

As mentioned before, two scenarios are possible, where a reallocation of tasks is necessary. However, we handle the scenarios differently to improve the performance of the system.

Increased Load. Increased requirements or decreased resource availability indicates a higher system load. Therefore, the number of feasible allocations will decrease, since less combinations of tasks will satisfy the constraints of the CSP. Therefore, the value of the changed resource or availability is updated in the set of feasible allocations *F*. Finally, the feasible allocations are re-checked and infeasible allocations are removed from *F*.

This re-checking process does not require recalculating all possible allocations, therefore, the time required for determination of the new allocation is reduced dramatically.

Reduced Load. In contrast to the upper scenario, the decrease of requirements or increase of resource availability can be seen as a reduced load of the system. Consequently, the number of both, partial and complete, allocations will increase. Therefore, all feasible allocations have to be re-computed. This approach leads to longer execution times, however, as the service quality will rise by this reallocation, the real-time deadlines are not as firm as in the upper scenario.

Implementation

For the implementation of our surveillance system we have chosen to use mobile agent technology. Mobile agents are most suitable for our system, since they support mobility, autonomy, and platform independence.

4.4 Agent System

We have chosen to use the diet-agents (see http://diet-agents.sourceforge.net) system since it includes all required features like mobility, autonomy, and platform independence and is reasonable small, which is inevitable for embedded systems.

However, extensions to the agent system were necessary to add the support for the DSPs, and the decentralized management of the surveillance clusters.

DSP Agencies. Agencies provide the environment for agents to live in. An extension to standard agencies is required to support multiple surveillance clusters. Therefore, a logical grouping of agents within the agency (cp. surveillance clusters A, B and x in figure 5) is desirable. We have implemented a *cluster-information agent* (CIA), which hosts local knowledge of other cameras belonging to the surveillance cluster, and provides lookup services for agents within the surveillance cluster.

The integration of the DSPs into the agent system is done by one *DSP integration agent* (DIA) per DSP in the system. This DIA is a stationary agent, which manages message registrations, message dispatching,

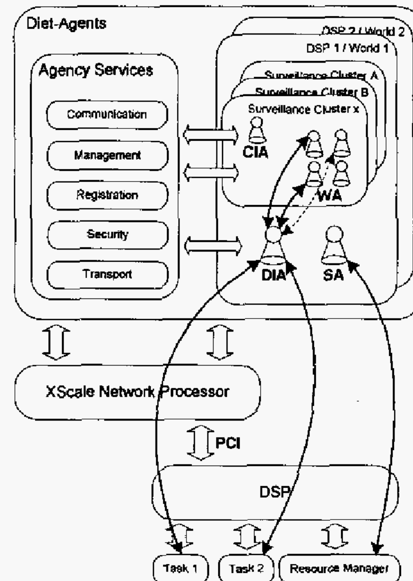

Figure 5: A DSP agency

and binary downloads to the DSP. Finally, the *System Information Agent* (SA) gathers the system status and performance data from the DSPs and the network processor.
Two basic agent types are deployed within the agent system: (1) Management-oriented SmartCam-agents, which are not included in the task allocation system, and (2) DSP-agents, which implement algorithmic functionality to be executed on a DSP. Since the implementation of the SmartCam agents is quite straightforward, only the DSP agents will be discussed in detail in the next section.

DSP Agents. DSP agents are an enhancement of usual agents, as they are the base of the task allocation system. Therefore, DSP agents (depicted as *Worker Agents (WA)* in figure 5) include their requirements to the system and their cost parameters. Additionally, the cost calculation functionality is added to these agents (cp. figure 6). As the computational intensive parts of these agents are executed on the DSPs, the agents contain a binary, which is downloaded to the DSP and dynamically integrated into the system by the DSP framework. Therefore, the agents maintain communication channels to the DSP by using the world's DSP integration agent (DIA).
In case of migration the agent notifies the DSPs task to stop computation and to transmit the persistent intermediate results to the agent. After migration the agent loads the binary to the DSP and transmits the stored intermediate results to the DSP. This enables the algorithm to continue computation or to use the intermediate results as new starting values, respectively.

