Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 285160, 10 pages
doi:10.1155/2008/285160

Research Article

Reconfiguration Management in the Context of RTOS-Based HW/SW Embedded Systems

Yvan Eustache and Jean-Philippe Diguet

LESTER Lab, CNRS/UBS, 56100 Lorient, France

Correspondence should be addressed to Yvan Eustache, yvan.eustache@univ-ubs.fr

Received 1 April 2007; Revised 24 August 2007; Accepted 19 October 2007

Recommended by Alfons Crespo

This paper presents a safe and efficient solution to manage asynchronous configurations of dynamically reconfigurable systems-on-chip. We first define our unified RTOS-based framework for HW/SW task communication and configuration management. Then three issues are discussed and solutions are given: the formalization of configuration space modeling including its different dimensions, the synchronization of configuration that mainly addresses the question of task configuration ordering, and the configuration coherency that solves the way a task accepts a new configuration. Finally, we present the global method and give some implementation figures from a smart camera case study.

Copyright © 2008 Y. Eustache and J.-P. Diguet. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

1.1. Self-adaptive RTOS-controlled SoC

Upcoming embedded systems will implement complex multistandard multimode applications competing for resources within a heterogeneous architecture. One of the most important challenges in this context is the tradeoff between flexibility needed for fast design of mass products, performance, power management, and quality of service (QoS). This tradeoff can only be partially obtained at design time. CPU and communication loads vary with tasks' changeable execution times. This can be due to network access conditions, data values, architecture hazards (e.g., cache miss), or user choices.

One of the most promising directions to deal with such constraints is the reconfigurable architecture that can be tuned accordingly to resource requirements.

Embedded systems, in the domain of wireless networks and multimedia applications, manage concurrent applications with fluctuating loads. It means that reconfiguration requests are not usually synchronized with running tasks and can happen at any time independently of data and control flows of concurrent applications. In such a context, one of the main tedious problems is to guarantee a safe reconfiguration synchronization that takes care of data dependencies and algorithm coherency. The resulting complexity of the system control requires RTOS services for synchronization, communication, and concurrency management. In addition to that, extensions of RTOS services are needed for configuration management in order to provide two kinds of capabilities: (i) transparent hardware and software communication services and (ii) configuration management in order to facilitate the design of complex heterogeneous systems. Another important RTOS advantage is the abstraction to increase portability and to facilitate reuse of code and repartitioning.

The objective of this work is to formalize and implement an abstraction layer suitable for usual RTOS in order to handle hardware and software communications and configuration management. In our experiments, we use μCOSII which is implemented on different targets such as NIOS, PowerPC, and MicroBlaze cores on Altera and Xilinx devices, respectively.

The background of this paper is our solution for implementing self-adaptive systems. The foreground is the tedious question of configuration switching in the context of hardware and software implementations. This point is generally simplified, whereas it raises in practice complex questions about configuration granularity and synchronization.

The rest of the paper is organized as follows. Section 2 provides a description of our adaptive system model including the HW/SW unified interface and the configuration model. In Section 3, we present the main issues and solutions

involved with the reconfiguration and propose the new concept of configuration granularity to manage a coherent configuration. Finally, a smart camera case study is introduced in Section 4 to validate our approach.

1.2. Related works

In this section, we survey a selection of related works in the area of RTOS-based reconfigurable SoC. Firstly a lot of work has been produced in the domain of adaptive architectures. Different techniques have been introduced for clock and voltage scaling [1], cache resources [2], and functional units [3] allocations. These approaches can be classified in the category of local configurations based on specific aspects. Our aim is to add global configuration management including both algorithmic and architectural aspects.

Secondly RTOS for HW management has been recently introduced. Proofs of concepts are exhibited in [4, 5]. These experiments show that RTOS level management of reconfigurable architectures can be considered as being available from a research perspective. In [5], the RTOS is mainly dedicated to the management (placement/communication) of HW tasks. Run-time scheduling and 1D and 2D placement of HW tasks' algorithms are described in [6]. In [4], the OS4RS layer is an OS extension that abstracts the task implementation in hardware or software; the main contribution of this work is the communication API based on message passing where communication between HW and SW tasks is handled with a hardware abstraction layer. Moreover, the heterogeneous context switch issue is solved by defining *switching points* in the data flow graph, but no computation details are given. In [7], abstraction of the CPU/FPGA component boundary is supported by HW thread interface concept, and specific RTOS services are implemented in HW. In [8], more details are given about a network-on-chip communication scheme.

The current state of the art shows that adaptive, reconfigurable, and programmable architectures are already available, but there is no real complete solution proposed to guarantee safe configuration management. Some concepts are presented but no systematic solutions are proposed.

2. ADAPTIVE SYSTEM MODEL

2.1. System model

The targeted system is a multitask heterogeneous system composed by an embedded processor for the SW tasks and accelerators implementing HW tasks; both are masters on a communication medium, which is a shared bus or a network-on-chip. Such a system improves performances since several data-dependent or independent tasks can run concurrently. Communication between data-dependent tasks is based on shared memories for data and RTOS message passing services for events and address notification (e.g., mailbox and message queue). Data-dependent tasks are gathered in a cluster; each cluster can have one or more source tasks and sink tasks; a minimum cluster is reduced to a single task. At the highest level, applications can be composed by several clusters.

