



HAL
open science

Embedded Real-Time Virtualization Technology for Reconfigurable Platforms

Tian Xia

► **To cite this version:**

Tian Xia. Embedded Real-Time Virtualization Technology for Reconfigurable Platforms. Embedded Systems. INSA Rennes; Université Bretagne Loire, 2016. English. NNT: 2016ISAR0009. tel-01418453v1

HAL Id: tel-01418453

<https://hal.science/tel-01418453v1>

Submitted on 26 Oct 2018 (v1), last revised 16 Dec 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE
BRETAGNE
LOIRE

THESE INSA Rennes
sous le sceau de Université Bretagne Loire
pour obtenir le titre de

DOCTEUR DE L'INSA DE RENNES
Spécialité : Electronique et Télécommunications

présentée par
Tian Xia

ECOLE DOCTORALE : MATISSE
LABORATOIRE : IETR

Embedded Real-Time Virtualization Technology for Reconfigurable Platforms

Thèse soutenue le 05.07.2016
devant le jury composé de :

François Verdier

Professeur, Université de Nice Sophia-Antipolis, Nice / *président*

Emmanuel Grolleau

Professeur, ISAE-ENSMA, Chasseneuil-Futuroscope / *Rapporteur*

Guy Gogniat

Professeur, Université de Bretagne-Sud, Lorient / *Rapporteur*

Jean-Luc Bechenec

Chargé de Recherche, Ecole Centrale de Nantes, Nantes / *Examineur*

Jean-Christophe Prévotet

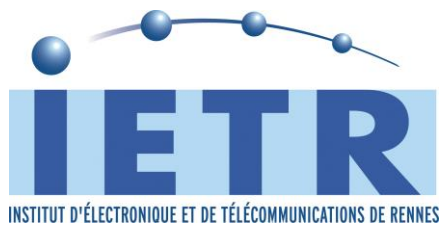
Maître de Conférence, INSA, Rennes / *Co-encadrant de thèse*

Fabienne Nouvel

Maître de Conférence HDR, INSA, Rennes / *Directrice de thèse*

Embedded Real-Time Virtualization Technology for Reconfigurable Platforms

Tian Xia



To my parents, and my beloved wife.

ACKNOWLEDGEMENT

First of all, I would like to express my great gratitude to my advisors, Fabienne Nouvel and Jean-Christophe Prévotet, who are excellent scholars. During my 4 years' stay in Rennes, you have given me countless precious advices and kind helps for both my studies and my life, and have always been supportive for my research. None of the work in this thesis could be done without your kindness and guidance.

I would like to sincerely thank all the members of my committee, Professor Emmanuel Grolleau, Professor Guy Gogniat, Professor François Verdier and Doctor Jean-Luc Bechenec, for their valuable suggestions and advices on my research. It is such an honor of mine that you were present in my committee and attended my defense. I am also thankful for Professor Mei Kuizhi, for reviewing my thesis, and being present at my defense.

I would like to thank the China Scholarship Council and the INSA-UT CSC project for giving me the opportunity to study at Rennes.

I would also like to express my thanks and appreciation to all my colleagues at the IETR laboratory : Yaset Oliva, Yvan Kokar, Jordane Lorandel, Medhi Aichouch, Mohamad-Al-Fadl Rihani, Hiba Bawab, Papa Moussa Ndao, Roua Youssef, Samar Sindian, Ali Cheaito, Hussein Kudoh, Rida El Chall, Mohamed Maaz, and all the others whose name I forgot to mention. Thank you all for your kindness, and for the beautiful memories we have shared in our lovely office. I am also very grateful for Aurore Gouin, Pascal Richard, Katell Kervella, Christèle Delaunay, Yolande Sambin, Jérémy Dossin and Laurent Guillaume, for their kind helps and professional service.

I would like to thank my friends at Rennes : Sun Lu, Ge Di, Luo Tao, Chu Xingrong, Bai Xiao, Wang Yu, Bai Cong, Ji Hui, Si Haifang, Zou Wenbin, Wang Hongquan, Zhang Jinglin, Zhang Xunying, Yi Xiaohui, Lu Weizhi, Li Weiyu, Fan Xiao, Yao Dandan, Liu Wei, Liu Yi, Tang Liang, Li Bo, Shuai Wenjing, Wang Cheng, Yuan Han, Yao Zhigang, Gu Qingyuan, Yang Yang, Zhao Yu, Wang Duo, Fu Jia, Wu Xiguang, Wang Yanping, Xu Jiali, Zhang Xu, Huang Gang, Liu Shibo, Chen Zhaoxing, Wang Shijian, Song Xiao, Huang Yong, Lu Hua, Wei Hengyang, Jiao Wenting, Fan Jianhua, Yao Haiyun, Wang Qiong and Tian Shishun. The time we spent together was full of laughter and joy. Thank you all for these wonderful and unforgettable days we had.

I would like to express special thanks to my friends : Fu Hua, Liu Ming and Peng Linning, for your daily company, your friendship and your heart-warming helps. Thanks to you, my days at Rennes were filled with wonderful experiences and colorful stories

that I will always cherish in my heart.