4.5 Task Allocation System

Determining Partial Allocations. We are using a backtracking-based algorithm in order to determine the feasible allocations of tasks to a smart camera. Figure 7 shows two solution trees (containing all solutions) from a partial CSP. Since the order of the tasks is not relevant, the redundant edges (dashed) (figure 7(a)) are removed (figure 7(b)). In order to improve performance,

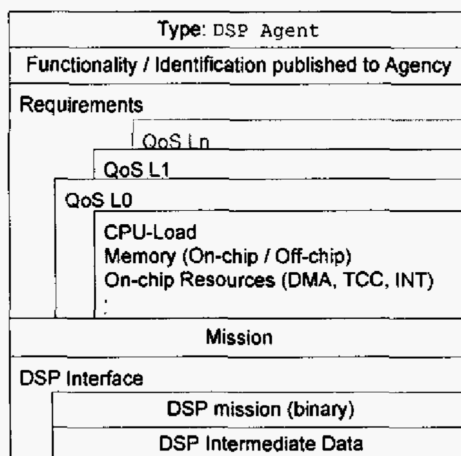

Figure 6: A DSP agent

Backtracking means, that a certain sub-tree is not further evaluated, as soon as an infeasible allocation is found. Supposing *AB* is not a feasible solution, *ABC* would not be tested for feasibility.

Merging of partial allocations. As mentioned, the constraint-satisfaction problem is solved on all smart cameras in parallel. Consequently, the determined partial allocations have to be merged in order to receive a feasible mapping. Therefore, *allocation-merging agents (AMA)* travel from camera to camera and merge the partial solutions into a global one. Since the merging is also done in parallel, agents are best suitable for merging, due to three features of agents: (a) communication to gather all required information, (b) mobility to migrate between the cameras, and (c) autonomous behavior to prevent a camera of being the control host.
During merging, trees (as depicted in figure 7(b)) are merged left to right. The merging also makes use of a backtracking algorithm, since the merging of a sub-tree

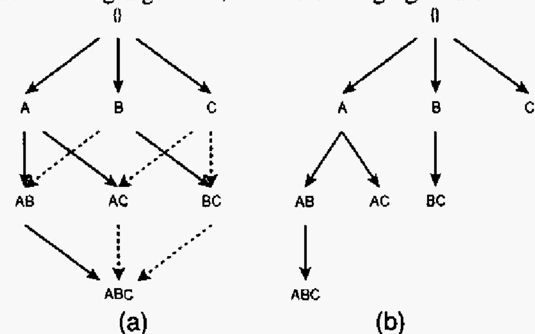

Figure 7: Possible Partial Allocations

is stopped, as soon as its root is not a feasible allocation. The number of feasible allocations after merging rises significantly.

Reconfiguring a surveillance cluster. This section outlines the actions taken during a reconfiguration of a surveillance cluster. Since our goal is a flexible, scalable and distributed implementation we have chosen to realize the task allocation system using mobile agents.
A new allocation is always initiated by a smart camera, either due to a hardware event (changes of availability) or a software event (changes of requirements). This camera initiates the reallocation by broadcasting the *request-for-requirements* to all worker agents within the surveillance cluster. After the agents receive this request, they broadcast their requirements to *all* smart cameras in their surveillance cluster. Additionally, all agents calculate their costs for every smart camera and transmit the cost values to the initiating camera. The smart cameras determine in parallel the *partial allocations*, which have to be merged in the next step. All smart cameras, which are equipped with more DSPs, have to merge the partial allocations for the DSPs first. In order to further merge the partial allocations, the half of the smart cameras (determined during the creation of the surveillance cluster) create allocation-merging

