Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 349465, 17 pages
doi:10.1155/2008/349465

Research Article

Using High-Level RTOS Models for HW/SW Embedded Architecture Exploration: Case Study on Mobile Robotic Vision

François Verdier, Benoît Miramond, Mickaël Maillard, Emmanuel Huck, and Thomas Lefebvre

ETIS Laboratory, CNRS UMR 8051/University of Cergy-Pontoise/ENSEA, 6 avenue du Ponceau, 95000 Cergy-Pontoise Cedex, France

Correspondence should be addressed to François Verdier, verdier@ensea.fr

Received 1 March 2008; Revised 19 June 2008; Accepted 22 July 2008

Recommended by Guy Gogniat

We are interested in the design of a system-on-chip implementing the vision system of a mobile robot. Following a biologically inspired approach, this vision architecture belongs to a larger sensorimotor loop. This regulation loop both creates and exploits dynamics properties to achieve a wide variety of target tracking and navigation objectives. Such a system is representative of numerous flexible and dynamic applications which are more and more encountered in embedded systems. In order to deal with all of the dynamic aspects of these applications, it appears necessary to embed a dedicated real-time operating system on the chip. The presence of this on-chip custom executive layer constitutes a major scientific obstacle in the traditional hardware and software design flows. Classical exploration and simulation tools are particularly inappropriate in this case. We detail in this paper the specific mechanisms necessary to build a high-level model of an embedded custom operating system able to manage such a real-time but flexible application. We also describe our executable RTOS model written in SystemC allowing an early simulation of our application on top of its specific scheduling layer. Based on this model, a methodology is discussed and results are given on the exploration and validation of a distributed platform adapted to this vision system.

1. INTRODUCTION

Today, real-time visual scene processing represents one of the major problem for autonomous robots. Lots of robot behaviours are based on this processing: navigation, object recognition and manipulation, target tracking, and even social interactions between human and robots. Currently, visual systems require large computing capabilities making them hard to embed. Indeed, most of such heavy vision tasks are often performed by distant host computers via network connections.

However, for several years, new approaches developed for visual processing have been proposed. The visual system is not considered isolated anymore but as part of an architecture integrated in its environment. They take into account more and more parameters related to the dynamic properties of the systems they belong to (see active vision [1]). These new visual processing algorithms strongly depend on the dynamics of interactions between the whole system and its environment by continuous feedbacks regulating even the low-level visual stages (see, e.g., the attentional mechanisms in biological systems).

The studied application consists in a subset of a cognitive system allowing a robot equipped with a charge-coupled device (CCD) camera to navigate and to perceive objects. The global architecture in which the visual system is integrated is biologically inspired and based on the interactions between the processing of the visual flow and the robot movements (Per-Ac architecture [2]). The learning of the sensorimotor associations allows the system to regulate its dynamics [3] and, therefore, navigate, recognise objects, or create a visual compass [4].

In this paper, we aim at designing an embedded visual processing system in the form of a single chip that could be used for building the CCD-based smart camera of our robot. On one hand, the embedded processing part should be flexible enough in order to allow a variety of navigation missions. It also needs to adapt to evolutive constraints due to the global system intrinsic dynamics (see Figure 1). On the other hand, the architecture should also provide intensive computation capabilities to deal with low-level

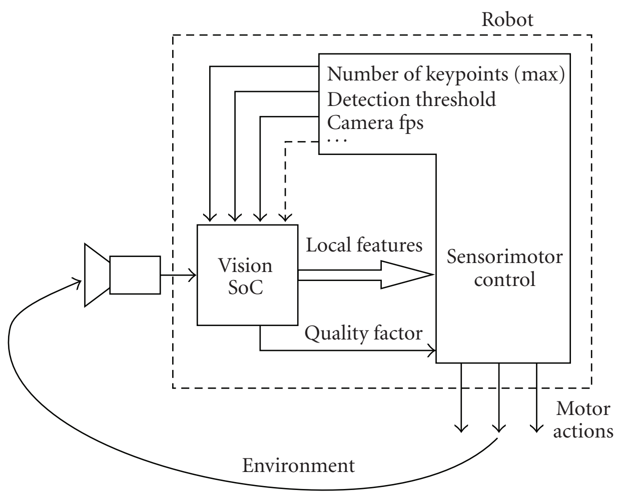

Figure 1: The dedicated SoC architecture used in a robot global sensorimotor loop.

image processing. One of the solutions to provide efficiency and flexibility may consist in implementing the application both in hardware and software. All regular and intensive computation tasks should be implemented in pipelined hardware modules and all irregular and control tasks should be mapped onto classical processing elements. These design choices implicitly lead to a heterogeneous and probably distributed application composed of multiple computation tasks. Such an application will be managed by a dedicated real-time operating system (RTOS). In our case, the use of an RTOS becomes essential in a domain where applications exhibit dynamic and adaptive behaviours.

1.1. Related works

When designers are faced with an SoC implementation of a new application, the hardware and software parts of the SoC must be designed according to the computing properties of the application. Precisely, our vision application has a specific dynamical real-time behaviour. As a result, we advocate for the use of high-level system models to early validate architecture alternatives and the corresponding real-time behaviours according to the constraints. Practically, the definition of a custom SoC architecture following a high-level design methodology is based on [5] the following:

(i) high-level modelling of the hardware and software components of the SoC,

(ii) exploration of the design space,

(iii) validation of the selected design solutions.

Some system level methodologies and tools already help to design SoC architectures composed of hardware and software processing elements. These methodologies are often based on system level design languages (SLDLs) such as SystemC [6] or SpecC [7] and are presented in the following. Unfortunately, only few of the proposed techniques include the definition of a dedicated RTOS into their design flow. Indeed, hardware/software codesign now includes the problem of the RTOS design which is considered as the main component responsible for the control of the global system.

For example, some works have proposed efficient solutions for the automatic synthesis of distributed implementations of dataflow-dominated applications under real-time constraints. For software-based applications, the SynDex tool [8] allows, for example, to describe an application as a hierarchical dataflow graph (and the corresponding code of elementary blocks) and to automatically generate a fully static software implementation on a heterogeneous multiprocessor architecture. With this kind of scheduling solution, the execution order of the elementary blocks must be defined at compile time. It thus cannot be used efficiently in application domains where potential parallelism can change dynamically according to input data (except if a worst-case architectural dimensioning is done). Unfortunately, visual computations such as the one found in autonomous robotics cannot always be predetermined. The degree of parallelism can vary during its execution. Static scheduling solutions are thus inappropriate in a real-time context and a dynamical scheduling must be defined online by a real-time operating system.

In the codesign context, many automatic HW/SW synthesis methods have been proposed [9, 10] but do not provide anymore dynamic behaviours management. Some codesign methods such as COSYN [11] deal with multitask application specifications but in the same static context. Realistic embedded systems design need methods to rapidly define RTOS in an application-specific way. This need has been identified by recent research works. First of all, in the context of high-level design methods, solutions have been proposed to model an RTOS at a high level. Gerstlauer et al. in [12] have recently initiated this research activity by presenting an RTOS model on top of the SpecC SLDL. As SystemC or SpecC SLDLs allow timed simulations of written models, the work in [12] takes advantage of SpecC primitives to explicitly model dynamic behaviour of multitasks systems at high levels of abstraction.

SoCoS in [13] is a C++ library for system-level design providing the user automatic linking with operating system (OS) services. The main difference with [12] is that SoCoS requires its own proprietary simulation engine.

In [14], Le Moigne et al. describe a SystemC model of a multitask RTOS. This model is a part of the Cofluent tool [15] which allows timing parametrisation and validation of the RTOS model by configuring context load, context save, and scheduling duration.

After modelling and simulating high-level RTOS representation, another problem addressed by Gauthier et al. is the automatic generation of RTOS code. In [16], they present a method of automatic generation of operating systems for a target processor. This method finds OS services required in the code of the application SW and generates the corresponding code deduced from dependencies between services in an OS service library.

Putting aside the works of [12], none of the existing RTOS modelling approaches deals with creation of dynamic processes. However, as it will be explained in this paper, this property is needed to early validate the real-time behaviour of our application. Hence, our method addresses this design challenge by introducing a high-level RTOS model for custom SoC design. Working at a high level of abstraction allows the designer to jointly explore the RTOS architecture in terms of custom services adapted to the application and the parallel SoC architecture. Both dynamic behaviour control and embedded constraints satisfaction problems can thus be solved by a single approach.

Contributions of this work consist in proposing a SystemC functional accurate dynamical RTOS model allowing a high-level simulation of a distributed architecture. This simulation is done at a *Service Accurate* level in the sense that allows functional and timed verification without the need of modelling explicitly processing resources. By working at this level of abstraction, an early exploration of the architecture dimensioning and the validation of application real-time constraints are feasible.

1.2. Paper organisation

The rest of the paper is organised as follows. Section 2 presents our robotic vision application in more details and stresses its dynamical properties. Section 3 describes the proposed RTOS modelling approach based on the system-level design language SystemC. Results are also given on the corresponding dynamical implementation of the vision application.

We then discuss in Section 4 how our simple RTOS model can be used for building a parallel and distributed multiprocessor architecture coupled with dedicated hardware accelerators. In Section 5, we give the results of the proposed Hw/Sw exploration process permitted to define the multiprocessor system-on-chip (MPSoC) platform dedicated to our vision application. Finally, we conclude and discuss some perspectives in Section 6.

2. A VISION APPLICATION FOR ROBOT PERCEPTION AND NAVIGATION

In the following, we first describe the considered vision application. This application mainly consists in applying classical filtering, subsampling, and extraction operators, and it can be considered as a pure data flow process. However, we will detail in Section 2.2 how this application is inserted in a global dynamic and adaptive regulation loop (see Figure 1). We will then show why that forbids the use of classical implementation flows in this specific context.

2.1. Application description

The current visual system here is close to the one used by Leprêtre et al. in [17] and integrate a multiscale approach to extract the visual primitives. In doing so, it also allows a wider range of applications. Roughly the visual system provides a local characterisation of the keypoints detected on the image flow of an 8-bit gray-scale CCD camera (382×286 pixels). This local characterisation feeds a neural network which can associate motor actions with visual information: this neural network can learn, for example, the direction of a displacement of the robot as a function of the scene recognition. The studied visual system can be divided into two main modules:

(i) a multiscale mechanism for characteristic points extraction (keypoints detection),

(ii) a mechanism supplying a local feature of each keypoint.