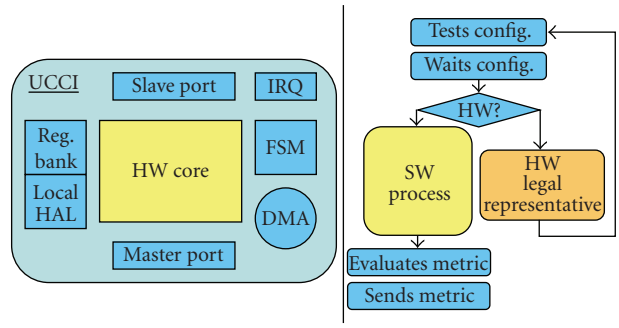

Figure 1: Unified communication and configuration interface.

We define the model of the system as a directed acyclic graph where a task is represented by a node and the shared memory by an edge connecting two nodes (details are given in Section 3.3.3).

2.2. UCCI concept

2.2.1. OS services access for HW tasks

Abstract HW/SW interface for an embedded system-on-chip is an important topic. It allows communications between tasks independently of their implementations. In our adaptive strategy, communications are not limited to data since local data (task context: recursive data, counters), configuration (pointers, configuration ID), and metrics are also exchanged. Thus, task communications are encapsulated within a unified interface called UCCI (see Figure 1) that handles configuration, data, and control communications in such a way that usual SW and HW implementations can be directly instantiated in the new framework.

Each HW task has a legal representative (LR) which is a lightweight SW task that maintains communications between RTOS and tasks implemented in hardware. Local data can be read (resp., written) by the LR in case of HW to SW (resp., SW to HW) migration or by the HW UCCI in case of HW to HW migration. Basically, a context transfer is equivalent to a data transfer performed between reconfigurations in case of HW to SW or HW to HW migrations.

The UCCI implementation imposes some constraints regarding I/O format. For instance, local data are based on identical stacks for all HW and SW versions. Moreover, some SW routines have been added for data format adaptation; the more used one is the bit/byte conversion routine required when SW tasks accede to binary images where pixel is coded as bit for bandwidth and area optimization.

2.2.2. Tasks communication

Communications with an SW task are involved to call the intertask communication RTOS services. Two kinds of operation can be used: post and pend communications. For SW → SW communications, traditional communication services are used, whereas in HW ↔ SW communications, the LR task

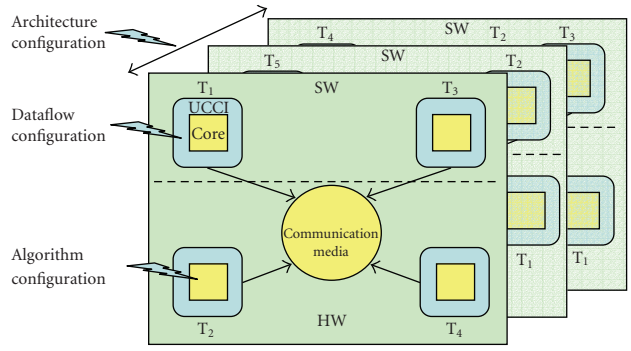

Figure 2: Configuration space model.

is mainly used to handle the HW pend request. Postservices are supported directly by the interrupt service routine (ISR). If using RTOS communication services appears to be essential to abstract the communication between tasks, the overhead due to the HW to LR communication via the ISR and the involved context switches can be important for an application using mainly message passing. Communication overhead between two HW tasks via RTOS communication service has been drastically reduced while delegating post and pend communication management to HW UCCI which provides message passing services. Thus, a first task can write directly the message in a specific register of a second HW task. "Pend" is represented by a wait state until a message is written in the register. Other services such as mutex, semaphore, and flag services are provided by the RTOS and not implemented in UCCI since no distributed implementation was justified. However, independently of our work, some specific HW implementations of mutex, for instance, can be added to improve performances as shown in [9].

UCCI concept brings efficient communication between HW tasks. However, control is added to the HW task to manage direct or OS message passing according to the implementation of the other tasks (more details are given in [10]).

2.3. Configuration space modeling

Based on our analysis, we propose to partition the configuration space into three levels (see Figure 2).

Algorithm reconfiguration targets the functionality of a task. A new configuration may occur to improve performance, QoS, or energy. In this paper, we consider that each configurable task implements different algorithms with various complexity/performance tradeoffs; it can be, for instance, different lowpass filters based on an average of a variable number of images (2, 3, 4), image processing based on various mask sizes, or image thresholding based on static or dynamic thresholds.

Data flow reconfiguration represents the intertask communication scheme at the application level. The data flow configuration mainly impacts task UCCI and principally communication parameters (read/write addresses) located in the HAL. Such a configuration is used, for instance, when a task is inserted in or removed from the application. Since data flow and algorithm modifications are very close in terms of frequency and management, we gather these two kinds of configurations under the term of algorithm configuration in the rest of the paper.

Architecture configuration is the arrangement of HW and SW task implementations for a set of tasks with associated selected algorithms. Architecture reconfiguration may occur to migrate an SW task to an HW implementation. In opposition, the system may switch a task from HW to SW. However, this possibility of task migration leads to important time and resource overheads (HW ↔ SW context store/load, bitstream generation, and download). In consequence, the modification of an architecture cannot occur at the application frequency. A tradeoff has to be solved between the rate of architecture reconfiguration and the gain it brings. We assume that HW dynamic configuration is available, but we do not address this point which is very technology-dependent (e.g., Xilinx bitstream loading). Besides, we have implemented our smart camera prototype on an Altera StratixII FPGA, which does not allow for dynamic reconfiguration; so all tasks are implemented at design time. Thus, HW task activation is implemented with a clock gating control.

