# Architecture for Adaptive Resource Assignment to Virtualized Mixed-Criticality Real-Time Systems

Stefan Groesbrink
Design of Distributed Embedded Systems
Heinz Nixdorf Institute
University of Paderborn
s.groesbrink@upb.de

Simon Oberthür
Design of Distributed Embedded Systems
Heinz Nixdorf Institute
University of Paderborn
oberthuer@upb.de

Daniel Baldin
Design of Distributed Embedded Systems
Heinz Nixdorf Institute
University of Paderborn
dbaldin@upb.de

## ABSTRACT

System virtualization is a powerful approach for the creation of integrated systems, which meet the high functionality and reliability requirements of complex embedded applications. It is in particular well-suited for mixed-criticality systems, since the often applied pessimistic manner of critical system engineering leads to heavily under-utilized resources. Existing static resource management approaches for virtualized systems are inappropriate for the dynamically varying resource requirements of upcoming adaptive systems. In this paper, we propose a dynamic resource management protocol for system virtualization that factors criticality levels in and allows the addition of subsystems at runtime. The two-level architecture offers flexibility across virtual machine borders and has the potential to improve the resource utilization. In addition, it provides the capability to adapt at runtime according to defects or changes of the environment.

## Categories and Subject Descriptors

J.7 [**Computers in other Systems**]: [Real time]; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

## General Terms

Algorithms, Design, Performance, Reliability

## Keywords

System Virtualization, Resource Management, Multicore

## 1. INTRODUCTION

For the next generation of advanced embedded and cyber-physical systems, there is a trend towards adaptability and inclusion of self-optimization [14]. Systems that adjust their goals and behavior at runtime are characterized by varying resource requirements and demand dynamic resource management architectures. Integrated systems can often provide a more resource-efficient implementation compared to multiple separated hardware systems. System virtualization is a promising approach for the integration of multiple systems with maintained separation, and is therefore gaining significant interest in the embedded real-time world [8]. The hypervisor allows the sharing of the underlying hardware among multiple operating systems (OS) in isolated virtual machines (VM).

Virtualization is in particular well-suited for the consolidation of subsystems of differing criticality (e.g. safety-critical subsystems, mission-critical subsystems, and subsystems of minor importance [7, 3]). First, virtualization facilitates multi-OS platforms: adequate operating systems can be provided for the very differing demands of the subsystems, e.g. a highly efficient real-time operating system for safety-critical tasks and a feature-rich general purpose operating system for tasks with a human machine interface. Multiple existing software stacks including operating system can be reused to build a system of systems. Second, the resource utilization can be increased significantly by the addition of subsystems of low criticality to critical subsystems. The dimensioning of the resources of critical systems has to expect the often overestimated worst-case demand at all times, which usually results in a poor utilization.

Virtualization solutions for the server and desktop market apply highly dynamic approaches, e.g. VMware [17], are however not real-time capable. Padala et al. [13] presented an adaptive resource control system that dynamically adjusts the resource shares, however only in order to meet quality of service goals. In order to guarantee real-time requirements, existing virtualization solutions for embedded systems typically assign the resources statically to the VMs. This is not compatible with the dynamics of adaptive systems.

We present a more flexible multi-mode resource management architecture that overcomes these limitations. A survey of real-time mode change protocols can be found in [15], virtualized systems are however not covered. The architecture enables open systems, in which the addition of applications or even subsystems at runtime is possible. This can for example be utilized to allow the users of mobile systems to add applications, with isolated virtual machines ensuring against risks for the critical parts of the system [4].