2.1.1. The multiscale keypoints detection

The multiresolution approach is now well known in the vision community. A wide variety of keypoints detectors based on multiresolution mechanisms can be found in the literature. Amongst them are the Lindeberg interest point detector [18], the Lowe detector [19]—based on local maxima of the image filtered by difference of Gaussians (DoGs)—or the Mikolajczyk detector [20], where keypoints correspond to those provided by the computation of a 2D Harris function and fit local maxima of the Laplacian over scales.

The visual system described here is psychophysically inspired in the sense that it takes into account the work done on the Müller-Lyer illusions in [21]. The used detector extracts points in the neighbourhood of the keypoints, which are sharp corners of the robot's visual environment. More precisely, the keypoints correspond to the local maxima of the gradient magnitude image filtered by DoGs (Figure 2). Moreover, the detector remains computationally reasonable and is characterised by a good stability. It also automatically sorts the keypoint lists. The gradient magnitude Grad is computed by the following equation (where $I(x, y)$ corresponds to the pixel magnitude at coordinates (x, y)):

$$\operatorname{Grad}(x, y) = \sqrt{\frac{\left(I(x+1, y)-I(x-1, y)\right)^2+\left(I(x, y+1)-I(x, y-1)\right)^2}{2}}. \quad (1)$$

Keypoints are detected in a sampled scale space based on an image pyramid. Pyramids are used in multiresolution methods to avoid expensive computations due to filtering operations. The algorithm used to construct the pyramid is detailed and evaluated in [22]. The pyramid is based on successive image filtering with 2D Gaussian kernels ($G_\sigma(x, y)$) normalised by a factor S:

$$G_\sigma(x, y) = \frac{e^{(-(x^2+y^2)/2\sigma^2)}}{S}. \quad (2)$$

These operations achieve successive smoothing of the images. Two successive smoothing are carried out by two Gaussian kernels with variance $\sigma^2 = 1$ and $\sigma^2 = 2$. The scale factor doubles (achievement of an octave) and thus the image is decimated by a factor of two without loss of information. The same Gaussian kernels can be reused to continue the pyramid construction. Interestingly, the kernel sizes remain small (half-width and half-height of $3 \times \sigma$) allowing a fast

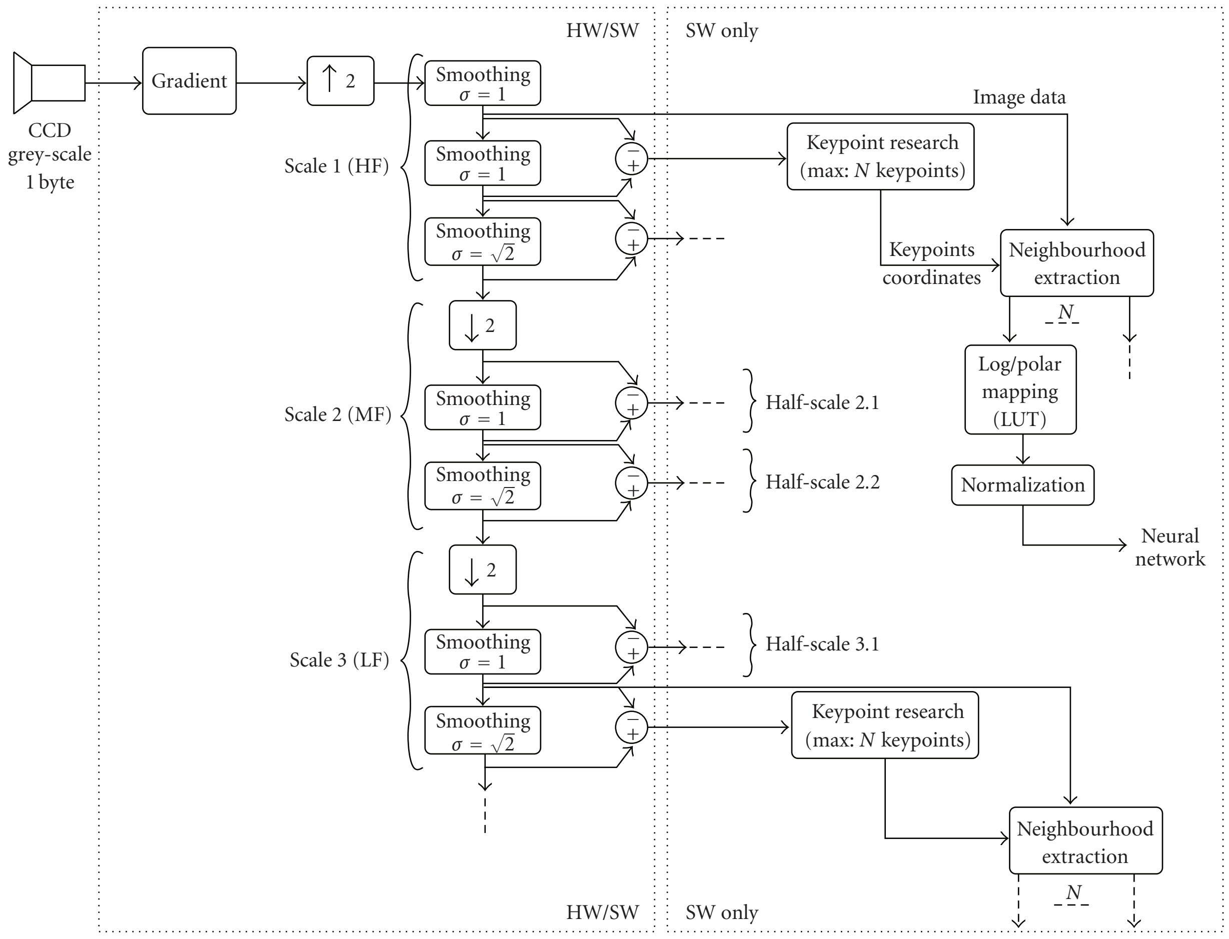

Figure 2: Global architecture of the algorithm. Local features are extracted from the neighbourhood of characteristic points detected on each image of the pyramid.

computation of the pyramid. Finally, the images filtered by DoGs in the pyramid can be simply obtained by subtracting two consecutive images.

Keypoints detected on the images are the first N local maxima existing in each DoG image of the pyramid. Thus, the keypoint research algorithm orders the N first local maxima according to their intensities and extracts their coordinates. The shape of the neighbourhood for the research of maxima is a circular region with a radius of 20 pixels.

The number N which parametrises the algorithm corresponds to a maximal number of detections. Indeed, the robot may explore various visual environments (indoor versus outdoor) and particularly more or less cluttered scenes may be captured (e.g., walls with no salience versus complex objects as illustrated in Figure 3). A detection threshold (γ) is set to avoid nonsalient keypoints (Figure 3 illustrates the effect of this parameter on different images). This threshold is based on a minimal value of the local maxima detected. The presence of this threshold is even more important in the lowest resolutions since the information is very coarse at these resolutions. This particularity of the algorithm confers it a dynamical aspect. Precisely, the number of keypoints (and consequently the number of local features) depends on the visual scene and is not known a priori. Furthermore, the threshold γ could be set dynamically through a context recognition feedback but discussing here this mechanism is not our purpose (see an example of context recognition in [23]). However, even if this threshold is considered as a constant value, the number of detected keypoints varies dynamically according to the input visual scene. Consequently, the number of computations (neighbourhood extractions) also depends on the input data.

2.1.2. *The local image feature extraction*

At this stage, the neighbourhood of each keypoint has to be characterised in order to be learnt by the neural network. Existing approaches to locally characterise keypoints are numerous in the literature: local jets, scale invariant feature transform (SIFT) and its variants, steerable filters, and so forth (see [24] for a review of local descriptors).

In the current application, we simply reuse a view-based characterisation where keypoint neighbourhoods are represented in a log-polar space. This representation has good

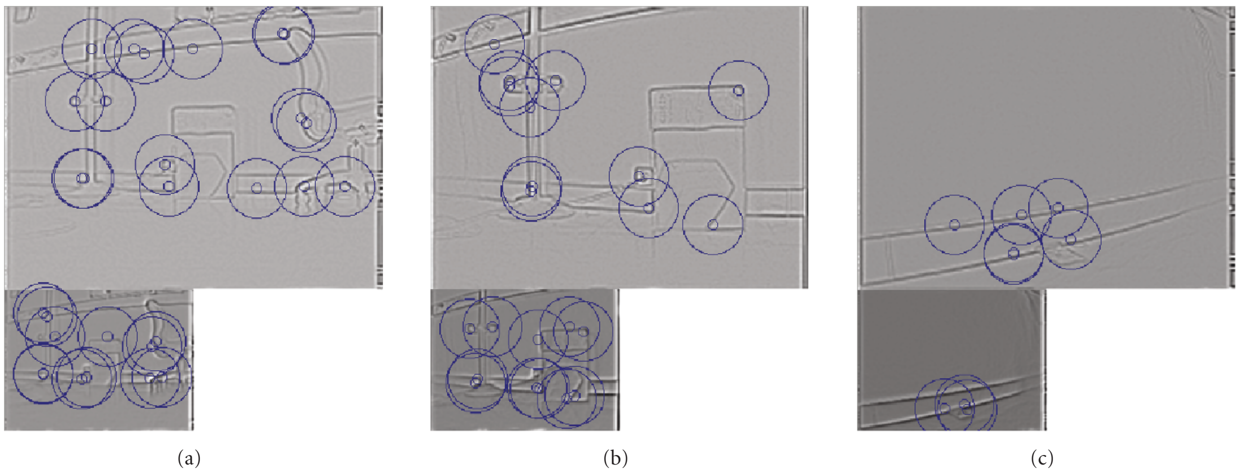

Figure 3: Keypoints detected on different visual viewpoints. The same detection threshold is used in the three cases. Keypoints coming from the same octave are gathered on one image (2 octaves represented): (a) cluttered scene (29 keypoint detected); (b) same scene but closer, less cluttered (22 keypoints); (c) view captured during a wall following (9 keypoints).