Finally, reconfiguration raises the questions of algorithm/architecture selection, task activation control, communication scheme setup, and HW clock management. So, a configuration can be modeled by a unique configuration identifier (CID), which represents a point in a three-dimensional space, namely, the combination of a set of algorithms, a data flow, and an architecture.

2.4. Configuration management

2.4.1. Of/online configuration management

Embedded systems are characterized by scarce resources in terms of energy, memory, and processing to be used efficiently. Our low-cost configuration management reduces the time and memory overheads separating offline selection and adaptation at run time. The first step of the management is the offline selection of potential configurations. The space of configuration (algorithm, data flow, and architecture configurations) is reduced to a unit of best configurations, in terms of performance, consumption, and QoS. An incertitude due to environment fluctuations or internal risks (cache misses, bus collisions, etc.) is also taken into account. In addition, the system adapts itself to its configuration by selecting online the best configuration within the preselected configuration finite space. Thus, no generation of configuration is processed and adaptation time is minimized.

2.4.2. Local/global separation of concerns

Online adaptation issue encompasses different aspects that drive the implementation of the reconfiguration decision control. The first relevant point is locality. A reconfiguration can be decided at the application level or system level. Based on application-specific data, a local decision provides a short reaction delay and metrics to compute the QoS. However, some decisions must be considered globally when a tradeoff

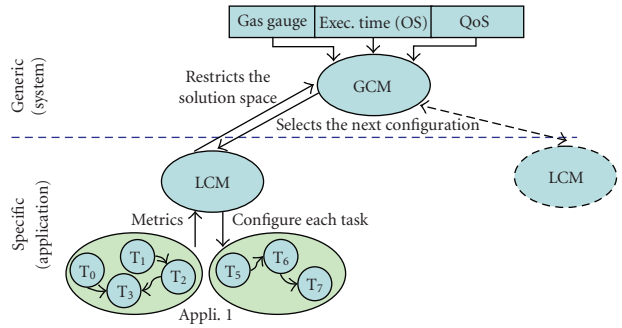

Figure 3: Application versus system configuration management.

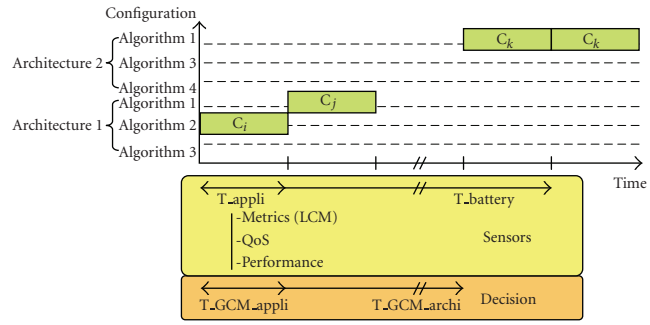

Figure 4: Configuration management timing.

has to be found over the complete system between power consumption and computation efficiency. Another pertinent point is the complexity of the decision implementation. In the context of embedded systems, only low-cost solutions must be considered. To cope with these issues, we implement a two-step configuration management (see Figure 3). The local configuration manager (LCM) is the first level; it is application-specific. The second one is the global configuration manager (GCM); it is generic and so independent of embedded applications.

LCM is in charge of the algorithm configuration for all the tasks of the application it controls. Given results from application tasks, it first transforms application-specific metrics into normalized QoS metrics for the GCM. This is a kind of "application sensors" for the global manager. Secondly, it selects the minimum algorithm configuration to be considered by the GCM. Finally, the LCM controls the application configuration while applying the GCM decisions.

GCM is in charge of global system parameters (e.g., I/O data rates) and HW/SW implementation decisions. It collects information for configuration decisions from sensors such as CPU load from the OS, application-specific QoS from LCMs, and battery level and power from the gas gauge controller or from estimators when no measures are available. The GCM decides upon the new system configuration according to user requirements (system references) and configuration solutions issued from the local managers' design space restrictions.

LCM and GCM as new RTOS services

LCM is application-specific; it can be interpreted as a kind of driver or abstraction layer to get algorithmic configuration wishes issued from application tasks and to send new configurations to application tasks. A GCM is a new RTOS service comparable to scheduling or resource allocation.

In practice, LCM and GCM can be implemented as high-priority tasks within an existing RTOS (e.g., μCos). In such a scheme, each LCM pends on a message queue where application metrics are loaded and reacts according to its configuration management policy. The GCM wakes up each time a configuration algorithm is notified and reacts according to its configuration management policy. A simple policy consists in waiting all notifications from application tasks or LCM, and it decides upon the new configuration compliant with user requirements.