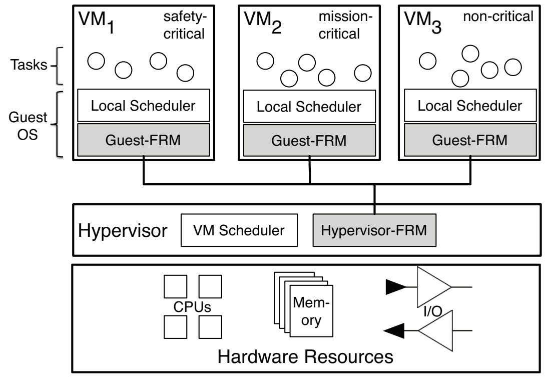


**Figure 1: General Architecture: Hierarchical FRM**

## 2. DYNAMIC RESOURCE MANAGEMENT ACROSS VM BORDERS

### 2.1 The Flexible Resource Manager

We propose a flexible resource management approach for the virtualized execution of mixed-criticality systems. Considered resources are computation time and memory. The main characteristics are a two-level hierarchy as an implication of system virtualization, a mode change protocol, and a real-time conflict resolution.

### 2.2 Two-level Hierarchy

System virtualization requires resource management decisions on two levels. The hypervisor retains the ultimate control of the hardware resources and assigns them to the VMs. The guest OSs on the second level assign the obtained resources to their tasks. Our two-level resource management architecture is depicted in Figure 1. This example includes three virtual machines of different criticality. It may be assumed that the highest criticality level is assigned to safety-critical applications under hard real-time restrictions. The medium criticality level may be assigned to adaptation activities that are responsible for potential corrections of the local system, aligning it to updated global goals. The lowest criticality level may be assigned to some non-mission critical applications, e.g. a routine memory check or a software update.

Core component is the *Flexible Resource Manager (FRM)*. The FRM is an OS extension to improve the resource utilization of self-optimizing real-time systems, which we developed in previous work [9]. Until now, the FRM addressed non-virtualized architectures with a single operating system. In this work, we adapt the concept to system virtualization and FRMs are added on both levels. The *Guest-FRM* is part of the guest OS and the *Hypervisor-FRM* is part of the hypervisor. A partitioned approach with decisions on both levels and communication in both directions follows. The Guest-FRMs inform the Hypervisor-FRM about the dynamic resource requirements and current resource utilization. The Hypervisor-FRM's resource allocation among the VMs is based on this information. The Hypervisor-FRM informs the Guest-FRMs about the assigned resources, which facilitates each Guest-FRM to manage its resource share.

In contrast to static virtualization techniques where the partitions are assigned a priori to the VMs, our approach allows for a dynamic resource allocation even across VM borders. This is achieved by the cooperation of Hypervisor-FRM and Guest-FRMs, made possible by the communication between these components. The mechanism to realize the dynamic resource management is based on a mode change protocol, which is described in the following section.

### 2.3 Protocol Definition

Mode change protocols are highly suitable to support self-optimizing real-time applications [12]. We refer to profiles and transitions between them. A non-empty set of *task profiles* is assigned to each task. Task profiles represent service levels or implementation alternatives, as they exist for example in case of different optimization levels for self-optimizing applications (relax optimality for lower resource utilization). Each profile contains information about minimum and maximum resource requirements and is characterized by a quality. Profiles are specified by the developers of an application. The behavior of a task profile is defined by three functions. The *enter*-function is called on profile activation in order to initialize the profile. The *main*-function is executed periodically as long as the profile is active. In case of a resource share reduction, the *leave*-function is used to release resources in a controlled manner.

The Guest-FRM is in charge of switching between these profiles at runtime. It decides for each point in time and for each task in which of its profiles it runs. A task can only allocate resources in the range that is defined by its active profile. The allocation of more resources requires a preceding profile switch by the Guest-FRM. A profile $P_{\tau_j}$ of task $\tau_j$ is therefore characterized by:

- resource allocation minimums and maximums: $\forall\ resources\ R_k\ with\ limit\ \hat{R}_k :$ $0 \leq \phi_{j,k}^{min} \leq \phi_{j,k}^{max} \leq \hat{R}_k$
- profile quality $Q(\tau_j) \in [0, 1]$
- functions *enter, main, leave*
- subset of the set of task profiles to which the Guest-FRM can switch from $P_{\tau_j}$

In addition to task profiles, there are *VM profiles*, which specify the minimal and maximal resource limits of a VM. VM profiles unite the active profiles of the VM's tasks. In case of a task profile transition, the VM profile is updated and communicated to the Hypervisor-FRM and used for the resource assignment among the virtual machines. A VM profile $P_{VM_i}$ is defined as follows:

- resource allocation minimums and maximums: $\forall\ resources\ R_k : \forall\ tasks\ j\ of\ VM_i :$ $\Phi_{i,k}^{min} = \sum_j \phi_{j,k}^{min}, \quad \Phi_{i,k}^{max} = \sum_j \phi_{j,k}^{max}$
- VM criticality $\chi(VM_i) \in \mathbb{N}^+$
- profile quality $Q(VM_i) \in \mathbb{N} : Q(VM_i) = \sum_j Q(\tau_j)$
- subset of the set of VM profiles to which the Hypervisor-FRM can switch from $P_{VM_i}$

A criticality level $\chi$ is assigned to each VM. A value of one is set if the system is non-critical and a larger value denotes higher criticality. The assignment of criticality on VM level is no restriction, since our use case deals with the consolidation of entire systems.

The set of active profiles is called *configuration*. The possibility to switch between profiles on task level and on VM level enables a dynamic resource assignment on task level across VM borders. A Guest-FRM can shift resources from task $\tau_i$ to another task $\tau_j$ by activating a profile with a lower resource allocation maximum for $\tau_i$ and activating a profile with a higher resource allocation maximum for $\tau_j$. Similarly, due to the cooperation of the FRMs on the two levels, resources can be reallocated from task $\tau_i$ of $VM_1$ to task $\tau_k$ of $VM_2$. The Hypervisor-FRM activates a VM profile with a lower resource allocation maximum for $VM_1$ and according to this, the Guest-FRM of $VM_1$ activates a task profile with a lower resource allocation for $\tau_i$. This allows the Hypervisor-FRM to activate a VM profile with a higher resource allocation maximum for $VM_2$ and the Guest-level FRM of $VM_2$ to activate a task profile with a higher resource allocation for $\tau_k$.

Common practice for real-time systems is the static allocation of the maximal required resources to each task. This results in a poor utilization and lack of adaptability. Instead, the FRM tries to assign fractions of these resources at runtime to other tasks whenever a task does not use the complete amount of resources as needed in the worst case. This is realized by the introduced profile switching mechanism. If resources were reallocated from a task to another task and the resource lending task at a later point in time needs more resources than left, a *resource conflict* occurs. Resource conflicts have to be solved under real-time constraints.

## 2.4 Real-time Conflict Resolution

We have to distinguish between two kinds of resource conflicts, caused by two kinds of dynamic resource reallocation. The Guest-FRMs can reallocate resources among their tasks and the Hypervisor-FRM can reallocate resources among VMs. In both cases, an acceptance test precedes and a resource reallocation is accepted if and only if:

- $\forall\ Resources\ R_k\ ,\ \forall\ tasks\ 1..n : \sum_{i=1}^{n} \phi_{i,k}^{max} \leq \hat{R_k}$
- the FRM identifies a feasible *reconfiguration*

A reconfiguration is a set of profile switches that can be performed by the FRMs to activate a configuration, which fulfills the worst-case requirements of all tasks. The FRM uses a *configuration reachability graph* with a node for each configuration and edges that denote a possible configuration switch. The duration to perform the configuration is assigned to each edge. It is calculated by the sum of the WCETs of the enter/leave functions that have to be executed to realize the associated profile switches. If this reconfiguration plan includes VM profile switches, and by consequence profile switches by Hypervisor-FRM and at least two Guest-FRMs, it is called *global reconfiguration*. In contrast,

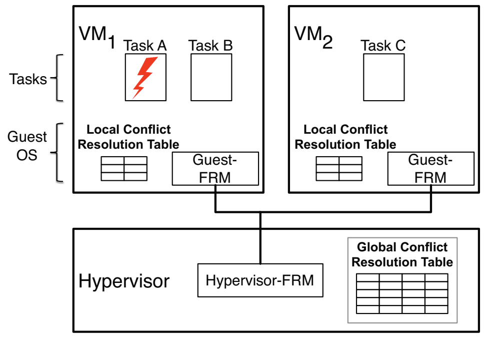

VM1
Task A
Task B
Tasks
Guest OS
Local Conflict Resolution Table
Guest-FRM
VM2
Task C
Local Conflict Resolution Table
Guest-FRM
Hypervisor
Hypervisor-FRM
Global Conflict Resolution Table