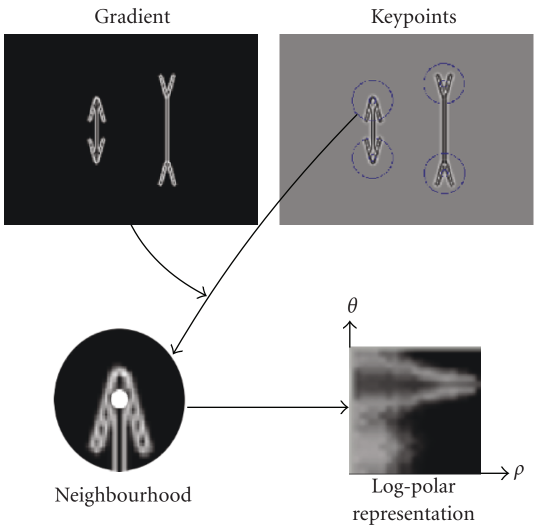

Figure 4: Local features extracted at one scale on Müller-Lyer's illusion.

properties in terms of scale and rotation robustness [25]. This kind of mapping is also used by the *GLOH* descriptor [24]. The local feature of each keypoint is, therefore, a small image resulting from the log-polar transformation of the neighbourhood (Figure 4). This transformation is done by a lookup table (LUT) allowing the mapping: $(\rho, \theta) = f(x, y)$.

The neighbourhoods are extracted from the gradient magnitude image at the scale the keypoint was found by the detector. Each neighbourhood extracted is a ring of radius $(5, 36)$ pixels. Excluding the small interior disc avoids multiple representations of the central pixels in the log-polar coordinates.

The angular and logarithmic radius scale of the log-polar mapping are sampled with 20 values. Each feature is thus an image of dimension 20×20 pixels. The sizes of the rings and feature images have been determined experimentally for an indoor object recognition. The given parameters represent a tradeoff between stability and specificity of the features [26]. Finally, the small log-polar images are normalised before their use by the rest of the neural architecture. By associating the data provided by the visual system with actions, the global system allows the robot to behave coherently in its environment [3].

Generally, the visual system must not be considered isolated but integrated in a whole architecture whose modules interact dynamically with each other and through the environment [27]. Hence, the evaluation of the parameters of the visual system depends on the rest of the robot system architecture.

2.2. A custom RTOS as a solution for domain-specific implementation

When integrated in mobile robots with navigation or object recognition objectives, this application must obviously satisfy some real-time constraints. For example, the local image features extracted in the neighbourhood of keypoints may be used for obstacle perception and, therefore, for trajectory guidance. In the general case, this vision subsystem must match real-time constraints if it is used in a global sensorimotor loop. Robot and environment integrities depend on these constraints.

However, due to the complex dynamic behaviour of our vision application, a precise characterisation of its timing behaviour is not trivial a priori. Expressing a global application deadline or period matching all context conditions (robot motivation and internal state, nature of the visual scenes, etc.) is impossible at compile time and mainly depends on the global system dynamics (see Figure 1). In our use case, the period constraint (the camera fps) is a parameter, amongst other regulation signals (maximum number of extracted keypoints N, detection threshold γ), given by the external control system. All of these parameters will dynamically vary during the system lifetime.

More precisely, let us suppose the rough functional partitioning of our application into separate tasks as given in Figure 2 (at this step we do not consider the Hw/Sw partitioning). In this case, the number of concurrent computation subtasks (number of extracted keypoints thus the parallelism degree) and their deadlines have different configurations according to the current system mode. In these conditions, all classical implementation tools are clearly inappropriate for the design of our SoC architecture. Since deadlines are variable, all static scheduling-based compilers or synthesisers will be inapplicable except if the number of processing resources corresponds to the maximum degree of parallelism. The latter is of course unthinkable in embedded systems.

As an example, we have done a temporal profiling of the vision application and measured the execution time of the application tasks. This experiment has been done with a pure software version of the application executed on a single Nios2-embedded 32 bits microprocessor from Altera with a 100 Mhz clock frequency. Application tasks can be divided in three groups according to the execution time:

(i) intensive data-flow computation tasks that execute in a constant time,

(ii) tasks with execution time correlated with the number of interest points (Figure 5),

(iii) tasks with unpredictable execution time (Figure 6).

These preliminary results obviously show that intensive computation tasks would interestingly be implemented as pure (eventually pipelined) hardware modules. On the other hand, execution results of the search, sort, and extraction tasks confirm a software implementation poorly adapted to a static scheduling.

It is well known that multiscale systems take advantage of distinguishing the visual processing done at the different scales. In the classical but limited *coarse-to-fine* approach, lowest resolutions are considered with highest priorities. Visual data from these resolutions are integrated quickly in the system to get a first coarse description of the environment. This description is then refined by other scales. More reasonably, the visual system behaves in a more flexible way but remains still based on these different priorities. It may favour the utilisation of different frequency domains according to its objectives. For example, the recognition of facial human expressions can be done *selectively* in different frequency domains (see [28] for a more detailed discussion on frequency selection). In a navigation task, this approach can be extended: in noncluttered environments, a priority to the low scales can be given to navigate coarsely. The robot speed and/or the camera acquisition speed (the frame per second) can be increased. At the opposite, to precisely recognise objects, keypoints from other scales must also be taken into account (more scales would have high priorities) and the robot and camera speeds must be reduced accordingly.

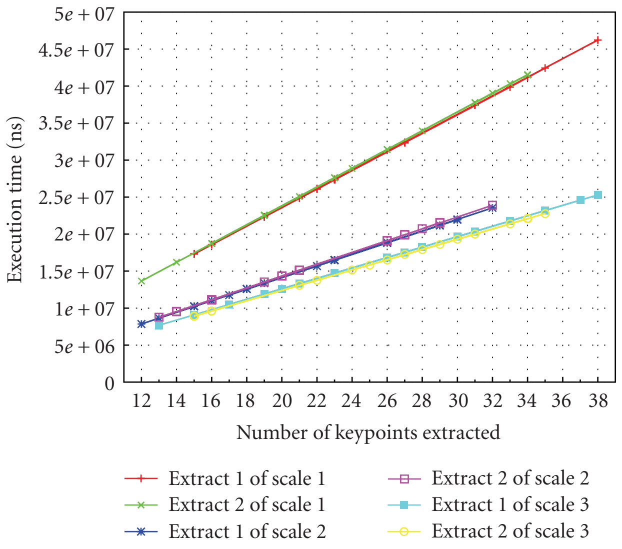

FIGURE 5: Tasks with execution time linearly dependent with the number of keypoints processed on representative samples.

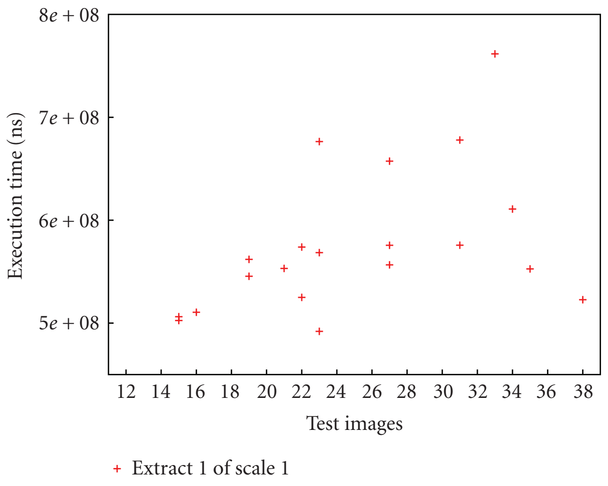

FIGURE 6: Tasks with unpredictable execution time on representative samples.

Yet designing a hard real-time system imposes the static knowledge on upper bounds of application parameters. So our application, with its particular properties, seems not adapted to execute in real time with deterministic and predictive times. For that reason, we have defined three modes (ranges of parameters and constraints) based on the selective coarse-to-fine approach, which correspond to the three main types of behaviour for the robot.

(i) Fast mode: the robot moves around in a learnt environment, it only needs coarse description of the landscape but at a high frame rate. Thus, only the lower scales are processed but the keypoints of the current image must be provided to the neural network (sensorimotor control in Figure 1) at the rate corresponding to the robot speed.

Table 1: Application modes.

Mode	N	FPS	deadline	mission
Fast	10	20	50 ms	Obstacle detection in known environment
Intermediate	30	5	200 ms	Exploration of unknown environment
High detail	120	1	1s	Static detailed analysis of environment

It is the reason why in this first mode we fix the maximum number of extracted keypoint $N = 10$, and the number of frames processed per second (fps) to 20.

(ii) Intermediate mode: the robot moves slowly, for example, when the passage is blocked (obstacle avoidance, door passage) and needs more precision on its environment. In this mode, the middle and the lower scales are processed. So, keypoints with information on a larger frequency band are provided to the neural network.

In the intermediate mode, we fix $N = 20$ and the system works at a rate of 5 fps.

(iii) High-detail mode: the robot is stopped in a recognition phase (object tracking, new place exploration). All scales are fully processed and full information on the visual environment is provided at the expense of the processing time.

For this last mode, we fix $N = 120$ and the rate of the system to 1 fps.

For all of these modes, summarised in Table 1, we consider a constant value for the detection threshold $\gamma = 200$. The system parameters, such as the number of frames per second processed by the system (and consequently the period), have been defined from measures on a representative sample of images (acquired by the robot) on the embedded softcore processor we use to determine realistic times (the Nios II introduced previously). Moreover, the upper, average, and lower bounds of execution times for those three modes have been analysed in order to extract predictive behaviours from the global dynamics. Exactly this approach limits the system dynamics so that in each mode the system would now be under hard real-time constraints. It also allows to explore and define the dimensioning of the embedded architecture.

Inside each mode, and from a task-scheduling point of view, these mechanisms can be modelled by attributing different priorities to the local features extraction and normalisation tasks. Moreover, in some cases. the lowest priority tasks may not be executed without significantly damaging the robot behaviour. According to the external conditions, scheduling or not the lowest priority tasks must be integrated in a quality-of-service (QoS) information. This information will be necessary to insure an efficient global sensorimotor loop. Such a *quality factor* would easily be computed locally with the number of extracted features versus the total number of keypoints ratio. This execution scenario can only be achieved by an embedded dynamical real-time operating system.

So far, we are faced with a twofold problem. First, to integrate the development of an embedded RTOS in an HW/SW design flow. Second, to propose a design methodology for exploring application-specific dynamical scheduling strategies. Such a SoC design approach requires a methodical design process based on the concept of high-level modelling which allows designers to explore and validate application deployment alternatives on a dedicated architecture. It permits also to incrementally refine this model towards the final hardware and software implementation. Today, developing a high-level RTOS model constitutes a challenge and exploring dedicated dynamical scheduling policies is not addressed by existing approaches such as [11, 14, 16].

Since version 2.1 of the SLDL SystemC simulation engine, we developed the basic mechanisms for building such a model of an embedded RTOS. Those mechanisms will be essential for the development of our future SoC platform. From these basic blocks, we developed a modular SystemC RTOS model. It provides the RTOS services responsible for the dynamic control of the execution sequencing and the possibility to dynamically create applicative processes. The high-level model of this is described in the Section 3. Section 4 will bring the first guidelines for using our model as a basis for the exploration and validation of a dedicated distributed embedded platform needed by our application.