2.4.3. Configuration management timing

Configuration frequency is also an important issue regarding configuration overhead. As shown in Figure 4, application execution time is defined as the base period; after each execution, the LCM receives metrics from tasks and collects QoS and execution time. Sensors acquisition (e.g., battery level) is managed with lower frequencies and can use linear estimators [11] to alleviate the measure overhead. In opposition, battery sensor, due to its important overhead of resources (control, computation time, consumption), is monitored with a lower frequency. At the application period, the GCM selects suitable algorithm configuration according to the actual architecture. The GCM is not synchronized with LCMs. In case of a single application, its period is the application one with an important restriction due to the configuration overhead of current reconfigurable SoC (FPGA). Thus, when architecture reconfiguration occurs, a *provision* period is computed; during this period, the GCM can only allow algorithmic reconfigurations. This two-step configuration management is very flexible and allows a short reaction delay for algorithm adaptation compared to the architectural modification.

3. SYNCHRONIZATION AND COHERENCY

Adaptation process involves three main steps. First, the system has to be able to monitor its environment. For example, in our smart camera case study, the LCM locally uses the number of objects and the image resulting noise (number of isolated white pixels) to configure/select algorithms. Moreover, the LCM provides GCM with the application QoS being computed as the difference between object position based on image processing and estimations based on linear extrapolations. Other global information used is the battery level and task execution times.

Based on this set of information, the system (GCM) takes a decision (e.g., selection of a more robust 2D low-pass filter). Finally, it must guarantee a safe transition from an old to a new configuration. We propose to partition it into three subissues: local data management, communication

dependencies, and functional dependencies that guarantee application coherency.

3.1. Issues

Local data management: architectural reconfiguration raises the question of management of local data. Close to the SW context switch, HW task preemption involves saving processing data. It allows the task to resume its process with another implementation (*HW/SW* switches or HW move) with identical conditions. The main difficulty is to define a context model and store/load management.

Communication dependency: the communication topology can be modified according to data flow reconfiguration. So, the reconfiguration may be performed in a particular order compliant with communication dependencies to avoid memory or mail queue address mismatches. It means that a sender task will be configured before the task waiting for its data.

Functional dependency: configuration may also involve functional coherency management. However, if communication dependency is a run-time issue, management of functional coherencies can be processed offline. Such a modification may affect three process characteristics of a task: the functionality (e.g., compression/decompression algorithm), the amount of data (e.g., image size), and the data type (e.g., bit or byte processing). Thus, it may involve configuration dependencies between communicating tasks. Algorithm modification of configuration-dependent tasks involves defining configuration management to insure coherent data exchanges. On the other side, some algorithm configurations involve no configuration dependencies between tasks (e.g., the coefficient modification in a filtering task). In this case, the modification may be accepted at any time.

Finally, the problem of the coherency management is"how to be sure that a task receives sufficient homogeneous data from other tasks to complete its process and to provide sufficient homogeneous data to the downstream tasks before accepting a reconfiguration." This issue has to be solved regarding the complete system from source to sink tasks of all clusters and for each configuration.

3.2. Solutions

To solve the issues described above, we mainly target two points: synchronization which corresponds to the timing order to reconfigure the tasks, and algorithm coherency which corresponds to the integrity of data exchanged between tasks being reconfigured.

3.2.1. Local data management

Local data management is a key point in the reconfigurable HW system research topics and mainly for the preemption of HW tasks. Usually, it is solved by considering that HW tasks are not preemptible, and so no context needs to be saved. Walder et al. target this issue in [12] and indicate that such services must be included, but no more details are given. This issue is strongly technology-dependent. In our application case study, we define task local data context (LDC) as a task-specific stack containing resident computation data and data pointers, which are unique for both HW and SW tasks. For SW to HW and HW to SW reconfigurations, contexts can be communicated directly via the LR. Regarding HW to HW reconfigurations, we use the same scheme as that implemented for data communications: shared memories and message (addresses) passing. Thus, contexts are saved in a context memory reloaded after each configuration. LDC is automatically loaded after a reconfiguration as the initialization step before running.

3.2.2. Synchronization

Online synchronization issue can be solved with configuration diffusion mechanisms. Our concept is based on the utilization of existing communication channel (direct HW to HW configuration or with RTOS communication services). Thus, we define a two-step strategy. Firstly, the configuration manager sends the CID to all source tasks of each cluster through a multicast diffusion. Secondly, the CID is propagated gradually from the source to the sink tasks over data channels. More details are given in Section 3.4

With such diffusion principles, we guarantee that all tasks will be configured starting with the source tasks. Note that propagation principle increases the HW control. Indeed, HW tasks receive and send the configuration ID via OS communication services or directly according to the implementation of other tasks. Moreover, it has also to inform its LR when it receives directly the CID from an HW task.

3.2.3. Algorithm configuration coherencies

To solve configuration coherency, we introduce the new concept of configuration granularity that guarantees for each task consistent production of data. For each configuration and for each task, it can be computed offline, and it corresponds to a task static parameter.

3.3. Configuration coherency

3.3.1. Granularity concept

By providing algorithm configuration capability, adaptive system has the objective to handle safe transitions between such configurations. In the case of two configurations involving coherencies, the adaptation manager has to be assured that tasks reach a synchronization point before they are reconfigured; namely, enough data must be available or produced before a configuration can be accepted. To solve this issue, we introduce the concept of configuration granularity, *Gc*.

Definition 1. *Gc* corresponds to the data quantity a task has to produce for a given output before it accepts a new configuration. It guarantees that sink tasks will receive enough data to complete their cycle before switching to another configuration.