**Figure 2: Conflict Resolution**

a *local reconfiguration* includes only task profile switches and is accomplished by a single Guest-FRM. Actually, it is not sufficient to identify such a reconfiguration plan. The schedulability analysis has to check whether the time required to execute the reconfiguration does not lead to a deadline miss. At the moment, our approach includes only Earliest Deadline First Scheduling (EDF) [11]. EDF is characterized by a simple schedulability test (utilization $\leq$ 1). The aperiodic reconfiguration is integrated into the periodic schedule by a Total Bandwidth Server [16].

The reconfiguration plans for conflict resolution are stored in *conflict resolution tables*. An entry is created after a reconfiguration was accepted and lists the profile switches that have to be performed to reach a state that guarantees all deadlines. If a conflict is always solved by reconfiguration to the initial state, there is at most one entry per task profile. A larger table with the possibility to reconfigure to multiple optimization levels is more promising, but requires additional memory. The conflict resolution protocol is explained using the example of Figure 2.

**Example.** It is assumed that task $A$, executed in virtual machine $VM_1$, has a specific worst-case requirement of a resource and consequently, this resource share was assigned. Since the actual resource usage of task $A$ was significantly below the reserved amount, the Guest-FRM switched to another profile and made a fraction of the assigned resources available to task $B$ of the same VM. In case of a resource conflict, i.e. task $A$ requires a larger resource share than left, the Guest-FRM resolves the conflict by switching to task profiles with a resource distribution that fulfills the timing requirements of task $A$. The sequence of profile switches that have to be performed to obtain this state was stored in $VM_1$'s local conflict resolution table when the acceptance test for the resource reallocation was passed.

It is possible that the Guest-FRM cannot resolve the resource conflict, since a global reconfiguration is required to achieve this. This is the case, if it was caused by a resource reallocation to another guest system. A share of the resource reserved for virtual machine $VM_1$ could have been assigned to virtual machine $VM_2$ by the Hypervisor-FRM, and further assigned by $VM_2$'s Guest-FRM to task $C$. The Hypervisor-FRM informed $VM_1$'s Guest-FRM about this resource reallocation and the Guest-FRM noted this in the local conflict resolution table. The conflict resolution is depicted in the UML sequence diagram of Figure 3. In case of a resource conflict of task $A$, the Guest-FRM of $VM_1$ informs the Hypervisor-FRM, which prompts the Guest-FRM of $VM_2$ to release the supplemental resources. The Guest-FRM of $VM_2$ switches the profile of task $C$ to one of lower resource utilization. The Hypervisor-FRM can accordingly switch the profile of both $VM_1$ and $VM_2$ and inform the Guest-FRM of $VM_1$ to ultimately activate the conflict resolving profile switch for task $A$.

## 2.5 Reconfiguration Management

The FRMs are in charge of determining the resource allocations at runtime by selecting the active profiles. Profile switches for optimization reasons are only induced by the Hypervisor-FRM, which has information about the current resource requirements of the entire system. As introduced in the last section, the Guest-FRMs invoke local reconfigurations in order to solve a conflict, however, they do not take decisions on their own to improve the quality. A cooperation of Hypervisor-FRM and Guest-FRM is nevertheless required, since profile switches on task level ultimately realize the resource reallocation. The Hypervisor-FRM informs the Guest-FRMs, which then invoke the task profile switches.

A heuristic approach is applied for taking these decisions, comparable to [9]. Whether a reconfiguration has positive effects or not, depends on the ratio between overhead of the reconfiguration process to benefit, and on the duration after which the system has to be re-reconfigured. The duration depends on various stochastic events which are caused by both the applications themselves and the environment. The FRM permanently gathers information about the resource utilization and computes with the help of *Dynamic Bayesian Networks* a likeliness of change of the resource requirements. The probabilistic analysis produces a mean time of resource requirement changes, which is used as a forecast.