Foremost, I am deeply grateful to my dear parents Fanglin and Liguang, and my beloved wife, Yicong, for their generous and selfless support, understanding and encouragement. Life is never easy, but somehow you make my life the best I can ever imagine. Thank you and I love you all.

In the end, I want to take a minute to mourn my dear aunt, who passed away during the writing of this manuscript. I love you and I really miss you.

Résumé de la Thèse

Aujourd'hui, les systèmes embarqués jouent un rôle prépondérant dans la vie quotidienne des utilisateurs. Ces systèmes sont très hétérogènes et regroupent une énorme diversité de produits tels que les smartphones, les dispositifs de contrôle, les systèmes communicants, etc. Couvrant une large gamme d'applications, ces systèmes ont évolué en différentes catégories. Il existe des systèmes avec une grande puissance de calcul pouvant mettre en œuvre des logiciels complexes et permettre la gestion de ressources complexes. D'autres systèmes embarqués à faible coût, avec des ressources limitées, sont destinés à l'implantation de dispositifs simples, tel que ceux mis en œuvre dans l'Internet des Objets (IdO). Fondamentalement, la plupart de ces appareils partagent des caractéristiques communes telles que la taille, le poids et la faible consommation d'énergie.

Tandis que la complexité des systèmes embarqués augmente, il devient de plus en plus coûteux d'améliorer les performances des processeurs par des approches technologiques classiques comme la diminution de la taille des transistors, par exemple. Dans ce contexte, l'idée d'une architecture hétérogène CPU-FPGA est devenue une solution prometteuse pour les concepteurs de systèmes sur puce. D'autre part, la forte capacité d'adaptation et son faible coût en font une solution très prisée. Cette solution permet de faire bénéficier aux architectures matérielles classiques des avantages et de la flexibilité d'un processeur. Elle permet également d'étendre les concepts logiciels classiques, tels que la virtualisation, aux circuits matériels.

Dans cette thèse, notre recherche apporte une contribution dans ce domaine, en étudiant la virtualisation en temps réel des systèmes embarqués mettant en œuvre la reconfiguration partielle (DPR) de circuits, et ce, dynamiquement. Les objets cibles de la thèse sont les systèmes embarqués hétérogènes comportant au moins un processeur et une architecture reconfigurable de type FPGA.

Dans ce type d'architectures, les principales limitations sont les ressources de calculs restreintes, la faible quantité de mémoire, et des contraintes d'exécution temps réel. Nos travaux se concentrent sur deux aspects en : 1) proposant un micro-noyau léger, personnalisé, hautement adaptable, nommé Ker-ONE, qui prend en charge la virtualisation en temps réel, et 2) en proposant un mécanisme de coordination innovant des accélérateurs reconfigurables entre plusieurs machines virtuelles.

1. Concepts et état de l'art

1.1 Virtualization des systèmes embarqués

La virtualisation sur les systèmes embarqués nécessite de répondre à plusieurs contraintes spécifiques. Tout d'abord, la sécurité doit être garantie puisque les appareils embarqués peuvent contenir des informations personnelles sensibles telles que les numéros de téléphone, des informations bancaires, des clés privées, etc. Une machine virtuelle doit être protégée et ne plus dépendre d'autres machines. Au cas où une machine virtuelle viendrait à être défaillante, elle ne doit en aucun cas corrompre le système dans sa globalité. Deuxièmement, les processus temps réel sont indispensables afin d'effectuer des tâches critiques (comme la mise en oeuvre des systèmes d'airbags, la gestion des systèmes d'assistance au freinage des véhicules, ou le traitement des appels d'urgence dans les téléphones mobiles). Troisièmement, en raison de la limitation des ressources des systèmes embarqués, des systèmes virtuels doivent être conçus avec peu de complexité, ce qui peut être obtenu en utilisant un hyperviseur léger.

Dans les dispositifs embarqués qui sont dédiés à l'Internet des objets, les contraintes de conception sont encore plus grandes. Ces contraintes portent principalement sur le manque de ressources de calculs et sur la consommation d'énergie : les appareils sont généralement très simples, basés sur un seul micro-contrôleur ou un petit processeur intégré dans un FPGA avec une quantité limitée de mémoire et d'E/S. Ces derniers sont généralement alimentés par des batteries, ce qui impose une très faible consommation d'énergie associée à une gestion efficace de cette même énergie. Dans un tel scénario, une faible complexité du logiciel devient incontournable, de manière à implanter efficacement les machines virtuelles.

Certaines architectures x86 traditionnelles (par exemple Intel VT, AMD-V, etc.) comportent des extensions de virtualisation matérielle, permettant de "piéger" et "émuler" toutes les instructions sensibles. Les OS invités peuvent ainsi être hébergés sans modification. Toutefois, la couche ISA d'un processeur ARM traditionnel ne convient pas à la virtualisation. L'architecture ARMv7 offre deux niveaux de privilège : PL0 et PL1. Si l'on peut considérer que PL0 peut accéder librement à des ressources globales du processeur, les ressources vitales ne sont accessibles que dans les modes de PL1, ce qui garantit la sécurité du code privilégié.