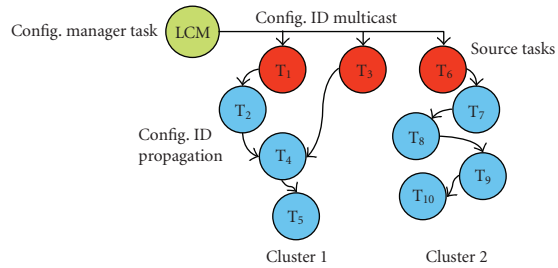

Figure 5: Synchronization with multicasting and propagation.

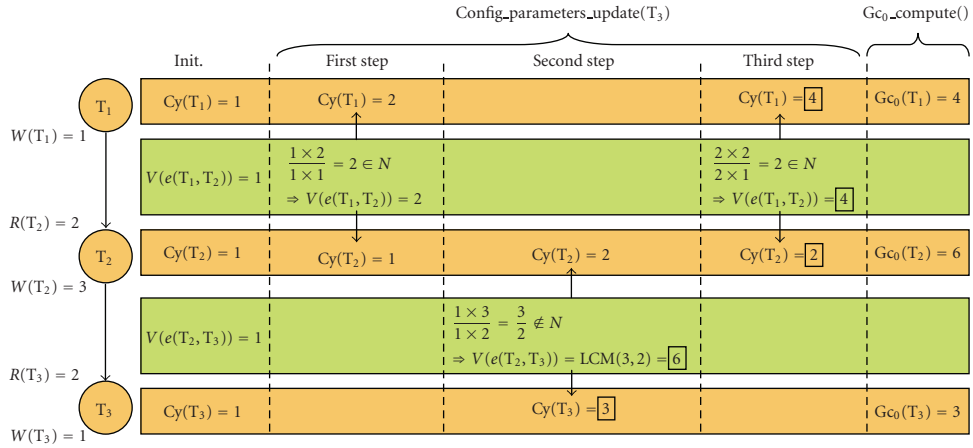

Figure 6: Gc_0 computation example.

3.3.2. Reconfiguration process

In our model, data communication is based on shared memories; it means that local data only deals with counters, residual, or recursive data. These are two distinct issues necessary for reconfiguration success, but they are driven by configuration granularity constraints. Thus, the configuration process is as follows:

(1) CID reception,
(2) no change until Gc is not reached,
(3) save local data context (if necessary),
(4) propagate CID (if successors),
(5) task configuration:

(i) HW↔ SW or HW→ HW migration, LDC loading,
(ii) datapath modification: HAL loading,
(iii) HW task suppression: bitstream removing or clock stopping, corresponding to SW task in SW state,
(iv) HW task insertion: bitstream loading, clock starting, or algorithm configuration, corresponding to SW task in LR state.

3.3.3. Graph model for a given configuration

Let $G_C(N, E)$ be the system configured with the configuration C, where a node $n \in N$ and a directed weighted edge $e \in E$ connecting two nodes represent a task and a shared memory between two tasks, respectively. The weight of the edge corresponds to the size of the minimum shared memory.

The configuration model is presented as follows.

(1) $nb(\text{pred}(n))$: predecessor number of node n.
(2) $\text{pred}(n)[i]$: predecessor node connected to input i of n.
(3) $\text{succ}(n)[i]$: successor node connected to output i of n.
(4) $R(n, o)$: amount of data produced by the predecessor node o; a node n reads per task cycle.
(5) $W(m, n)$: amount of data the precedent node m produces per task cycle.
(6) $\text{Cy}(n)$: number of task cycles needed by the node n to produce and consume data.
(7) $e(\text{pred}(n)[i], n)$: edge connecting the node n and its predecessor $\text{pred}(n)$ with the ith output of the node n. It represents the memory between $\text{pred}(n)$ and n.
(8) $V(e(\text{pred}(n)[i], e))$: value of the edge connecting the node n and its predecessor node $\text{pred}(n)$. The weight of the edge corresponds to the shared memory size.
(9) $\text{Gc}_0^n[j]$: configuration granularity of the node n on the edge connected to the jth output.

For a given configuration, $R(\cdots)$ and $W(\cdots)$ are constant characteristics of a node, whereas $\text{Cy}(n)$, $V(e(n, \text{pred}(n)[i]))$, and $\text{Gc}_0^n[j]$ must be homogenized.

Definition 2. Gc_0 can be deduced from the number of process cycles that a task needs to produce sufficient data

according to the given output writing constraints (the constant quantity of data produced per cycle). For a multioutput task, the configuration granularity of each output respects the following equation:

$$\forall i, \quad \frac{\mathrm{Gc}_0^n[i]}{W(n, \mathrm{succ}(n)[i])} = \mathrm{Cy}(n). \tag{1}$$

3.3.4. SDF analogy

Our system can be modelized by a synchronous data flow (SDF) [13], where one SDF model corresponds to our graph model for a given configuration. On one side, the SDF graph traveling goal is to find a periodic admissible schedule, meaning that tasks will be scheduled to run only when data are available and finite amount of data is required. On the other side, our goal is to find a homogeneous data computation configuration, meaning the cycle number of tasks to produce sufficient data before accepting a new configuration. The number of occurrences of a node then corresponds to the cycle number of a task in our graph model. Our contribution is to find the minimum value of configuration granularity $\mathrm{Gc}_0^n[i]$ for each output i of tasks n. Equation (1) gives the relation between $\mathrm{Gc}_0^n[i]$ and $\mathrm{Cy}(n)$.