agents (AMA), which pick up the merged partial allocation and migrate to predetermined smart cameras (which comprise the other half of the surveillance cluster), where they grab the local partial allocation, and merge both, the local and the included partial allocations. After the merging process, the agents migrate to the next smart camera, as defined by the fixed itinerary, and merge their partial allocation with the partial allocation provided by another AMA. This way, the partial allocations are merged to a final allocation, whereas the last allocation merge is done on the initiating smart camera. The merging process corresponds to a binary tree, therefore $ld(m)$ (where m is the number of smart cameras) sequential merging steps have to be done.
Finally, the initiating smart camera has to determine the most appropriate task allocation, therefore, the costs, as submitted by the agents, are accumulated for every task allocation. Consequently, the solution with the lowest cost is selected as the new task allocation. Since the selection of the most appropriate task allocation is based on the costs of the agents, the desired optimization goal like the number of migrations, degradation of quality-of-service, or balanced use of resources can be achieved by adapting the scaling factors of the cost classes. After this process, the new task allocation is broadcast to all smart cameras and agents, which then update their QoS level, or migrate to another smart camera. To enable the fast reconfiguration by removing feasible task allocations, also the set of feasible task allocations is broadcast to all smart cameras in the surveillance cluster after the system has settled.

4.6 Hardware Setup

To verify, test and evaluate the presented agent system, we have used two hardware platforms. A prototype of our smart camera and PCs equipped with DSP boards.
The prototype of our smart camera consists of an *Intel IXDP425 Development Board*, which is equipped with an *Intel IXP425* network processor running at 533 MHz. The board is operated with Linux Kernel 2.6.8.1, which allows the usage of standard software packages. Image acquisition is done using the *National Semiconductor LM9618* monochrome CMOS image sensor, which is connected to one of the DSP boards. The prototype supports up to four DSP boards, however, our prototype is equipped with two *Network Video Development Kits (NVDK)* from ATEME. Each board is comprised of a TMS320C6416 DSP from Texas Instruments, running at 600 MHz, with a total of 264 MB of on-board memory.
Due to the lack of additional smart camera prototypes, we are using two PCs, which are equipped with one Network Video Development Kit.

5. EXPERIMENTS

In order to verify and evaluate the presented task allocation system, we created a surveillance cluster, comprised of the smart camera prototype and two PCs.

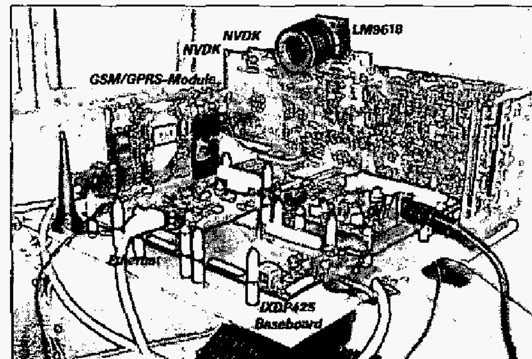

Figure 8: The prototype of the smart camera

We have used four different agent types for our experiments: (1) A MPEG-4 video compression agent, (2) a stationary-vehicle detection (StVD) agent, (3) a vehicle count agent, and (4) a vehicle classifier agent. The MPEG-4 agent and the stationary-vehicle detection agent (2) are well tested and evaluated agents, however, the vehicle count and vehicle classify agents only simulated real behavior. A total of six agents have been instantiated from these four classes, where we used three MPEG-agents (as every camera has to transmit a live video stream), and one agent of every other agent-type.

Table 1 enlists the execution times of the two stages of the task allocation. On the PCs we have used two Java virtual-machines: (a) Suns JDK 1.4.2, which uses a just-in-time (JIT) compiler, and therefore reaches better performance values, while (b) the JamVM virtual machine (see http://jamvm.sourceforge.net) is designed as an interpreter and therefore requires higher execution times. Lines 1a, 2a, and 3a represent the execution times if all six tasks had to be considered. As line 2a shows, that the merging process, resulting with a total of 2880 feasible task allocations, requires the most amount of time, therefore we have implemented the core calculation functions of steps 1 and 2 also in C++ to achieve acceptable performance. In a second scenario we have removed the MPEG-4 encoding agents from the surveillance cluster, and allocated them statically to the cameras. The results of the dynamic allocation of the resulting three agents are enlisted in lines 1b, 2b, and 3b in table 1.

Finally, we have also evaluated the performance of the task allocation system at increasing system load. Line 4 enlists the times required to check all 2880 allocations for feasibility. These results show, that the system responds in a timely manner to increased system load.