3. BUILDING THE SYSTEMC MODEL OF A REAL-TIME OS

We present in this section how it is possible to define a SystemC model of a relatively simple but realistic real-time operating system layer. We will particularly focus on the dynamical mechanisms of the RTOS (i.e., the dynamical creation and preemption of processes) needed by our vision application.

For the sake of illustration, Figure 7(a) gives a subset of the vision application, partitioned into several tasks, that will be used to highlight the mechanisms we propose. `Sampling`, `Smoothing` and `DoG` tasks correspond to the subsampling, Gaussian filtering, and difference of Gaussian operations. `Search` task searches, sorts, and extracts the coordinates of the first keypoints above the detection threshold. It takes as data input the result of `DoG` and its associated `Smoothing`, and its parameters are N and γ. `Extract` task builds the corresponding neighbourhoods and `Norm` task transforms and normalises the local features. The exact way the application is partitioned into tasks is out of the scope of this work. Our purpose here is only to illustrate how we can model and simulate the execution of a multitask application on top of an embedded real-time OS.

In order to obtain the attended dynamic behaviour, our application needs to create, depending on some parameters or data, new treatments with different parallelism degrees. This behaviour cannot be estimated during development nor at compile time. The SystemC simulation kernel can neither

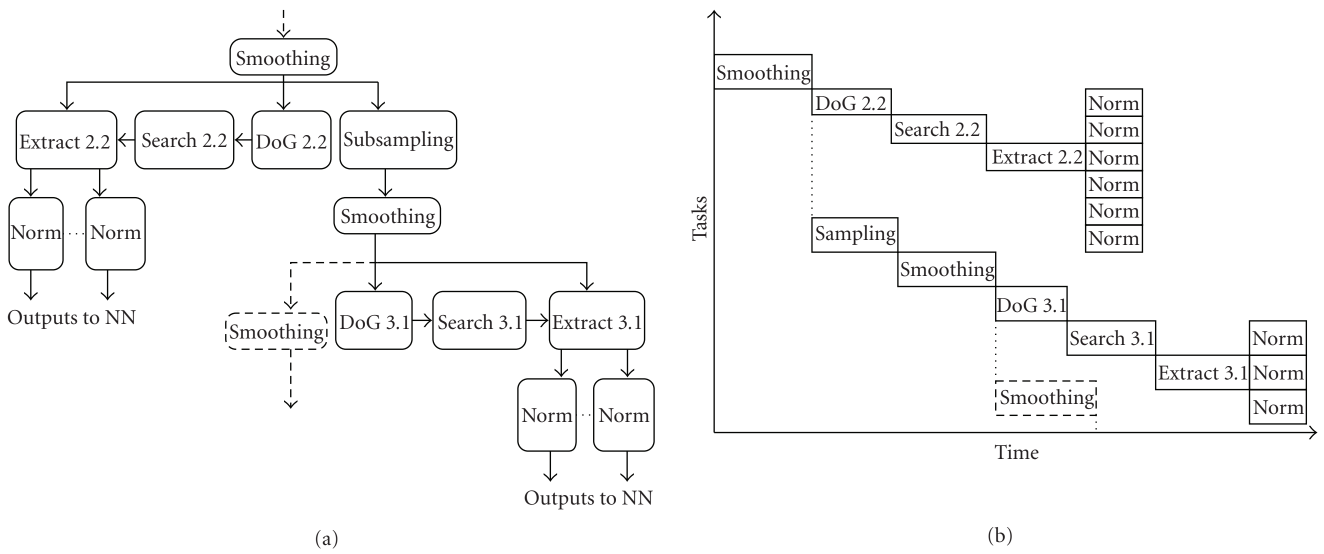

Figure 7: Example of the vision application partitioned into a task graph in (a). Only two half scales belonging to two different scales are illustrated. If an infinite number of resources are available and an ASAP (as soon as possible) scheduling is used, this graph would be scheduled as illustrated in (b) where the two half-scales contain, respectively, 6 and 3 keypoints.

handle such an execution model. Indeed, standard SystemC processes must be declared before starting a simulation (at elaboration time) and cannot vary until the end of simulation. This constitutes the first limitation of modelling dynamical-embedded software as classical processes in SystemC 2.0.

The second problem of modelling an OS in SystemC consists in finding a mechanism allowing an explicit scheduling of applicative tasks instead of the native event-based SystemC scheduler. The native SystemC simulation kernel acts as an abstraction layer and provides to the developer a model of a pure virtual architecture with an infinite number of resources. It thus simulates the execution of all processes in a parallel way (see the example of Figure 7(b)). If an architecture with a finite number of resources (processors) is targeted, the exact behaviour of an application composed of multiple tasks sharing the same resource cannot easily be simulated.

As a third keypoint, scheduling a task in an RTOS not only consists in starting or killing it but also, if preemptive schedulers are targeted, in interrupting tasks and switching between contexts. SystemC does not allow interrupting a thread without an explicit declaration of breakpoints by using the SystemC `wait()` primitive. Unfortunately, the `wait()` call cannot take into account several preemption and resume cycles. Thus a specific mechanism must be developed for modelling such an execution scheme.

3.1. Modelling the bare mechanisms

Dynamical creation of tasks—creation of SystemC processes—is now possible by using the new SystemC kernel version 2.1 that comes with the public `boost` library. This library offers primitives for spawning processes and attaching them to SystemC modules after the elaboration phase, thus dynamically during simulation. Moreover, thanks to the dynamic lists of sensitivity, SystemC threads can be started and resumed depending on dynamically created events.

3.1.1. Task creation service

As shown in Figure 7(b), the potential parallelism of the normalisation tasks (`Norm`) depends on the dynamical threshold γ set by an external context recognition feedback (see also Figure 1). The `Extract` task thus needs to dynamically create a variable number of `Norm` subtasks depending on the results of `Search` task.

We have implemented the `create_task()` service of the RTOS with the `sc_spawn()` primitive of the kernel. The `create_task()` service allows an applicative task to dynamically create several instances of new tasks. Our RTOS model implements tasks as C++ classes containing several data as member variables: priority, period, deadline, function code, and so forth. It is important to note that our task objects are not modelled as SystemC modules. Instead, each task is given as a global function that comes in a separate C++ file linked with our RTOS model. `create_task()` creates an instance of a new thread attached to the RTOS model. It creates also a new event that will be used by the scheduler for starting and resuming the thread. Finally, it creates the corresponding C++ structure containing all task data (e.g., task control block) for an easy manipulation by the RTOS.

By specifying tasks as functions instead of SystemC modules, application engineers do not need to take care about detailed implementation of the model. Moreover, as it will be mentioned in Section 4, the same application code could be used in a refined architectural model without any modification of the source code. For example, it could be linked with a more complicated platform model having

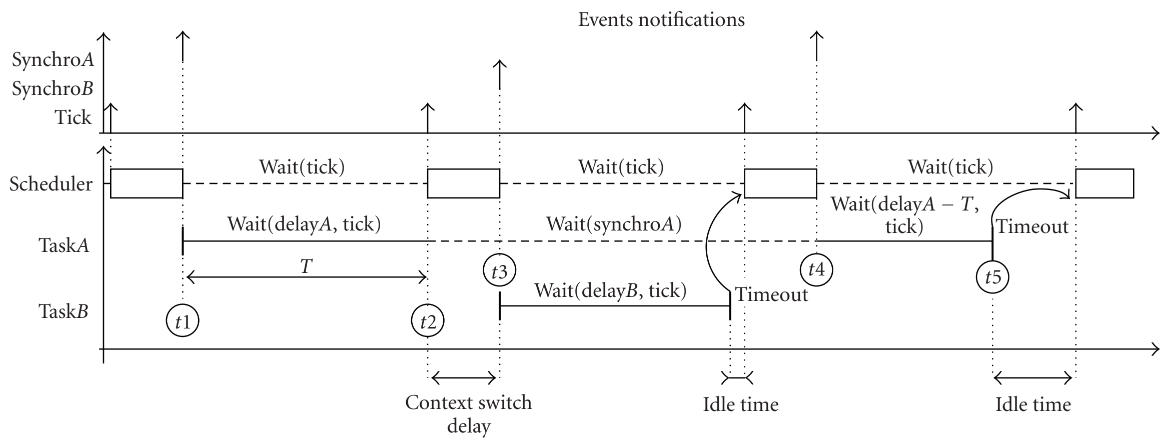

FIGURE 8: Details of the preemption mechanism provided by the os_wait() service.

several RTOS instances, thus modelling a multiprocessor architecture.

3.1.2. Scheduling

The main module of our RTOS model is the scheduler function. This function is developed as a member method of the scheduling module inside the RTOS model. Thanks to the object-oriented SystemC kernel, modifications of the scheduler can be done by using inheritance. Method overload is a light way for exploring different scheduling strategies. Even without the preemption mechanism, the model can help the developers to set the right task priorities and explore the scheduling algorithm. However, with the execution time modelling, a refined preemptive scheduling could be simulated and could give results on the OS overhead (number of context switches).

3.1.3. Tasks preemption

For modelling task interruption and preemption, we have created a dedicated `os_wait()` service that gives control to the OS model and allows taking into account preemption and context switches. Tasks are viewed as sequence of functional code and system calls. So their code is splitted into unbreakable portions of code each having their own estimated execution time, mentioned by using the `os_wait()` primitive. The `os_wait()` primitive has been written as a blocking call that returns after a specific time delay and allow to model preemption. This delay is the estimated task duration added dynamically by the sum of all interrupt treatments or scheduler invocations durations suspending the task. In this manner, `os_wait()` service acts as a way to model exact execution time and concurrency of tasks.

As an illustration, Figure 8 gives details about how preemption is modelled on a single processor architecture. The call to the SystemC `wait` primitive is encapsulated into the `os_wait()` service. In this example, when task*A* is launched, it first runs its functional code portion in zero simulation time, then it simulates its execution time by the `os_wait()` call instead of the classical `wait()` with its duration given in parameter (delay*A*). In fact, the corresponding service code waits for both the duration timeout and any interruption event like the real-time clock tick which announce the wakeup of the scheduling service. As the execution time of Task*A* is greater than the clock period, the elapsed time T ($t2 - t1$ on Figure 8) is taken into account and subtracted to the attended execution time.