En général, un noyau d'OS fonctionne dans les modes de PL1, et gère toutes les ressources et les tâches. Les applications utilisateur s'exécutent dans le mode PL0, comme le montre la figure [FIGURE 1\(a\)](#). Cependant, en para-virtualisation, les systèmes d'exploitation invités devraient être portés à un niveau non-privilégié alors que le niveau privilégié est occupé par l'hyperviseur (ou VMM pour Virtual Machine Monitor). Dans ce cas, le code source d'un système d'exploitation client doit être modifié pour fonctionner correctement en mode utilisateur et les instructions sensibles doivent être remplacées par des hyper-calls, comme le montre la figure [FIGURE 1\(b\)](#).

Récemment, ARM a inclus des extensions matérielles pour permettre une virtualisation complète. Ces extensions offrent un mode supplémentaire d'exécution, le mode hyperviseur (HYP), pour les noyaux de virtualisation les plus privilégiés. Cette prise en

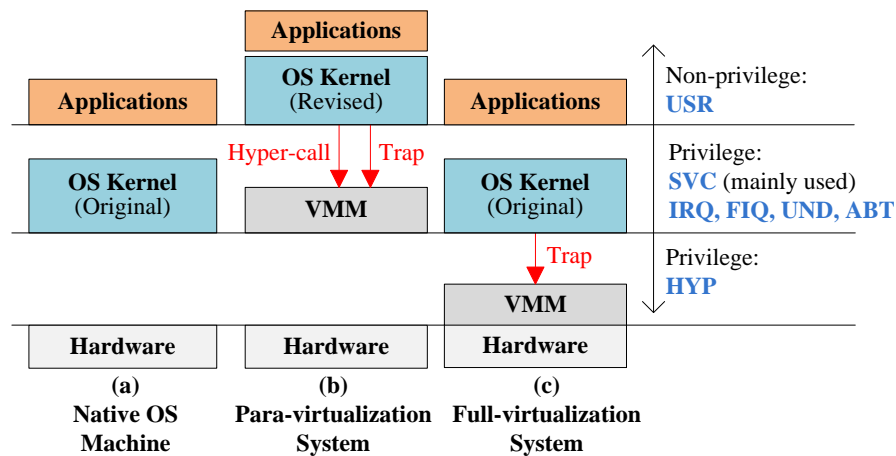


FIGURE 1 – Systèmes de virtualisation complète et de para-virtualisation sur une architecture ARM.

charge n'a été introduite que récemment (en 2010) pour le processeur Cortex-A15. Ce mécanisme complet de virtualisation est décrit dans la figure FIGURE 1(c).

Un autre défi dans la virtualisation intégrée de systèmes est une compatibilité avec les contraintes temps réel, ce qui signifie que le respect des latences des tâches doit être garanti en temps réel. Dans la littérature, l'ordonnabilité en temps réel a déjà été étudiée et correspond souvent à un ordonnancement hiérarchique. Nous définissons respectivement les concepts d'inter-VM et de programmation intra-VM pour l'ordonnancement des machines virtuelles, et l'ordonnancement des tâches locales au sein d'une même machine virtuelle. Puisque les OS invités partagent la même ressource processeur, l'ordonnabilité d'un jeu de tâches est non seulement déterminée par l'algorithme d'ordonnancement d'éployé au sein d'une machine virtuelle, mais aussi par l'algorithme d'ordonnancement des machines virtuelles. Par conséquent, afin de respecter les contraintes d'un OS temps réel, le respect des échéances des tâches temps réel doit être également garanti dans l'hyperviseur.

1.2 État de l'art sur la Virtualisation des architectures ARM

Dans les approches classiques de para-virtualisation, un système d'exploitation invité est normalement fourni avec un patch de virtualisation pour interagir avec l'hyperviseur, ce qui exige que le code source du système d'exploitation soit disponible. En conséquence, la majorité des OS actuellement pris en charge se résume à quelques OS libres, largement distribués, comme Linux embarqué et $\mu C/OS-II$.

Une autre solution envisageable permettant d'implanter la para-virtualisation consiste à utiliser un micro-noyau, qui, comme son nom l'indique est un petit noyau avec des ressources essentielles telles qu'un espace d'adressage, des mécanismes de communications inter-processus, etc. .

Toutes les autres fonctionnalités additionnelles sont normalement mises en œuvre au

niveau utilisateur. Les micro-noyaux de type L4 s'articulent généralement autour de Linux pour porter les machines virtuelles. Une solution prometteuse est microvisor de OKL4 d'Open Kernel Labs, qui a amélioré considérablement les noyaux L4 classiques. Cependant, le fait que OKL4 ne soit pas libre rend impossible l'acquisition de détails de mise en œuvre ainsi que de proposer d'autres études sur ce système.

Une autre solution disponible autour de la para-virtualisation consiste à utiliser de grands hyperviseurs monolithiques, dont la plupart sont d'abord conçus pour la virtualisation des architectures x86, tels que KVM et Xen. Kernel-based Virtual Machine (KVM), par exemple, a été revisité afin de fonctionner dans le ARMv5 et a été proposé en tant que KVM pour ARM (KVM/ARM). Il se compose d'un module de noyau léger qui doit être inclus dans le noyau Linux, et repose sur une version de QEMU modifiée pour mettre en œuvre l'émulation de l'hyperviseur. Les hyperviseurs de type XEN ont également été mis en œuvre sur les architectures ARM, comme Xen-on-ARM.