The criticality influences the resource distribution significantly. In general, the higher the criticality level, the lower the expected resource utilization. The initial resource allocation assigns the worst-case demand to critical VMs. A resource reallocation for optimization reasons takes therefore place from VMs of higher criticality to VMs of lower criticality. If multiple reconfigurations can be performed, those that favor the situation of a VM of higher criticality receive always a significant better assessment. The profile qualities are the tie-breaker parameter if multiple reconfigurations on one criticality level can be performed. It should be mentioned that the probabilistic techniques affect only the (secondary) objective to achieve (near) optimal usage of resources. The primary critical real-time objectives remain guaranteed in a fully predictable manner.

## 2.6 Scheduling

The most important resource is the central processing unit. Our approach targets homogeneous multiprocessor systems. System virtualization requires scheduling decisions on two levels (*hierarchical scheduling* [10]): both the virtual machines and the tasks within the VMs have to be scheduled. There are two main approaches in the multiprocessor scheduling domain. As opposed to *partitioned scheduling*, migration of tasks among processors is possible under *global scheduling* [6].

Partitioned scheduling is for multiple reasons the right approach for system virtualization. It introduces in general a lower overhead, since the task migration of global scheduling results in overhead for synchronization and lost performance due to cache misses [6]. Furthermore, the consolidation by system virtualization runs entire software stacks consisting of guest operating system and tasks within a virtual machine. It is therefore a coarse-grained approach to reuse systems with verified (or even certified) characteristics, which should not be split up. This is especially true for mixed-criticality systems, since a mixing of criticality levels within a subsystem should be avoided. Task migration is therefore neither desirable nor in general technically possible across operating system borders.

Our first approach allocates each safety-critical guest and mission-critical guest to a dedicated processor. A processor is not shared between multiple critical guests ($\chi > 1$), but an addition of non-critical guests ($\chi = 1$) is possible. If critical and non-critical guests share a processor, the non-critical guests are scheduled in background: whenever the critical guest is not running, the idle processor is used to execute the non-critical guest. Consequently, the execution of a system with $n$ safety-critical or mission-critical guests requires a hardware platform with at least $n$ processors. Contrary to task migration, the migration of an entire virtual machine from one processor to another is less problematic and in some situations of significant help, as explicated regarding *Open Systems* in the following subsection. To keep the introduced mapping of critical guests to processors, only non-critical guests are possibly migrated.

## 2.7 Open Systems

Open systems require dynamic scheduling algorithms and an online acceptance test which validates the schedulability whenever a new task enters the system [5]. The acceptance test checks whether it is possible to add the arriving task to the set of previously guaranteed tasks or not. Our approach enables the addition of both entire guest systems and tasks at runtime. Based on the fact that the FRM requires the scheduling algorithm EDF, under which the processor may be utilized up to 100% [11], a request to add a new real-time task $\tau_k$ with a worst-case execution time of $C_k$ and a period of $T_k$ to the existing task set of the subsystem of $n$ tasks can be accepted if and only if

$$\frac{C_k}{T_k} + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

The much more complex acceptance test for the addition of an entire new guest system is depicted in Figure 4.

(1) If there is a processor that is unused up to now, the new guest system is assigned to this processor and the request is accepted.

(2) If a non-critical guest system shall be added, it is assigned to the processor with the lowest utilization. The addition of a non-critical guest system does not endanger the