3.3.5. Granularity computation

For a safe transition (no data famine and no blocking behavior), configuration granularity is computed for each coherency-dependent couple of producer-consumer tasks and for all configurations. Its statical characteristics allow us to do it offline. Granularity computation is realized for each path from all sink tasks. The result depends not only on consumer task characteristics but also on characteristics of all tasks composing the cluster.

We have developed an algorithm to compute the configuration granularity in three steps. First, we assume that the system is consistent in the sense of SDF formalism where for a connected SDF graph with s nodes and topology matrix Γ, $\mathrm{rank}(\Gamma) = s - 1$ is a necessary condition for a periodic admissible sequential schedule to exist. Real HW/SW applications are consistent and our solution can be applied.

We first initialize the cycle number, the configuration granularity, and the weight of the edges to one (at least one cycle for computation, one dataset produced, and one cycle for reconfiguration). The algorithm then homogenizes the edge weight and the cycle number of each node according to their read and write constraints. Finally, it computes the configuration granularity for each node output.

The Config_Parameter_Update() function travels each branch of the graph from the sink to the source tasks. The computation core checks if data quantity produced by the precedent task $\mathrm{pred}(n)$ is sufficient and corresponds to an integer value compared with the consumption of the task n. Otherwise, the least common multiplier between these two values is computed. The cycle number of the tasks $\mathrm{pred}(n)$ and n is then updated according to the read and write constraints, $R(n, \mathrm{pred}(n)[i])$ and $W(\mathrm{pred}(n)[i], n)$. Finally, modifications of the number of cycles are propagated to the upstream and downstream tasks by the recursive characteristic of the algorithm. Applying the Gc_0_Compute function for each task output, we guarantee the minimum exchange of data between tasks before a new configuration can be taken into account without data loss and blocking states.

Algorithm 1 is launched for all sink tasks and repeated until no modifications are observed.

It is applied on a simple example in Figure 6. In a cluster, three data-dependent tasks communicate in a single path via shared memories. The tasks are also configuration-dependent. In consequence, the configuration granularity is computed. T_1, T_2, and T_3 have to produce, respectively, 4, 6, and 3 data before accepting a reconfiguration.

With its recursive characteristic, Algorithm 1 targets also multipath graph. Each path is traveled from the sink to the source tasks.

An additional termination condition is added due to hardware limitation. Thus, intertask memory size is bounded by the resource limitations involved by embedded systems characteristics.

3.3.6. Application rescaling

While the configuration granularity Gc_0 represents the minimum protection against famine and blocking behavior, application constraints may impose rescaling of the configuration granularity. Rescaling is processed task by task according to application and user requirements:

$$\forall n, \quad \mathrm{Gc}^n = K^n \times \mathrm{Gc}_0^n \text{ with } K^n \in \mathbb{N}^*. \tag{2}$$

The scaling factor K^i of the task Ti can be computed according to the minimal configuration granularity Gc_0 and the application-constrained configuration granularity Gc_{appli}:

$$\forall n, \quad K^n = \frac{\mathrm{LCM}(\mathrm{Gc}_0^n, \mathrm{Gc}_{\mathrm{appli}}^n)}{\mathrm{Gc}_0^n}. \tag{3}$$

For instance, in a 20-row image application, the task $T2$, after system homogenization by granularity functions (see Figure 6), has a minimal configuration granularity equal to six rows, $\mathrm{Gc}_0^n = 6$ (the task accepts a reconfiguration after the computation of six rows). However, a designer can impose that the task cannot be reconfigured before the complete image process for QoS reasons (image homogeneity), $\mathrm{Gc}_{\mathrm{appli}}^n = 20$. The scaling factor K^n is then equal to $10(\mathrm{LCM}(6, 20)/6 = 10)$. Finally, the application-constrained configuration granularity imposes that the task produces 60 lines (3 images) before accepting a reconfiguration.

3.3.7. K-setup rules

Two add-on rules ensure a safe and coherent configuration transition. We consider $K^{\mathrm{pred}(n)}$ and K^n the scaling factors applied to the minimal granularity $\mathrm{Gc}_0^{\mathrm{pred}(n)}$ and Gc_0^n of tasks $T^{\mathrm{pred}(n)}$ and T^n (the producer task and the consumer task, resp.)

```
(1)  Initialization();
(2)
(3)  for all sink Task S do
(4)     Config_Parameter_Update(S);
(5)  end for
(6)
(7)  for all n do
(8)     Gc0_Compute(n);
(9)  end for
(10)
(11)  Config_Parameter_Update(n)
(12)
(13)  NP = nb(pred(n))
(14)  i = 0
(15)  while NP > 0 do
(16)     Config_Parameter_Update(pred(n)[i])
(17)     Modif(Cy(n)) = 0
(18)     Modif(Cy(pred(n)[i])) = 0
(19)     A = Cy(n) × R(n, pred(n)[i])
(20)     B = Cy(pred(n)[i], n)×W(pred(n)[i], n)
(21)     if A/B ∈ ℕ then
(22)        V(e(n, pred(n)[i])) = A
(23)     else
(24)        V(e(n, pred(n)[i])) = LCM(A, B)
(25)        C = V(e(n, pred(n)[i])) / R(n, pred(n)[i])
(26)        if Cy(n)≠C then
(27)           Cy(n) = C
(28)           Modif(Cy(n)) = 1
(29)        end if
(30)     end if
(31)     D = V(e(n, pred(n)[i])) / W(pred(n)[i], n)
(32)     if Cy(pred(n)[i])≠D then
(33)        Cy(pred(n)[i]) = D
(34)        Modif(Cy(pred(n)[i])) = 1
(35)     end if
(36)     if Modif(Cy(n)) = 1 then
(37)        i = 0
(38)        NP = nb(pred(n))
(39)     else if Modif Cy(pred(n)[i])) = 1 then
(40)        i = i
(41)        NP = NP
(42)     else
(43)        i = i + 1
(44)        NP = NP − 1
(45)     end if
(46)  end while
(47)
(48)
(49)  Gc0_Compute(n)
(50)
(51)  for all j do
(52)     Gc0^n[j] = Cy(n) × W(n, succ(n)[j]);
(53)  end for
(54)
```