At date *t2*, the scheduler is executed and launches Task*B* which has the same priority, following the round-robin policy. After the simulated context switching duration ($t3 - t2$), the synchronisation event of Task*B* is notified and the schedule process waits until a new clock tick. At time, *t3* Task*B* executes its functional code till the first encountered `os_wait()` call. This functional code is executed in zero time in the SystemC simulation engine. At its turn, Task*B* calls the `os_wait()` service for a duration lower than the tick period and terminates. At date *t4*, Task*A* is scheduled again and completes its execution time simulation (*t5*).

With this preemption model, the number of context switches between SystemC threads is approximately equal to the one in a real-preemptive RTOS kernel [29]. This leads to an efficient simulation with no significant overhead (see Section 3.3).

3.2. Building modular RTOS models

Thanks to the object-oriented nature of the SystemC library, the RTOS model can be developed in a modular way [29]. Adding new or specific services and modules can be easily done by using objects aggregation or inheritance and objects relationships. The RTOS interface (its application programming interface) works exactly the same way and can be updated and augmented according to the application needs.

Our RTOS model is implemented as a single hierarchical SystemC module and instantiates multiple service modules (see Figure 9). Each service module is also a hierarchical SystemC channel (is a SystemC module that inherits from SystemC interfaces). The application programming interface of a given service module is a part of the global RTOS API. This global API provided to the application is constructed

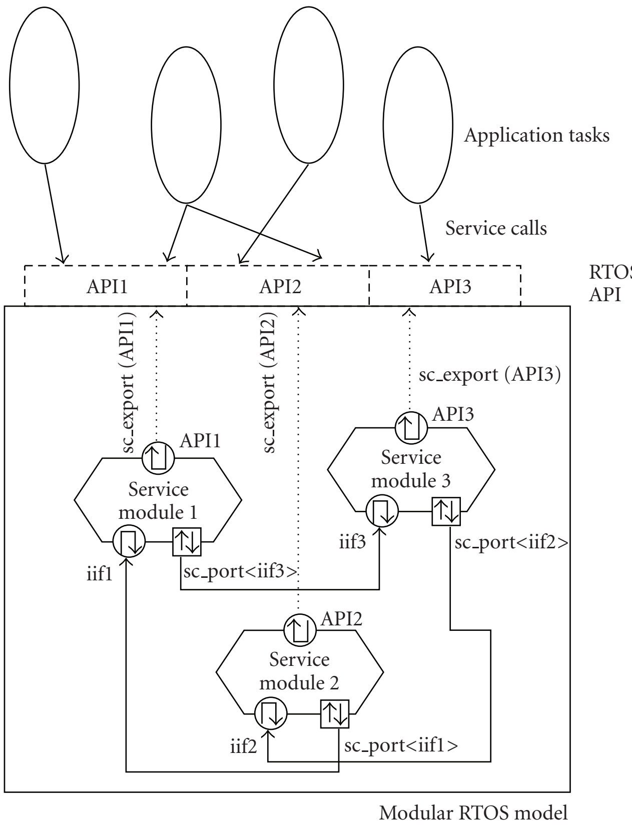

FIGURE 9: The modular RTOS model is a hierarchical SystemC module containing multiple interconnected service modules. Each service module is a SystemC channel requiring and implementing some dedicated interfaces.

in a modular way in the sense that it is the union of the all API interfaces instantiated by service modules. This can be done automatically at design time thanks to the `sc_export()` facility.

In addition, each service module of an RTOS model can provide an internal interface through which other service modules can interact. Respectively, each service module can have an internal communication port (`sc_port⟨⟩`) requiring the internal interface of another service module. Communication between modules are for the moment modelled as simple method invocations. An example of such a communication is the internal `change_task_state()` service call offered by the task management module: it allows other modules (as for the scheduling module for example) to change the internal state of an active task (see Algorithm 1 for the SystemC skeleton).

In order to propose a generic RTOS structure, we have defined several service categories by taking into account the potential locality of shared internal data as an important criterion. It is important to notice that minimising the service modules granularity—by building elementary modules—could enhance the exploration capability of the OS model at the expense of more complex communication infrastructure between modules. Our nonexhaustive service module list is then mainly composed of the following services: task management, interrupt requests management, semaphore service, scheduling, timer, and real-time clock management.

```
// Task management service API
class task_mgr_if: virtual public sc_interface {
public:
          virtual task∗ create_task (· · · ) = 0;
          virtual void kill_task (· · · ) = 0;
          virtual int get_pid (· · · ) = 0;
          · · ·
};
// Task management service internal interface
class task_mgr_iif: virtual public sc_interface {
public:
          virtual void change_task_state (· · · ) = 0;
          · · ·
};
// Task management service module header
class task_mgr: public sc_channel,
                    public task_mgr_if,
                    public task_mgr_iif {
private:
          · · ·
}; // End class taskmgr


// RTOS SystemC module
class my_OS: public sc_module{
public:
          // export all services API
          sc_export⟨task_mgr_if⟩ TMGR_IF;
          · · ·
SC_HAS_PROCESS (my_OS);
· · ·
// instanciates all service modules
task_mgr        my_task_mgr;
scheduler       my_sheduler;
· · ·
// inter_module binding
my_scheduler·task_mgr_port (my_task_mgr);
· · ·
// modules/exports binding is done in
// the constructor
· · ·
}; // End class my_OS
```

ALGORITHM 1: Example of the modular construction of the RTOS module. This example shows how a service module is declared with both a global interface exported to the application and an internal interface offered by this module to other service modules. The whole API is built by exporting each service's interface.

3.3. *Validation of the abstract OS*

Our RTOS model has been written in SystemC and was firstly validated under a single processor assumption with all of the presented services and RTOS modules. Two schedulers with different strategies have been tested in the model (fixed priorities without preemption and real-time round-robin).

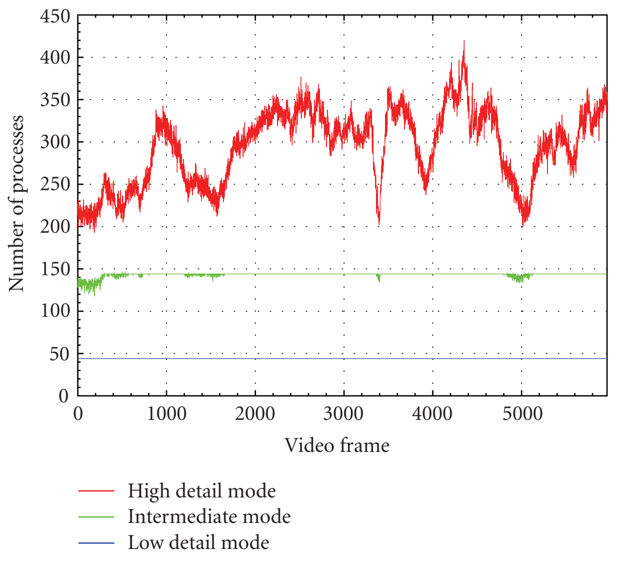

FIGURE 10: Number of process created and managed by the OS model during the application simulation.

The vision application source code has also been validated on top of the RTOS model. One can notice that porting this source code on the RTOS API did not represent a significant effort. The simulation of our application with the RTOS layer produced interesting results: a graphical C++ library (`gtk` + 2.0) has been included in the SystemC code and allowed us a quick functional verification (Figure 11). In addition, the model produces execution traces that allows a deep examination of the application behaviour. For example, simulation results presented in Figure 10 illustrate the dynamic behaviour of the RTOS model. Figure 10 shows the evolution of the number of active process in the different modes according to the complexity of the visual environment. In high-detail mode, all scales are processed and the number of process depends on the number of keypoints in the video frames. In this example, the OS dynamically creates, schedules, and deletes up to 420 processes. In the intermediate and fast modes, the number of process is, respectively, bounded to 30 and 10 processes by half-scale.

Moreover, a timed simulation is possible with this model. Indeed, timing data can be inserted in the SystemC code. Execution time estimations can be added in the RTOS model by using the dedicated `wait()` function inside the each RTOS module for modelling durations of system calls. By doing so, it becomes possible to simulate the timing overhead produced by the RTOS calls and scheduling operations. In addition, a global performance evaluation is also possible by giving worst-case execution time (WCET) estimations of each code portion as parameters to the `os_wait()` pseudoservice calls in the application code.

We evaluated the accuracy of our modelling approach by performing several sets of experiments and comparing the simulated execution times relatively to actual board measurements for multiple sets of data. The average application times are depicted in Table 2.

According to these results the high-level simulation, accuracy is within 3-4% of board measurements. This accuracy is acceptable at this level of description where the processing elements are abstracted. In addition, the exact task ordering and preemption realised by μC/OS-II on the board is modelled [29] thanks to the preemption modelling described precedently.

We have also compared the simulation duration of our application with or without the RTOS model. The simulation duration is about 2 minutes 53 seconds for the application described as pure functional C code (using Linux host API) and 3 minutes 12 seconds when the application runs over 4 SystemC RTOS models. These results have been obtained with a simulation host machine equipped with an Intel Dual-core at 1.66 GHz and on a set of 1000 images.

These results demonstrate that an annotated OS-based exploration methodology responds to the tradeoff between simulation time and estimation accuracy at early design steps in order to model and explore concurrency without taking into account specificities of the processing elements architecture. These properties will help us to explore the architecture dimensioning in Section 5.

4. TOWARDS A GLOBAL PLATFORM EXPLORATION AND DESIGN FLOW

In order to finalise the SoC platform, we must follow a dedicated methodology that includes the RTOS exploration and validation. We thus propose a global exploration and design flow based on successive refinement steps of both the architecture and the RTOS implementation strategies. This flow is partially inspired by the works of Gajski et al. and Jerraya et al. and is illustrated in Figure 12. A three-step modelling approach (specification, architecture, and implementation levels) has been proposed in [12]. The main drawback of this proposal is that there still exists a design gap between the architectural model (the one exhibiting the OS model) and the final implementation model (with processors modelled as instruction set simulators (ISSs)). For bridging this gap, we propose an intermediate distributed architecture model where the parallelism can be explored. The presented high-level RTOS model is used both in the second and third steps of this flow.

4.1. The SoC modelling flow

The proposed methodology then consists in four successive modelling levels (specification, executive, distribution, and implementation models) described in the following.