L'inconvénient des hyperviseurs basés sur KVM ou Xen est leur dépendance vis à vis de Linux en tant que système d'exploitation hôte, ce qui augmente considérablement la taille du code du système global.

L'ajout d'un support temps-réel pour la virtualisation des systèmes a été l'objet de nombreuses études. Un modèle type de planification en temps réel est le *Compositional Real-time Scheduling Framework* (CSF) basée sur une planification hiérarchique des tâches. Cet algorithme utilise le *modèle de ressource périodique* (PRM) afin de faire abstraction d'un groupe d'applications temps réel dans un modèle de charge de travail qui peut être considéré comme une seule tâche périodique. Sur la base de ce modèle de tâches périodiques, l'ordonnancement de machines virtuelles peut être analysé directement en utilisant des algorithmes de programmation classiques. Le micro-noyau L4 Fiasco a revisité le serveur L4Linux pour intégrer ce modèle dans la programmation en temps réel. RT-Xen a également été proposé afin d'étendre l'hyperviseur XEN et profiter ainsi de l'ordonnancement temps réel hiérarchique.

Un autre problème concernant la mise en œuvre de tels systèmes est le surcoût de latence provoqué par les interruptions du service d'horloge. Ce problème a été résolu dans [YY14]. Les auteurs ont proposé un nouvel algorithme d'ordonnancement pour XEN, notée SHQuantization. Ce dernier permet de prendre en compte les surcoûts temporels. Cependant, cette technologie nécessite la modification de l'interface de programmation du système d'exploitation invité. Dans cette approche, l'ordonnanceur de l'OS invité dépend également fortement de l'hyperviseur.

1.3 Accélérateur reconfigurable sur plate-forme CPU-FPGA hétérogène

Dans le domaine universitaire, le thème des systèmes hétérogènes basés sur le couple CPU-FPGA a été massivement étudié pour fournir des dispositifs FPGA reconfigurables actuels mettant en œuvre un système d'exploitation. Une approche réussie dans ce domaine est ReconOS, qui est basé sur un RTOS open-source (eCos) prenant en charge les tâches matérielles/logicielles multithread. ReconOS fournit une solution classique pour la gestion des accélérateurs matériels dans un système hybride et dans un modèle de

gestion classique de threads. Cependant, la possibilité de tirer profit de la virtualisation n'a pas été pleinement discutée dans ces travaux.

Une attention particulière a également été portée sur des accélérateurs FPGA partagés dans un contexte multi-clients, par exemple des OS complexes, des serveurs cloud ou des systèmes de machines virtuelles. Dans ces recherches, une technologie de virtualisation DPR est proposée. Cette technologie offre des accès efficaces à des accélérateurs virtuels et simplifie considérablement le développement logiciel.

Dans [HH09], ce concept est mis en œuvre à l'aide de OS4RS sous Linux. Le matériel virtuel permet aux périphériques matériels et aux ressources logiques d'être partagés simultanément entre les différentes applications logicielles. Cependant, cette approche est proposée dans le cadre d'un seul système d'exploitation, sans tenir compte des fonctionnalités de virtualisation. Une autre étude est présentée dans [WBP13]. Les auteurs tentent d'étendre l'hyperviseur Xen afin de partager un accélérateur FPGA entre les machines virtuelles. Cependant, cette recherche porte uniquement sur la méthode de transfert de données entre le CPU et le FPGA, et n'inclut pas la technologie de reconfiguration dynamique partielle.

La virtualisation des systèmes reconfigurables est beaucoup plus populaire sur les serveurs cloud et les centres de données, qui ont généralement un besoin plus fort en termes de performance et de flexibilité. Par exemple, dans des travaux tels que [BSB⁺14] et [KS15], les auteurs utilisent la reconfiguration partielle pour diviser un seul FPGA en plusieurs régions reconfigurables, dont chacune est gérée comme un seul FPGA virtualisé (VFR). Ce type de virtualisation n'est néanmoins pas approprié pour les systèmes embarqués, dont les ressources disponibles sont très limitées, comparées à celles qui sont disponibles dans les serveurs ou les centres de données.

Dans notre travail, la plate-forme cible est le SoC Xilinx Zynq-7000 comprenant un processeur Cortex-A9 dual-core et un FPGA reconfigurable partiellement. Ce dernier est actuellement un des systèmes SoPC (System on Programmable Chip) le plus utilisé. Dans cette plate-forme, nous souhaitons bénéficier d'une solution de virtualisation légère, appropriée pour des systèmes dédiés à l'internet des objets. Dans ce contexte, les technologies de virtualisation existantes ne semblent pas appropriées, car leur complexité étant très élevée, il devient inimaginable de les adapter à notre plate-forme cible. En outre, plusieurs technologies, par exemple OKL4, ne sont pas libres et sont donc in-envisageables pour nos travaux. Nous avons également écarté l'algorithme d'ordonnancement hiérarchique, décrit précédemment, car ce dernier nécessite le calcul d'interfaces PRM ou des ordonnancements de serveurs supplémentaires.