Algorithm 1: Configuration_granularity().

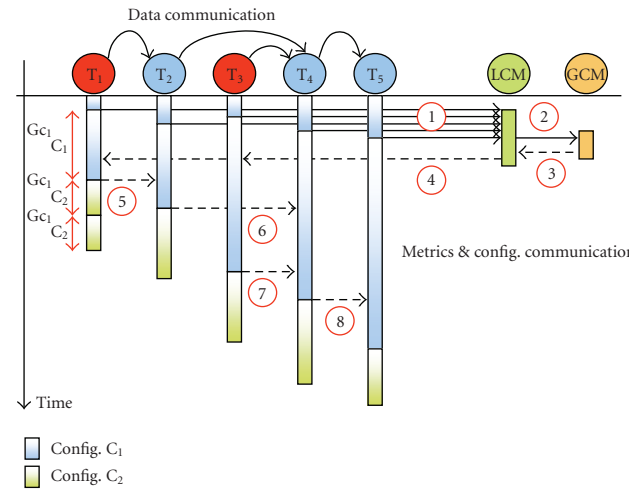

Figure 7: Global configuration management.

Table 1: Communication performance results.

	SW↔ SW	SW↔ HW	HW↔ HW
MB post	517 cy.	2035 cy.	15 cy.
MB pend	425 cy.	3087 cy.	11 cy.

Rule 1. "Data homogeneity": all data computed by the consumer task with a configuration have been produced with the same configuration:

$$\forall n, \quad K^{\mathrm{pred}(n)} \leq K^{n}. \tag{4}$$

Scaling factor K^n from source to sink tasks follows a monotonically decreasing function.

Rule 2. "Computation homogeneity": all data produced with a configuration have to be computed by the consumer task with the same configuration:

$$\forall n, \quad K^{\mathrm{pred}(n)} \geq K^{n}. \tag{5}$$

Scaling factor K^n from source to sink tasks follows a monotonically increasing function.

Rule 3. "Safe reconfiguration rule": applying these two precedent rules means that each task has to respect the following rule:

$$\forall n, \quad K^{\mathrm{pred}(n)} = K^{n} = \mathrm{Max}(K^{n}). \tag{6}$$

In consequence, when all tasks of a cluster use the same scaling factor, data and process homogeneities are ensured and a safe reconfiguration is managed.

3.3.8. Summary

Four cases can be considered. When tasks have no data dependency and there are no application constraints, each task belongs to separate clusters and so has a configuration granularity equal to the initial value of one. Thus, each task can be reconfigured after producing one dataset.

Table 2: Implementation results with a 50 MHz system clock.

(a)

T_1	T_2	T_3	T_4	T_5
Average & Background suppr.	Erosion & Dilation	Reconstruction	Labeling	Display

(b)

T_1-T_2-T_3-T_4-T_5	SW-SW-SW-SW-SW	SW-HW-HW-SW-HW	HW-HW-HW-HW-HW
Exec. Time	245.650.000 cy. ±0,01%	80.800.000 cy. ±0,04%	1.820.000 cy. ±0,2%
Frames/sec.	0,20	0,62	26
Area	19%	59%	92%
StratixII S60 Power	137 mW	228 mW	285 mW

Table 3: Fluctuating execution time due to data.

Task	Variation	Exec. time
Erosion	1 pixel	14.240.816 cy.
	320∗240 pixels	146.197.242 cy.
Reconstruction	1 iteration	15.641.863 cy.
	2 iterations	162.476.520 cy.
	3 iterations	172.675.410 cy.

When tasks have data dependency (they are gathered into a cluster) and no configuration dependency, and there are no application constraints, each task has a configuration granularity equal to the initial value of one. Thus, each task can be configured after one cycle.

When tasks have data dependency and configuration dependency (e.g., the type of data), and there are no application constraints, we compute Gc_0 for each task. Thus, each task can be configured after producing Gc_0 data.

When tasks have data dependency and configuration dependency and there are application constraints (e.g., complete image processing), we compute Gc_0 for each task and rescale it according to constraints. Thus, each task can be configured after producing $K \times Gc_0$ data. K of all tasks is set up according to the third K-setup rule to guarantee data and computation homogeneities.