4.1.1. Specification model

This model is written as a pure SystemC model of the application where each applicative task is a SystemC thread or process and where intertask communications take place through the SystemC primitive channel (`sc_mutex`, `sc_fifo`, etc.). Tasks are synchronised with `sc_events`. The whole application is specified as a single `sc_module` with all tasks as member functions. A complete functional verification of the

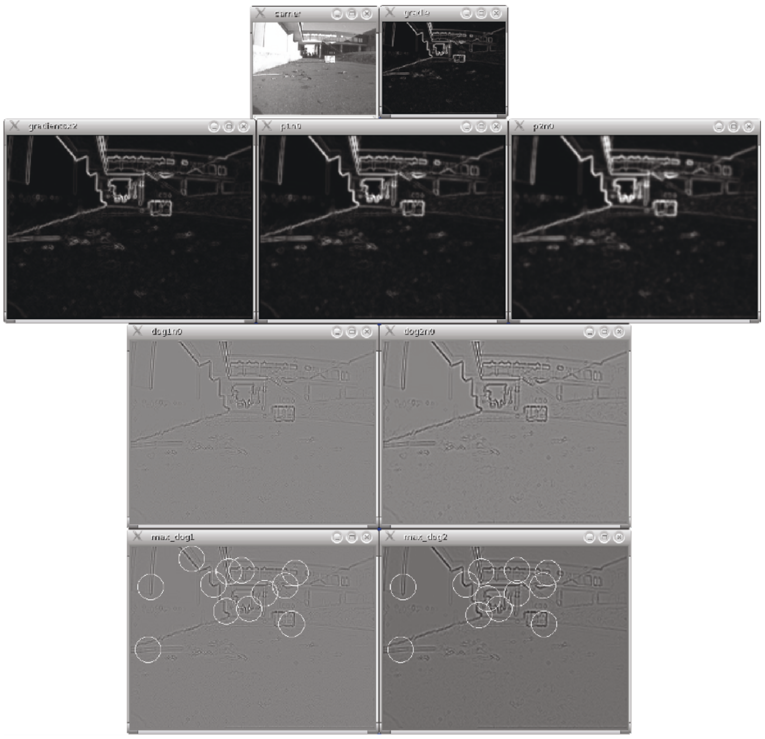

Figure 11: Graphical results of the SystemC functional model simulation obtained by using the `gtk` + 2.0 C++ library.

Table 2: Comparison between accuracy and simulation time of the OS model.

Board measurements	Simulation estimations	Error percentage	Simulation duration per image
29268.570400 ms	28369.598240 ms	3.07%	192 ms

application is possible at this level. The SystemC kernel and its API model the pure virtual architecture that executes the application. Due to the SystemC concurrency principle, an infinite number of resources is simulated at this level. Moreover, additional C++ libraries can be used for debug purpose and can give a functional trace of the application similar to the one illustrated in Figure 11.

4.1.2. *Executive model*

The second model refines the previous one by adding an explicit RTOS layer in the application model. At this level, a SystemC module models the whole application. This main module only contains the first main task of the application that acts as the boot code of a real-embedded RTOS: it initialises the scheduler (`OSInit`() service), creates the first task, and launches the OS scheduler (`OSStart`() service). The application source code has access to the RTOS services through the whole RTOS API. All accesses to SystemC primitive channels are thus replaced by OS calls. This can be done automatically by a simple C/C++ compiler preprocessing step. In addition, execution time of tasks may be modelled by `os_wait`() calls. No further modifications of the application code are necessary and a very fast validation can then be done. At this level, a number of strategies can be explored for customising the needed RTOS layer (scheduling policies, preemption model, and implementation of some optional services). Exploring the software or hardware implementation of the applicative tasks is not yet addressed at this level. The main objective is to explore the global sequencing of operations.

4.1.3. *Distribution model*

The distribution model is built by instantiating multiple (eventually different) OS models (see Figure 13). All models have access to the applicative C++ functions given in the specification source code and a global multiprocessor evaluation is allowed. In this case, each RTOS node integrates some specific modules dedicated to inter-OS synchronisation and communication. The associated RTOS services can then be explored at this level. One can note that simulating the architecture at this level allows an exploration of the number and type of resources without explicitly modelling processing resources (microprocessors). Up to this level, communication infrastructure refinement will be done by following the transaction level modelling (TLM) supported by the SystemC language [6].

4.1.4. *Implementation model*

This model is the last step before the physical synthesis of the platform. Hardware parts (memories, I/O devices, interrupt

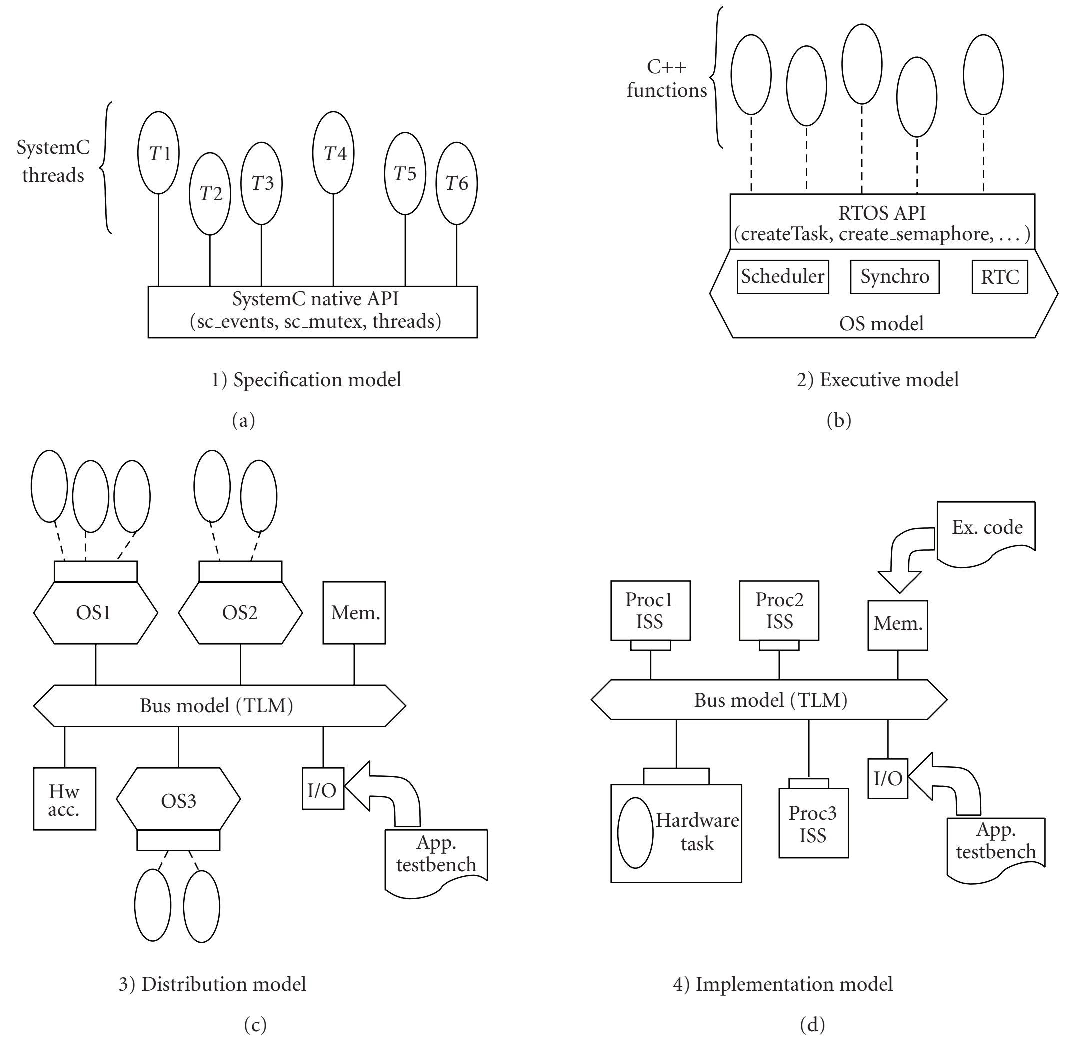

Figure 12: The proposed platform refinement flow starts with the specification model and provides at least four refinement steps until the final implementation model.

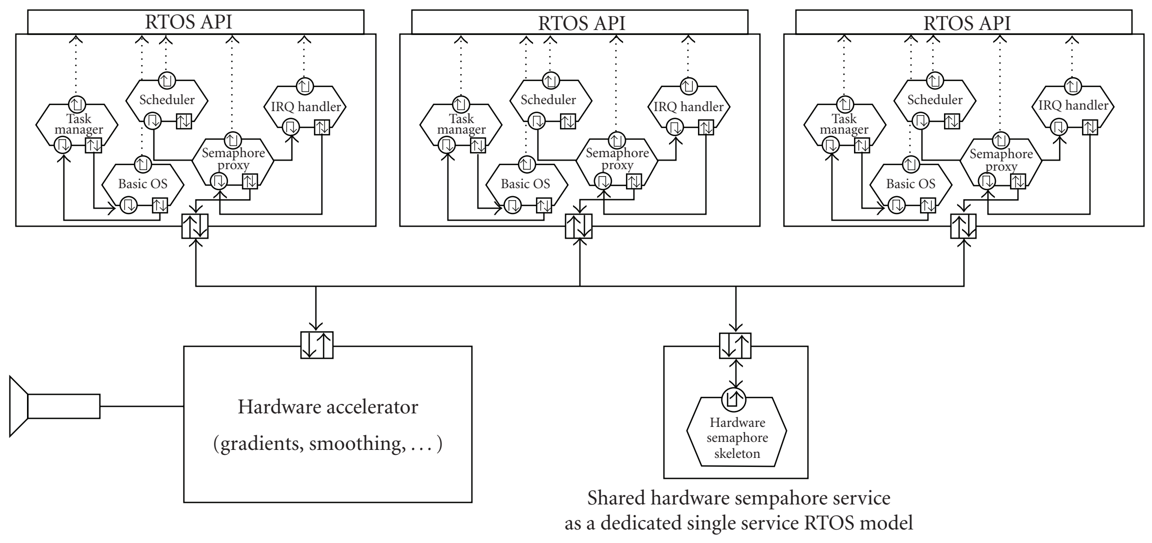
RTOS API
Task manager
Scheduler
IRQ handler
Semaphore proxy
Basic OS
RTOS API
RTOS API
Hardware accelerator
(gradients, smoothing, . . .)
Hardware semaphore skeleton
Shared hardware sempahore service
as a dedicated single service RTOS model

Figure 13: MPSoC platform modelled as a collection of interacting RTOS models.

handlers, hardware accelerators, etc.) of the platform are described as register transfer level (RTL) models. Processors are represented by their instruction set simulators. The application source code must be compiled and inserted in the platform model in the form of executable binaries. Simulation results can then be accurate at the clock cycle level (Cycle Accurate) or even at the signal level (Bit Accurate). At this level, the exploration and validation of a number of strategies have been done and the following physical synthesis of the platform can be realised by classical hardware design tools. We are not concerned in developing models at this level of abstraction.