Par conséquent, un objectif de nos travaux a consisté à proposer une approche micro-noyau qui prend en charge la virtualisation en temps réel avec un minimum de complexité. Nous avons également proposé une approche originale de virtualisation sur un hyperviseur embarqués que nous avons développé. Des efforts ont été réalisés pour assurer efficacement le partage des ressources reconfigurables entre les machines virtuelles.

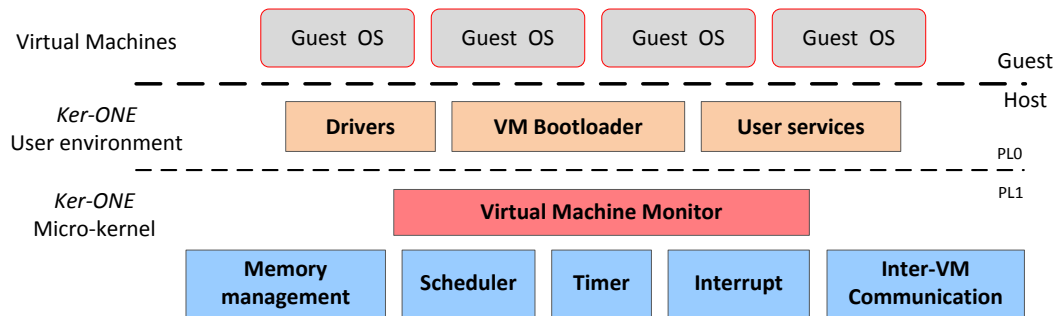


FIGURE 2 – Ker-ONE se compose d’un micro-noyau exécuté en mode privilégié et d’un environnement utilisateur s’exécutant au niveau non-privilégié.

2. KER-ONE : un micro-noyau léger et temps réel

La virtualisation sur plate-formes ARM-FPGA a quelques limitations. La virtualisation complète des processeurs ARM est actuellement disponible en utilisant des extensions de virtualisation ARM comme dans le Cortex-A15. En revanche, cette technologie est actuellement indisponible sur les plateformes ARM-FPGA actuelles. Par conséquent, Ker-ONE est développé sur la base de l’architecture ARMv7 en utilisant le principe de para-virtualisation. La conception de Ker-ONE est dictée par le principe de faible complexité, qui consiste à se concentrer uniquement sur la fonctionnalité de virtualisation critique et élimine tous les services non primordiaux. Ker-ONE offre une petite taille de TCB, et propose un mécanisme permettant de gérer les contraintes temps réel de la plupart des applications. Actuellement, notre recherche est basée sur les hypothèses suivantes :

- Dans une première étape, nous avons seulement considéré des architectures mono-cœur, renvoyant les systèmes multi-cœurs aux futures études.
- Nous nous concentrons sur la virtualisation des systèmes d’exploitation simples, au lieu des gros OSs monolithiques tels que Linux. En effet, para-virtualiser de tels systèmes d’exploitation serait très coûteux et complexe et irait à l’encontre des objectifs de notre travail.
- Afin de réaliser la virtualisation en temps réel avec un ordonnanceur moins complexe, nous supposons que toutes les tâches critiques temps réel s’exécutent dans un RTOS invité spécifique, alors que les autres tâches, moins critiques, s’exécutent sur un OS généraliste (GPOSs). Aussi, Ker-ONE est conçu pour abriter au moins un client RTOS et plusieurs GPOSs.

Ker-ONE se compose du micro-noyau à proprement parler et d’un environnement de niveau utilisateur. La FIGURE 2 représente les éléments clés de notre architecture.

Le micro-noyau est le seul composant fonctionnant dans le mode le plus privilégié, principalement en mode superviseur.

Les caractéristiques de base implantées dans le noyau sont : la gestion de la mémoire, les communications inter-VM, la gestion des exceptions, et l'ordonnanceur. L'hyperviseur tourne au-dessus des fonctions de base du micro-noyau pour permettre l'exécution d'un système d'exploitation invité dans la machine virtuelle associée. Il fournit une couche de matériel virtuel, émule des instructions sensibles et gère les hyper-calls provenant des machines virtuelles. L'environnement utilisateur fonctionne en mode (PL0) et propose des services supplémentaires, tels que les systèmes de fichiers, bootloaders et services spécialisés (comme le gestionnaire de tâches matérielles qui contrôle les accélérateurs matériels dans FPGA Zynq-7000). Une machine virtuelle peut accueillir un système d'exploitation para-virtualisé ou une image logicielle d'une application utilisateur. Chaque machine a un espace d'adressage indépendant, et s'exécute sur le matériel virtuel (CPU virtuel) fourni par l'hyperviseur.

2.1 Gestion de la mémoire

Dans la virtualisation des systèmes, la gestion de la mémoire est essentielle pour fournir des espaces isolés de mémoire pour les machines virtuelles. Ker-ONE offre trois niveaux de privilège de mémoire pour *hôte* (host) (pour VMM), *noyau invité* (pour le noyau de l'OS invité) et *utilisateur invité* (pour les applications de l'OS invité). Chaque niveau est protégé contre les niveaux inférieurs. Par exemple, les pages mémoire dans le *hôte* ne sont accessibles que par le micro-noyau. De plus, les applications en cours d'exécution dans le domaine *utilisateur client* ne peuvent accéder au noyau de l'OS.