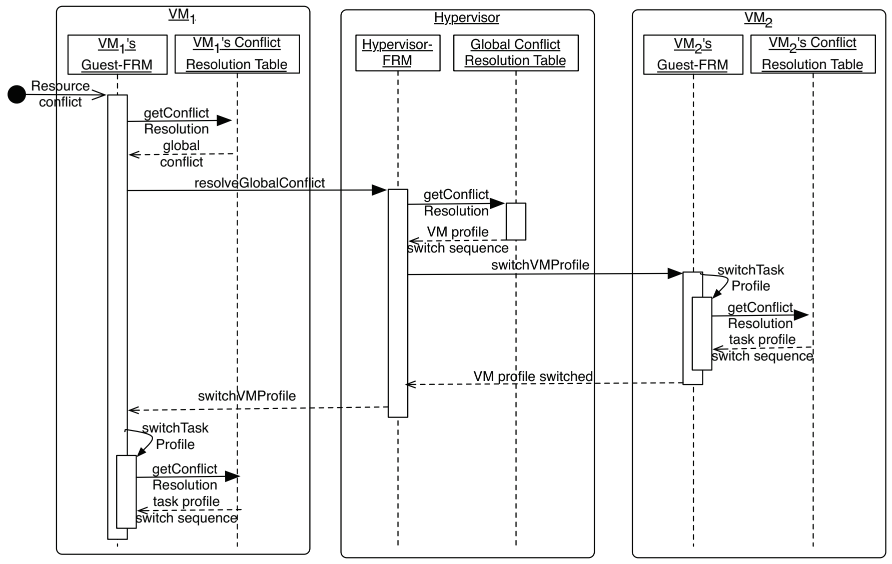


**Figure 3: Conflict Resolution: Global Reconfiguration Sequence**

real-time tasks, since it is scheduled in background.

(3) If there is not a critical guest assigned to each processor, the arriving guest system is assigned to the processor with the lowest utilization among the processors without critical guest. The schedulability of the new guest system is guaranteed, since it shares the processor only with non-critical guests and obtains therefore highest priority. It is possible that the addition of the new guest implies an unbalanced distribution of guest systems to processors. Therefore, non-critical guests are possibly migrated.

(4) If a critical guest is assigned to each processor, but there is at least one guest of lower criticality level, the arriving guest system replaces the critical guest of lowest criticality. This could again lead to an unbalanced distribution among the processors, for which reason non-critical guests are migrated, if this improves the distribution.

If none of the checked conditions was valid, the arriving guest system has to be rejected. Concerning step (4), it is debatable whether a higher criticality level legitimates automatically the replacement of an entire guest system. It depends on the application domain and may be unjustified and in this case has to be deactivated to favor system continuity. But there are definitely situations in which this replacement makes sense, for example when a mission-critical subsystem is replaced by a safety-critical subsystem to protect device or environment at the expense of an unsuccessful mission.

The Hypervisor-FRM performs the schedulability test for subsystems. The FRM approach requires that both arriving tasks and arriving subsystems provide profiles, if they have to be scheduled under real-time constraints. Non-real-time applications and subsystems can be accepted without profiles.

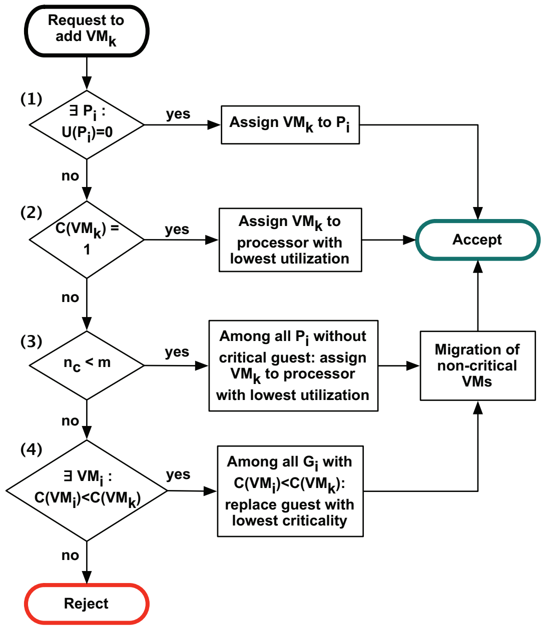
## 3. CONCLUSION

The presented resource management architecture focuses on three basic concepts. *System virtualization* and its ability to reuse subsystems is a powerful technique to meet the functionality and reliability requirements of increasingly complex systems and has potential to support the migration to multiprocessor platforms. *Mixed-criticality systems* and virtualization's integration of both multi-source software and subsystems with very differing characteristics are a good fit, and resource sharing is particularly promising for this combination. *Open systems* increase the flexibility enormously and are of paramount importance for upcoming adaptive embedded and cyber-physical systems.