This exploration methodology can be applied to a lot of different platform architectures and application domains. This flow is particularly well adapted to dynamic contexts, applications, and environments.

4.2. *Modelling a distributed architecture*

This part presents the latest results obtained when using the RTOS model presented in Section 3 for modelling a distributed multiprocessor architecture for the vision application.

Based on the presented SystemC RTOS model, and by taking advantage of modularity and genericity of SystemC models, we have developed a multi-RTOS model. At this level of abstraction, each processing element is represented by a single RTOS model. Each RTOS is responsible in executing its own part of the application (we assume that the application partitioning is done at design time). The application thus comes in the form of one main task per execution node. These main tasks are responsible in the creation of all application tasks and interrupt handlers. Each RTOS is also augmented by a communication port through which it can communicate with external world.

A common shared service is necessary to insure communication and synchronisation between tasks executed on different nodes. A shared semaphore module has been easily modelled by adding an extra RTOS model representing a unique semaphore service module. This module would be either refined as a single processor running a dedicated semaphore service or a hardware interprocessor synchronisation block. Figure 13 illustrates how multiple RTOS models can interact by using this shared module. Each local semaphore service module has been replaced by a proxy module implementing multi-RTOS communications. We inspired from the CORBA philosophy where proxies are used to implement a *service invocation* and skeletons are in charge of implementing the service itself. Distant communications between proxies and skeletons are modelled as simple method invocations and can be managed by any transport layer as for a TLM infrastructure for example.

5. EXPLORING THE APPLICATION ARCHITECTURE

Based on the presented design flow, we use our modelling framework to explore the architecture of the vision application. As described in Section 2.2, we made the profiling of the entire application on an embedded platform. We also built the profile of the μC/OS-II real-time services (deterministic). The timing data were measured and back-annotated into the high-level model in order to explore and evaluate the architecture dimensioning and the implementation strategies: tasks distribution, services distribution, scheduling algorithms, and so forth.

Table 3: Software profiling of the significant application tasks Average execution times on 20 representative images.

Task	Average execution time (ms)	Percentage
Gradients	13915.4	49%
HF Gaussian filtering	9795.8	35%
LF Gaussian filtering	426.0	1.5%
HF DoGs	443.0	1.6%
LF DoGs	15.3	0.05%
HF Searches	1286.1	4.6%
HF Extracts (+ norm.)	58.8	0.21%
LF Searches	45.1	0.16%
LF Extracts (+ norm.)	33.1	0.12%
Others (MF tasks, sampling)	2238.2	7.9%
Total	28257.2	100 %

As illustrated in Table 3, the on-board measurements helps to determine the critical portions of the application. In high-detail mode the gradient and the HF Gaussian filters represent more than 80% of the total execution time. Moreover, on the software implementation, all the treatments realised on high frequencies, except the extract task, are very critical and exceed the real-time constraint of the high-detail mode of the application: 1000 milliseconds. More generally, the actual software implementation of the common gradient task is incompatible with any of the three identified real-time behaviours.

For these reasons, we used the modularity advantage of the OS model to evaluate the gain of parallelism on the execution time of the three identified modes. Figure 14 shows the potential gain using multiple processors (from 2 to 5) for the high-detail mode. The better software partitioning among the explored ones is depicted on Figures 2 and 16; but the parallelisation has no significant effect beyond 2 processors. Indeed, the concurrency in the application only appears between the sequential Gaussian pyramid and the different scales (searches and extractions tasks), and between the normalisation tasks inside each scale, the latters only represent a small percentage of the total software application time. In addition, as illustrated in Table 3, the difference of complexity between scales explains the nonsignificant gain for the parallelisation of the normalisation tasks on a third processor.

Hence, only a hardware and software implementation could thus respect our variable constraints by accelerating the Gaussian pyramid.

According to the results of the first exploration phase, the identified critical and regular treatments (gradient, HF Gaussian filters, and DoGs) are candidates to a static

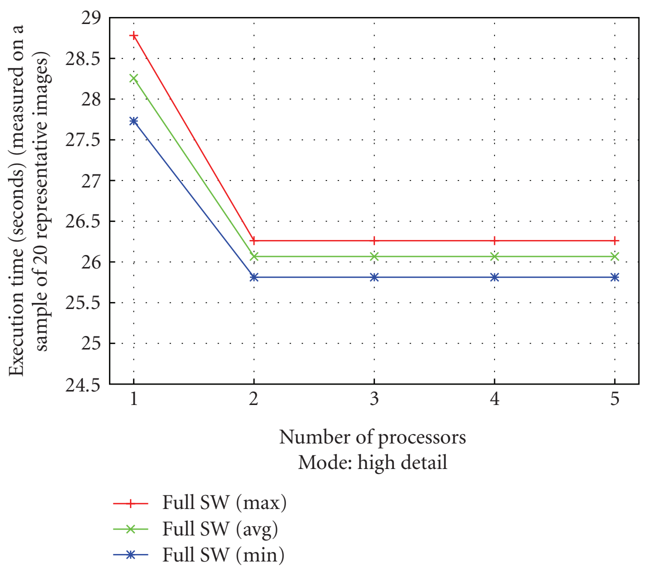

Figure 14: Execution times for different numbers of processors, in a full SW implementation.

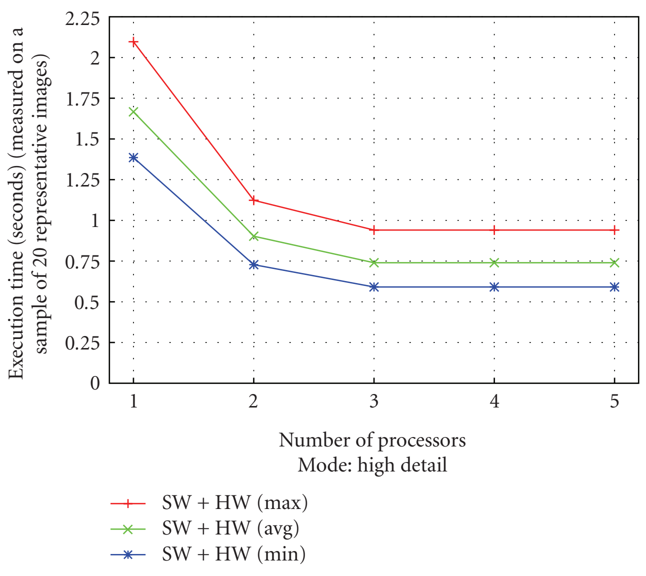

Figure 15: Execution times for different numbers of processors, on our hybrid HW/SW architecture.

hardware implementation as dedicated accelerators. These results conduct to a first-refinement process. Indeed, in order to evaluate the acceleration, we developed a VHDL description of the selected tasks. The temporal characteristics reported by the hardware synthesis tool (Altera's Quartus II for our example) were integrated into the model. The corresponding hardware tasks were modelled as independent and concurrent SystemC threads with back-annotated execution times. In addition, each hardware block provides an interruption line for synchronisation with the software part when data are produced. From the identified Hw/Sw partitioning, we led a second set of experiments resulting on the performance evaluations depicted in Figure 15 (the figure only represents results for the high-detail mode). Comparing Figures 14 and 15, we found a speed up factor of x17, thanks to the hardware acceleration. The hardware implementation of the pyramid also makes the parallelisation effects more significant on the third processor.

However, the challenge of the exploration was to find an architecture respecting the constraints corresponding to the three application modes. Exactly each mode executes different treatments under different rates. The variation of constraints finally leads to three embedded architectures that are presented in Figure 16. In fast mode, the lower scales are implemented on two processors. If two processors or more are used, the worst-case execution time of the application is about 48 milliseconds thus corresponding to the robot speed. In intermediate mode, 3 processors are needed leading to a total execution time under 150 milliseconds. In this mode, since the period constraint is relaxed, the two half lower scales can be implemented on the same processor. Finally, in the high-detail mode, the search and extract tasks of the two high-frequency half scales are processed on separate processors while low and medium frequencies are executed on a single processor. The maximum total execution time in this mode is 950 milliseconds on our three processor model. In the three cases, the architecture contains at least two common components: the hardware-gradient accelerator and a processor executing at least the low-frequency tasks. The performance results obtained with the following Hw/Sw solution are summarised in Figure 17.

The final system meets all the application requirements in each mode.

As a conclusion, we have successfully modelled a realistic multiprocessor platform with our distributed OS model. Thanks to the properties of the model, we have explored and defined the architecture adapted to the application requirements. The considered target platform is a System on Programmable Chip (SoPC) from ALTERA [30].

With our conclusions, an FPGA will be configured with 3 RISC microprocessors (Nios II) and the selected hardware blocks (accelerator and semaphore). Each processor will run an instance of a custom RTOS refined from the OS model. The interprocessor synchronisation will be realised through the shared hardware semaphore service. The entire application can thus be implemented on a single chip SoC.

In order to optimize the mode changes, we are now interested in refining the OS in order to support new specific services such as online software migration and dynamic voltage scaling (DVS) scheduling since only two of the three processors are useful in fast mode. These mechanisms will be managed by the dedicated and distributed OS. Another interesting perspective is the management of dynamically hardware reconfigurable units in order to propose a hardware adaptive architecture.