Pour chaque machine virtuelle, une table de page indépendante est créée par l'hyperviseur. Une structure de données de MMU virtuelle est associée à chaque machine virtuelle et contient les informations d'espace d'adressage. Les OSs invités peuvent changer le contenu des tables de pages sous la supervision de l'hyperviseur, à l'aide d'hyper-calls.

2.2 Interruptions

KER-ONE gère toutes les interruptions matérielles. Lors de l'exécution dans une machine virtuelle, toutes les interruptions matérielles sont d'abord prises au piège dans l'hyperviseur. KER-ONE gère les interruptions physiques et envoie ensuite une interruption virtuelle correspondant à la machine virtuelle cible si nécessaire. Dans le domaine de la machine virtuelle, ces interruptions virtuelles sont exactement gérées comme les interruptions physiques grâce à l'hyperviseur. Chaque machine virtuelle est libre de configurer les interruptions virtuelles dans son propre domaine, et est indépendante des autres machines virtuelles.

KER-ONE catégorise et divise les sources d'interruptions en différents groupes. Les interruptions utilisées par les systèmes d'exploitation invités sont divisées en deux niveaux : RTOS et GPOS. Une RTOS se voit attribuer un niveau de priorité supérieur à celui d'un GPOS, de sorte que les interruptions d'un RTOS particulier peuvent être ni désactivées, ni être bloquées par un GPOSs. Ceci garantit que les événements prioritaires peuvent être reçus par le RTOS, même lors de l'exécution de GPOSs. Ceci est très important pour assurer l'ordonnabilité des tâches en temps réel.

2.3 Les communications Inter-Process (IPC)

Un mécanisme de communication inter-processus efficace (IPC) est essentiel dans la plupart des micro-noyaux. KER-ONE utilise de simples approches basées sur la communication asynchrone, à base d'interruptions pour faciliter les IPCs.

Pour effectuer une communication inter-VM, l'hyperviseur lève l'interruption correspondante dans la machine virtuelle cible et délivre un message. Pendant ce temps, dans chaque machine virtuelle, une page de mémoire partagée VMM/VM est créée avec une structure nommée canal IPC (IPC Channel). Les machines virtuelles sont en mesure d'envoyer des messages IPC en programmant le canal IPC dans la région partagée. Ce message sera ensuite traité et livré à la machine virtuelle cible lors du prochain ordonnancement. Il est important de noter que ce mécanisme IPC nécessite seulement quelques cycles d'écriture/lecture mémoire sans utiliser d'hyper-calls, ce qui entraîne peu de surcoût en terme de latence.

2.4 Virtualisation des OS temps réel

KER-ONE met en œuvre un ordonnanceur round-robin préemptif, basé sur la priorité. Chaque machine virtuelle est créée avec un niveau de priorité fixe. L'ordonnanceur sélectionne toujours la machine virtuelle de plus haute priorité. Une machine avec une faible priorité ne peut fonctionner que lorsque les machines virtuelles plus prioritaires sont suspendues ou arrêtées. Dans le cas où les machines virtuelles sont de même priorité, le CPU est partagé équitablement.

Un RTOS doit fonctionner à un niveau de priorité supérieur à celui d'un GPOS ; l'hyperviseur permet aux GPOSs de s'exécuter, uniquement lorsque le RTOS est inactif. Le GPOS poursuivra alors son exécution jusqu'à ce que sa tranche de temps soit écoulée, ou jusqu'à ce qu'un autre événement se produise dans le RTOS. Dans les deux cas, l'hyperviseur ré-exécute le RTOS et préempte les GPOSs.

Par rapport à une exécution native, la virtualisation dégrade inévitablement les performances. Par conséquent, en ce qui concerne l'ordonnancement des tâches des RTOS, nous proposons un modèle de coût qui formalise les surcoûts liés à la virtualisation. Nous définissons le temps d'exécution réel d'une tâche qui se compose du WCET (e_i) et de la latence nécessaire à sa libération ou son ordonnancement. Le modèle est le suivant :

$$E_i = e_i + relEv. \quad (1)$$

Et e_i est le chemin d'exécution de la tâche, et $relEv$ est le temps de libération de cette même tâche.

La FIGURE 3 décrit la manière dont la virtualisation affecte le temps d'exécution. Le surcoût engendré par la virtualisation peut être inclus dans le temps d'exécution réel des tâches : E_i^{VM} :