6. CONCLUSION

In this paper we have presented a resource-aware dynamic task-allocation system targeting embedded smart cameras. Surveillance tasks are not allocated to

Case	Executed by	PC JDK 1.4	PC JamVM	SmartCam JamVM	SmartCam Native/C++
1a	Partial Allocations (6 Ag.)	20 ms	14 ms	79 ms	13 ms
1b	Partial Allocations (3 Ag)	14 ms	7 ms	55 ms	9 ms
2a	Merge Solutions (6 Ag)	766 ms	4.852 ms	21.363 ms	2.360 ms
2b	Merge Solutions (3 Ag)	9 ms	5 ms	31 ms	4 ms
3a	Overall allocation determination (6 Ag)	786 ms	4.866 ms	21.442 ms	2.373 ms
3b	Overall allocation determination (3 Ag)	23 ms	12 ms	86 ms	13 ms
4	Pruning of infeasible allocations (6 Ag)	14 ms	32 ms	172 ms	26 ms

Table 1: Running times of task allocation

the smart cameras directly, but to groups of smart cameras, surveillance clusters, within the tasks are allocated by the smart cameras in a distributed manner. We show that the task allocation can be formulated as a distributed constraint satisfaction problem (DCSP) and present a solution for this DCSP. Therefore we combine the results of the DCSP with a cost function to retrieve the optimal allocation of tasks. Finally, we discuss the mobile-agent based implementation of the system and present results achieved by evaluation of the implemented system.

Future work includes (1) the further evaluation of the system, using more complex scenarios, (2) the tighter combination of finding feasible allocations and calculating the costs for an allocation in order to delete expensive allocations at an early stage, (3) the test and evaluation of the system in real-world scenarios, and (4) the integration of learning agents into surveillance clusters, which influence the behavior of the system by adapting the cost function based on previous behavior.

REFERENCES

1. B. Abreu, L. Botelho, A. Cavallaro, D. Douxchamps, T. Ebrahimi, P. Figueiredo, B. Macq, B. Mory, L. Nunes, J. Orri, M. J. Trigueiros, and A. Violante. 2000. Video-Based Multi-Agent Traffic Surveillance System. Proceedings of the IEEE Intelligent Vehicles Conference.

2. M. Bramberger, J. Brunner, B. Rinner, and H. Schwabach. 2004. Real-Time Video Analysis on an Embedded Smart Camera for Traffic Surveillance. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, 174–181

3. M. Bramberger, B. Rinner, and H. Schwabach. 2004. An Embedded Smart Camera on a Scalable Heterogeneous Multi-DSP System. Proceedings of the European DSP Education and Research Symposium (EDERS 2004).

4. K.-P. Chow and Y.-K. Kwok. 2002. On Load Balancing for Distributed Multiagent Computing. IEEE Transactions on Parallel and Distributed Systems, 13(8). 787–801.

5. T. Mullanix, D. Magdic, V. Wan, B. Lee, B. Cruickshank, A. Campbell, and Y. DeGraw. 2003. "Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System". Technical Report: SPRA795A, Texas Instruments.

6. M. K. Perdikeas, F. G. Chatzipapadopoulos, I. S. Venieris, and G. Marino. 1999. Mobile agent standards and available platforms. Elsevier Computer Networks, (31).

7. X. Qin, Q. Zhu, and D. Swanson. 2003. A Dynamic Load Balancing Scheme for I/O-Intensive Applications in Distributed Systems. Proceedinges of the IEEE Intl. Conference on Parallel Processing Workshops.

8. N. Suri, P. Groth, and J. Bradshaw. 2001. While You're Away: A System for Load-Balancing and Resource Sharing based on Mobile Agents. Proceedings of the IEEE/ACM Intl. Symposium on Cluster Computing and the Grid, 470–473.

9. W. Wolf, B. Ozer, and T. Lv. 2002. Smart Cameras as Embedded Systems. IEEE Computer, 35(9).48–53.

10. M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. 1998. The distributed constraint satisfaction problem: formalization and algorithms. IEEE Transactions on Knowledge and Data Engineering, 10(5).