By combining these three concepts, our approach provides a flexible resource management architecture for the dynamically varying resource requirements of integrated adaptive systems. The two-level solution beyond virtual machine borders has the potential to increase the resource utilization significantly compared to static approaches. Multiple criticality levels are handled appropriately to achieve two goals: the deterministic behavior of critical missions is guaranteed and the service quality of non-critical missions is possibly increased by shifting resources to them. The resource assignment can adapt to chances in application behavior and environment and the addition of tasks and entire subsystems at runtime is possible.

Subsystems can temporarily use resources that were reserved for more critical subsystems, if those can be taken away again in real-time in case of worst-case scenarios of critical parts. The service quality of non-safety-critical subsystems can be dynamically reduced in case of worst-case behavior or overload scenarios of critical parts, if this is required to protect the correct execution of critical tasks.

The approach is currently under development. We integrate



Request to add VM$_k$
(1)
∃ P$_i$ : U(P$_i$)=0
yes
Assign VM$_k$ to P$_i$
no
(2)
C(VM$_k$) = 1
yes
Assign VM$_k$ to processor with lowest utilization
Accept
no
(3)
n$_c$ < m
yes
Among all P$_i$ without critical guest: assign VM$_k$ to processor with lowest utilization
Migration of non-critical VMs
no
(4)
∃ VM$_i$ : C(VM$_i$)<C(VM$_k$)
yes
Among all G$_i$ with C(VM$_i$)<C(VM$_k$): replace guest with lowest criticality
no
Reject

**Figure 4: Acceptance Test for Guest System (virtual machine $VM_k$, processors $P_1$ to $P_m$, utilization $U$, criticality $C$ (instead of $\chi$), number of critical guests $n_C$)**

it into our real-time hypervisor *Proteus* [1]. The implementation of our approach requires paravirtualization, since the guest-level FRMs have to pass information to the hypervisor. According to paravirtualization [2], modified guest operating systems that are able to communicate with the hypervisor are hosted. The requirement to modify the guest operating system is outweighed by the advantages gained in terms of flexibility of an explicit communication and cooperation of host and guest.

## Acknowledgment

This work was funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035. The responsibility for the content remains with the authors.

## 4. REFERENCES

[1] D. Baldin and T. Kerstan. Proteus, a hybrid virtualization platform for embedded systems. In *3rd IFIP TC 10 International Embedded Systems Symposium*, 2009.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[3] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2010.

[4] J. Brakensiek, A. Droege, M. Botteck, H. Haertig, and A. Lackorzynski. Virtualization as an enabler for security in mobile devices. In *1st Workshop on Isolation and Integration in Embedded Systems*, 2008.

[5] G. Buttazzo. *Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[6] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. In *ACM Computing Surveys*, 2010.

[7] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *30th IEEE Real-Time Systems Symposium*, 2009.

[8] G. Heiser. The role of virtualization in embedded systems. In *1st Workshop on Isolation and Integration in Embedded Systems*, 2008.

[9] H. S. Lichte and S. Oberthür. Schedulability Criteria and Analysis for Dynamic and Flexible Resource Management. *Electron. Notes Theor. Comput. Sci.*, 200(2):3–19, 2008.

[10] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. In *Journal of Embedded Computing*, volume 1(2), 2005.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM*, 1973.

[12] S. Oberthür, L. Zaramba, and H.-S. Lichte. Flexible resource management for self-x systems: An evaluation. In *Proc. of 1st IEEE Workshop on Self-Organizing Real-Time Systems - SORT*, 2010.

[13] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc. of the EuroSys*, 2007.

[14] S. Pook, J. Gausemeier, and R. Dorociak. Securing the reliability of tomorrow's systems with self-optimization. In *Proceedings of The Annual Reliability and Maintainability Symposium*, 2012.

[15] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. In *Real-Time Systems*, volume 26(2), 2004.

[16] M. Spuri and G. C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. In *Real-Time Systems*, volume 10(2), pages 179–210, 1996.

[17] VMware, Inc.. Resource Management with VMware DRS. VMWare Whitepaper, 2006.