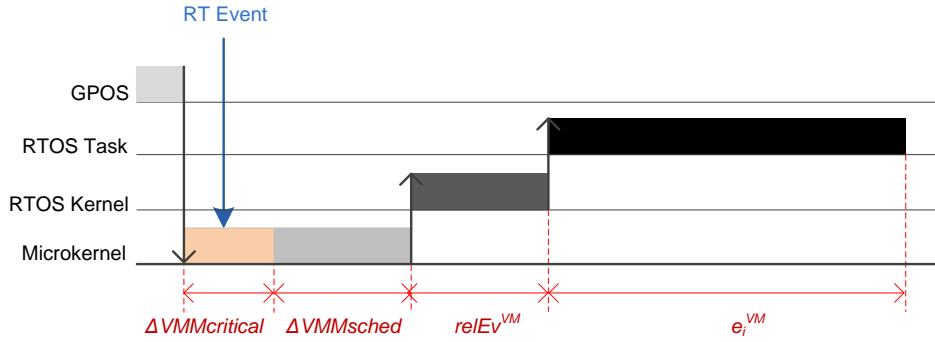


FIGURE 3 – Surcoût d’exécution des tâches d’un RTOS lorsqu’exécutée dans une machine virtuelle. Ce dernier est composé de surcoûts critiques liés à l’ordonnancement et les surcoûts intrinsèques liés au RTOS.

$$\begin{aligned}
 E_i^{VM} &= e_i^{VM} + relEv^{VM} + \Delta VMMsched + \Delta VMMcritical, \\
 \text{and } e_i^{VM} &= e_i^{Native} + \Delta_{VM}^{ei}, \\
 \text{and } relEv^{VM} &= relEv^{Native} + \Delta_{VM}^{relEv},
 \end{aligned} \tag{2}$$

e_i^{VM} et $relEv^{VM}$ sont respectivement le temps réel d’exécution et le temps de libération de la tâche dans une machine virtuelle. Ces temps sont majorés par Δ_{VM}^{relEv} et Δ_{VM}^{ei} , qui correspondent respectivement à la latence nécessaire à l’ordonnancement des tâches et au temps d’exécution dans une machine virtuelle.

$\Delta VMMsched$ est la latence supplémentaire requise pour l’ordonnancement du RTOS. $\Delta VMMcritical$ est le retard causé par l’exécution critique de l’hyperviseur. En résumé, le temps de réponse des machines virtuelles pour prendre en compte un événement temps réel peut être représenté comme suit :

$$Response^{VM} = relEv^{VM} + \Delta VMMsched + \Delta VMMcritical. \tag{3}$$

A partir de $Response^{VM}$, le surcoût sur le temps de réponse d’une tâche peut être obtenu par :

$$\begin{aligned}
 \Delta_{VM}^{Response} &= Response^{Native} - Response^{VM} \\
 &= \Delta_{VM}^{relEv} + \Delta VMMsched + \Delta VMMcritical,
 \end{aligned} \tag{4}$$

Etant donné que l’ordonnancement d’un RTOS est rythmé par les cycles du timer, tous les paramètres d’ordonnancement sont exprimés en nombre de cycles. Le nombre de cycles (Θ_i) représente le temps minimal nécessaire à l’exécution d’une tâche et est exprimé comme suit :

$$\Theta_i = \lceil \frac{E_i^{Native} + \Delta_{VM}^{ei} + \Delta_{VM}^{Response}}{\Delta^{Tick}} \rceil, \tag{5}$$

Δ^{Tick} représente la durée d’un cycle du timer, $\Delta_{VM}^{Response}$ est le surcoût lié au temps de réponse de la tâche.

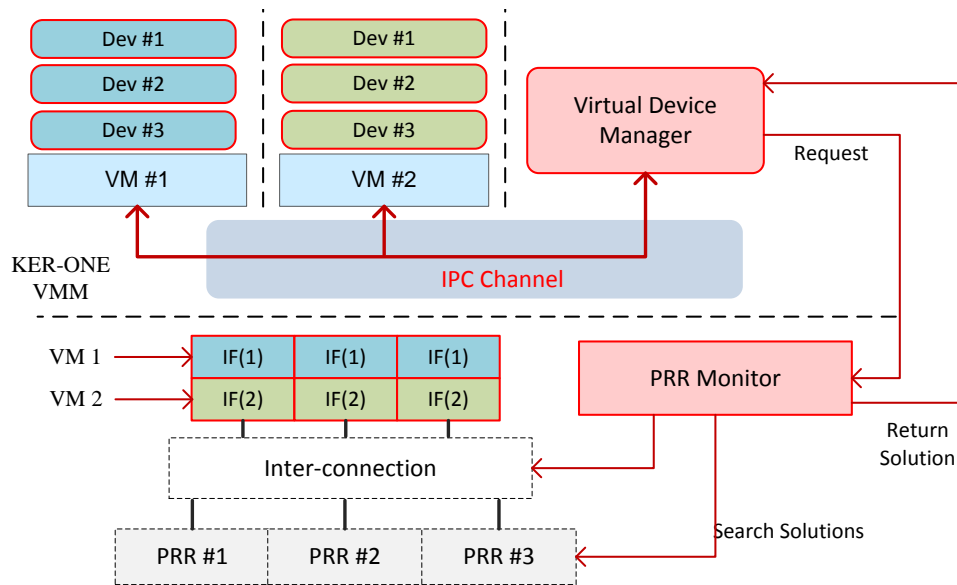


FIGURE 4 – Vue d’ensemble de la gestion de la zone reconfigurable dans KER-ONE.