6. CONCLUSION

We have presented in this paper a particular design problem dealing with the SoC implementation of a visual system

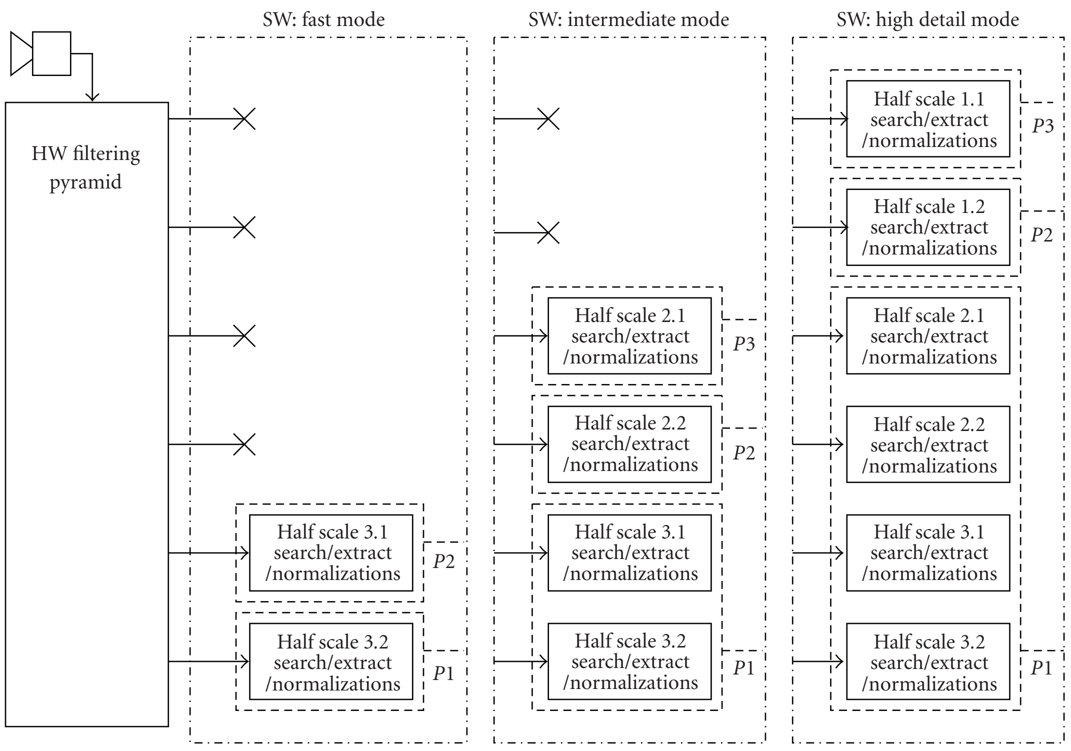
SW: fast mode
SW: intermediate mode
SW: high detail mode
HW filtering pyramid
Half scale 1.1 search/extract /normalizations
Half scale 1.2 search/extract /normalizations
Half scale 2.1 search/extract /normalizations
Half scale 2.2 search/extract /normalizations
Half scale 3.1 search/extract /normalizations
Half scale 3.2 search/extract /normalizations
P1
P2
P3

Figure 16: Architectures corresponding to the 3 application modes.

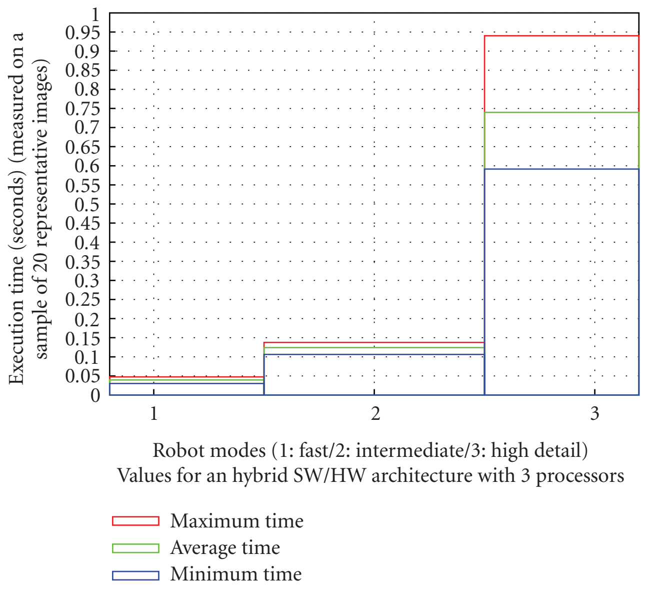

Figure 17: Execution times comparison on different modes on our hybrid architecture (HW + multiprocessor SW).

embedded in a mobile robot. The image-processing application works on a multiscale pyramidal decomposition of the images and extracts local features in the neighbourhoods of interest points. Such an application could be used for objects localisation, tracking, or recognition.

Since this application is inserted into a biologically inspired sensorimotor loop, it participates to the global system dynamics. As a first contribution, we precisely analysed the impact of the dynamics on the application behaviour. It has been shown that a lot of characteristics dynamically vary according to the regulation process imposed by the global system.

We have also shown that these variations are unpredictable without an entire a priori knowledge on the environment. We have thus demonstrated that classical design techniques based on static scheduling will fail to efficiently implement such an application. Only a custom RTOS with dedicated services could manage the adaptation of the architecture to the environment variations. The exploration and the definition of the required architecture have been made possible thanks to a high-level executable model of RTOS. This model facilitates early system dimensioning and application partitioning.

Our second contribution consists in detailing the bare dynamic mechanisms necessary to build a SystemC RTOS model. These mechanisms mainly provide the dynamical creation of SystemC processes and the preemption and execution time modelling. We have chosen the SystemC language for its ability to model and simulate both hardware and software systems at multiple levels of abstraction. We have also presented in this paper an operational executable RTOS model including these mechanisms. This model has been used to simulate the vision application on a representative set of data.

Finally, we have described how this RTOS model can be modified in order to be used for building a distributed architecture model. We have constructed the model of a realistic MPSoC platform containing multiple execution nodes and a shared-hardware semaphore. The application partitioning and the architecture dimensioning have been made by using and customizing our generic model during the proposed exploration process. The obtained results constitute a promising way for the final SoC design. More generally, this work falls into the scope of the OveRSoC project [31] that aims at developing an exploration methodology adapted to the design of dynamically reconfigurable systems. The high-level SystemC RTOS model presented in this paper is expected to be used also for the exploration of custom RTOS services dedicated to the management of these particular dynamically reconfigurable resources.

REFERENCES

[1] D. H. Ballard, "Animate vision," *Artificial Intelligence*, vol. 48, no. 1, pp. 57–86, 1991.

[2] P. Gaussier and S. Zrehen, "PerAc: a neural architecture to control artificial animals," *Robotics and Autonomous Systems*, vol. 16, no. 2–4, pp. 291–320, 1995.

[3] M. Maillard, O. Gapenne, L. Hafemeister, and P. Gaussier, "Perception as a dynamical sensori-motor attraction basin," in *Proceedings of the 8th European Conference on Advances in Artificial Life (ECAL '05)*, M. S. Capcarrère, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis, Eds., vol. 3630 of *Lecture Notes in Computer Science*, pp. 37–46, Canterbury, UK, September 2005.

[4] P. Gaussier, C. Joulain, J. P. Banquet, S. Leprêtre, and A. Revel, "The visual homing problem: an example of robotics/biology cross fertilization," *Robotics and Autonomous Systems*, vol. 30, no. 1-2, pp. 155–180, 2000.

[5] A. A. Jerraya and W. Wolf, "The what, why, and how of MPSoCs," in *Multiprocessor Systems-on-Chips*, chapter 1, pp. 1–18, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[6] "SystemC standard," http://www.systemc.org/.

[7] "SpecC language," http://www.specc.org/.

[8] Y. Sorel, "SynDEx: system-level cad software for optimizing distributed real-time embedded systems," *Journal ERCIM News*, vol. 59, pp. 68–69, 2004.

[9] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Norwell, Mass, USA, 1995, foreword By-Giovanni De Micheli.

[10] G. De Micheli, Ed., "Special issue on hardware/software co-design," *Proceedings of IEEE*, vol. 85, no. 3, 1997.

[11] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: hardware-software co-synthesis of embedded systems," in *Proceedings of the 34th Design Automation Conference*, pp. 703–708, Anaheim, Calif, USA, June 1997.

[12] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS modeling for system level design," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 130–135, Munich, Germany, March 2003.

[13] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for systems-on-chip," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 396–401, Los Angeles, Calif, USA, June 2000.

[14] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic RTOS model for real-time systems simulation with SystemC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '04)*, pp. 82–87, Paris, France, February 2004.

[15] "Cofluent Studio$^{\text{TM}}$," www.cofluentdesign.com.

[16] L. Gauthier, S. Yoo, and A. Jerraya, "Automatic generation and targeting of application specific operating systems and embedded systems software," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 679–685, IEEE Press, Munich, Germany, March 2001.

[17] S. Leprêtre, P. Gaussier, and J. Cocquerez, "From navigation to active object recognition," in *Proceedings of the 6th International Conference on Simulation of Adaptive Behavior (SAB '00)*, pp. 266–275, Paris, France, August 2000.

[18] T. Lindeberg, "Feature detection with automatic scale selection," *International Journal of Computer Vision*, vol. 30, no. 2, pp. 79–116, 1998.

[19] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[20] K. Mikolajczyk and C. Schmid, "Scale & affine invariant interest point detectors," *International Journal of Computer Vision*, vol. 60, no. 1, pp. 63–86, 2004.

[21] M. Seibert and A. M. Waxman, "Spreading activation layers, visual saccades, and invariant representations for neural pattern recognition systems," *Neural Networks*, vol. 2, no. 1, pp. 9–27, 1989.

[22] J. Crowley, O. Riff, and J. H. Piater, "Fast computation of characteristic scale using a half-octave pyramid," in *Proceedings of the International Workshop on Cognitive Computing (Cogvis '02)*, Zurich, Switzerland, October 2002.

[23] A. Torralba and A. Oliva, "Statistics of natural image categories," *Network: Computation in Neural Systems*, vol. 14, no. 3, pp. 391–412, 2003.

[24] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '03)*, vol. 2, pp. 257–263, Madison, Wis, USA, June 2003.

[25] E. L. Schwartz, "Computational anatomy and functional architecture of striate cortex: a spatial mapping approach to perceptual coding," *Vision Research*, vol. 20, no. 8, pp. 645–669, 1980.

[26] D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*, W.H. Freeman, San Francisco, Calif, USA, 1982.

[27] R. A. Brooks and L. A. Stein, "Building brains for bodies," *Autonomous Robots*, vol. 1, no. 1, pp. 7–25, 1994.

[28] P. G. Schyns and A. Oliva, "Dr. Angry and Mr. Smile: when categorization flexibly modifies the perception of faces in rapid visual presentations," *Cognition*, vol. 69, no. 3, pp. 243–265, 1999.

[29] E. Huck, B. Miramond, and F. Verdier, "A modular systemC RTOS model for embedded services exploration," in *Proceedings of the 1st European Workshop on Design and Architectures for Signal and Image Processing (DASIP '07)*, Grenoble, France, November 2007.

[30] Altera Corp, "Creating Multiprocessor Nios II systems, ver. 1.3," December 2007, http://www.altera.com/literature/lit-nio2.jsp.

[31] I. Benkhermi, A. Benkhelifa, D. Chillet, S. Pillement, J.-C. Prévotet, and F. Verdier, "System-level modelling for reconfigurable SoCs," in *Proceedings of the 20th Conference on Design of Circuits and Integrated Systems (DCIS '05)*, Lisboa, Portugal, November 2005.