3.4. Global configuration management

3.4.1. Online configuration process

Figure 7 summarizes the online configuration procedure; upload of metrics (1) from the tasks to the LCM and (2) between local and global configuration managers allows the GCM to select the next configuration. The CID is then downloaded to the LCM (3) which diffuses it by multicasting (4) to the source tasks (T_1 and T_3). Both propagate the message to their downstream tasks as soon as they leave their configuration granularity, and then they take into account the new configuration. The propagation follows the same process gradually (leave the configuration granularity, propagate to its neighbor, and reconfigure it) from the source to the sink tasks (5) and (8). Task T_4 waits the configuration signal from T_2 and T_3 before accepting the new configurations (6) and (7). Otherwise, without a new configuration, tasks continue their process with the current configuration.

3.4.2. Case of nondeterministic data production

In the previous development, the amount of data produced by each task in a given configuration is considered as being fixed; it means that the graph is synchronous in the sense of the SDF formalism. This assumption is no more valid if data production is data-dependent.

However, configuration managers offer the flexibility to solve this issue. In such a case, the amount of produced data is a metric to be sent to the LCM, that can react by sending specific configuration messages to tasks where new configuration granularity is specified. This decision is safe since the LCM can send granularity specification before any reconfiguration order. In practice, the configuration can be recomputed online according to Algorithm 1 or decided according to pre-computed modes.

4. CASE STUDY

Our application is an embedded smart camera for object tracking, as presented in [11], implemented on an Altera Stratix II. It is composed of several tasks which can be implemented in hardware or software. There are two types of tasks. The first one is the set of application tasks for image processing (e.g., averaging, background suppression, erosion, dilation, labeling, etc.). The second one is composed of configuration management tasks (GCM, LCM), sensor and peripheral tasks, which include camera, VGA, and gas gauge controllers. All message passing uses mailbox or message queues

(e.g., configuration or metric message). Image data are transferred through shared memories.

Table 1 shows the communication performances. Overhead of HW ↔ SW communication is due to context switch and control. Otherwise, we reduce drastically the HW to HW communication time. Some architecture configuration may involve an important time overhead (HW and SW tasks alternation). However, in our case study, message passing represents a low percentage of the whole communication.

With different algorithm and architectural configurations, we obtain various tracking system performances as shown in Table 2. We obtain for different architectural and algorithmic configurations a tracking system from 0,2 to 26 frames per second. Execution time results correspond to a tracking process with a standard input frame; so in that case, execution time variations are due to system architecture (e.g., cache miss, bus collision, etc.). However, Table 3 shows some causes of variation at task level due to data characteristics. The reconstruction task is a recursive task depending on object complexity. During each iteration, execution time depends on the number of white pixels. In the same way, erosion and labeling execution times depend on number of white pixels and number of objects, and number of white pixels and object complexity, respectively.

5. CONCLUSION AND PERSPECTIVES

In this paper, we formalize our concept of task configuration management in the context of self-adaptive systems. We propose three contributions implemented as new RTOS services compliant with asynchronous reconfiguration requests. We first define a unified framework for HW/SW task communication and configuration management. We then formalize and provide a systematic solution to compute configuration granularity that guarantees configuration coherency. Finally, we propose a solution to safely control reconfigurations within reconfigurable SOC through propagation principle. Our experience with the image processing application demonstrates the interest of adaptive systems in fluctuating resources requirements according to the environment. We now intend to combine this project with some auto and dynamic reconfiguration experiences already performed on Xilinx FPGA in order to include bitstream dynamic management.

REFERENCES

[1] J. L. Wong, G. Qu, and M. Potkonjak, "An on-line approach for power minimization in qos sensitive systems," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '03)*, pp. 59–64, Kitakyushu, Japan, January 2003.

[2] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 248–259, Haifa, Israel, November 1999.

[3] R. Maro, Y. Bai, and R. I. Bahar, "Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors," in *Proceedings of the 1st International Workshop on Power-Aware Computer Systems (PACS '00)*, p. 97, Cambridge, Mass, USA, November 2001.

[4] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, vol. 1, pp. 986–991, Munich, Germany, March 2003.

[5] H. Walder and M. Platzner, "Reconfigurable hardware operating systems: from design concepts to realizations," in *Proceedings of International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA '03)*, pp. 284–287, Las Vegas, Nev, USA, June 2003.

[6] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[7] D. Andrews, R. Sass, E. Anderson, et al., "The case for high level programming models for reconfigurable computers," in *Proceedings of International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA '06)*, pp. 21–32, Las Vegas, Nev, USA, June 2006.

[8] T. Marescaux, V. Nollet, J.-Y. Mignolet, et al., "Run-time support for heterogeneous multitasking on reconfigurable SoCs," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 107–130, 2004.

[9] V. J. Mooney and D. Blough, "A hardware-software realtime operating system framework for socs," *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 44–51, 2002.

[10] Y. Eustache, J.-P. Diguet, and M. Khodary, "RTOS-based hardware software communications and configuration management in the context of a smart camera," in *Proceedings of International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA '06)*, pp. 84–92, Las Vegas, Nev, USA, June 2006.

[11] J.-P. Diguet, Y. Eustache, and M. Kodhary, "Feedback control learning model for qos, power and performance management of reconfigurable embedded systems," in *Proceedings of the 8th International Symposium on DSP and Communication Systems (DSPCS '05)*, Noosa Heads, Australia, December 2005.

[12] H. Walder and M. Platzner, "Reconfigurable hardware OS prototype," April 2003, ftp://ftp.tik.ee.ethz.ch/pub/people/walder/TIKR168.pdf.

[13] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.