3. Gestion des accélérateurs reconfigurables

Dans notre système, les accélérateurs reconfigurables sont mis en œuvre dans des régions de reconfiguration partielle prédéterminées (PRR) du FPGA. Nous avons proposé une interface standard pour tous les accélérateurs DPR, de sorte qu’une PRR peut être utilisée par différents accélérateurs. Entre le logiciel et les accélérateurs reconfigurables, une couche intermédiaire composée d’interfaces (IF) a été proposée. Ces IF sont mappées dans l’espace d’adressage des machines virtuelles et sont vues comme étant différents périphériques virtuels. Chaque IF est exclusivement associée à un accélérateur virtuel spécifique dans une machine virtuelle spécifique. Un périphérique virtuel est toujours mappé à la même adresse dans toutes les machines virtuelles, mais est mis en œuvre dans différents IF dans la couche inférieure.

Une interface IF possède deux états, *connecté* à une certain PRR ou *sans lien*. Lorsqu’une interface est *connectée*, elle indique que l’accélérateur virtuel correspondant est mis en œuvre dans la PRR et qu’il est prêt à être utilisé. Etre dans l’état *sans lien* signifie que l’accélérateur est indisponible et les registres de l’interface IF sont mappés uniquement en lecture seule.

Quand une machine virtuelle tente d’utiliser ce périphérique virtuel en écrivant dans son interface, cette action provoquera un défaut de page et sera détectée par l’hyperviseur. Comme les accélérateurs reconfigurables sont indépendants, notre système introduit des mécanismes de gestion supplémentaires pour gérer dynamiquement les demandes d’utilisation des ressources provenant des machines virtuelles.

Dans la figure 4, le mécanisme de gestion proposé est décrit. Le *Gestionnaire de périphériques virtuel* est un service logiciel particulier proposé à une machine virtuelle

visant à allouer des ressources reconfigurables . Dans la partie statique du FPGA, un *PPR Moniteur* est créé et est en charge des interconnexions entre les IF et les PRR, ainsi que de la surveillance dynamique des accélérateurs reconfigurables. Ce service est également en charge de trouver des solutions disponibles pour l'affectation des ressources aux machines virtuelles.

Chaque fois qu'une machine virtuelle accède à une interface IF non connectée, l'hyperviseur exécute immédiatement le *Virtual Device Manager* pour gérer cette demande de ressources. Une requête est de la forme : *Request(vm_id, dev_id, prio)*, qui est composée de l'identifiant de machine virtuelle, de l'identifiant du périphérique virtuel ainsi que d'une priorité de demande qui est en réalité la priorité de la machine virtuelle appelante. Cette demande est envoyée au *Moniteur* du côté du FPGA pour rechercher une solution de configuration appropriée. Cette solution comporte différentes méthodes :

- **Allocate** (*prrr_id*) : alloue directement une zone reconfigurable (*prrr_id*) à une machine virtuelle. Si le périphérique *dev_id* n'est pas déjà implanté dans la zone, un drapeau *Reconfig* est positionné.
- **Preempt** (*prrr_id*) : aucune zone ne peut être directement attribuée, mais la zone (*prrr_id*) peut être préemptée et réaffectée. Si le périphérique *dev_id* n'est pas mis en œuvre dans la zone, le drapeau *Reconfig* sera positionné .
- **Unavailable** : cet état signifie que la zone reconfigurable n'est pas disponible actuellement. Une demande non aboutie est ensuite ajoutée à la liste des recherches dans le *PPR Moniteur* qui continue la recherche de solutions. De nouvelles solutions seront envoyées au *Gestionnaire de périphériques virtuels* immédiatement.

Le *Gestionnaire de périphériques virtuels* effectue ensuite l'allocation en fonction de la solution retournée. L'allocation peut être effectuée en plusieurs étapes : (1) déconnecter la zone reconfigurable des autres IF, et modifier la page de mémoire en lecture seule dans la machine virtuelle correspondante ; (2) la connecter à la machine virtuelle nécessitant la ressource et mettre à jour la page de mémoire en lecture/écriture ; (3) continuer l'exécution de la machine virtuelle à l'endroit où elle a été interrompue. Dans les solutions où l'exécution est différée (à savoir *Reconfig* et *Preempt*), le processus d'allocation ne peut pas être terminé directement car il doit attendre l'achèvement de la reconfiguration ou la préemption. Dans ce cas, le *Gestionnaire de périphériques virtuel* prévient la machine virtuelle en envoyant des requêtes IPC, et laisse au *PPR Moniteur* le suivi de ces solutions côté de FPGA. Ce dernier fournit des interruptions au *Gestionnaire de périphériques virtuels* lorsque la reconfiguration/preemption est terminée.

4. Implémentation

Le micro-noyau Ker-ONE est construit sur l'architecture ARMv7 basée sur une plateforme Zynq-7000. Dans le domaine du logiciel, le système est divisé en trois objets : le micro-noyau Ker-ONE, l'environnement utilisateur, et le client OS/applications. Tant le micro-noyau et de l'environnement de l'utilisateur sont construits de façon indépendante,

ils coopèrent pour établir l'environnement de virtualisation. Les clients/applications sont développés et construits par les utilisateurs.

Dans le cadre de cette thèse, nous avons para-virtualisé un RTOS bien connu, $\mu\text{C}/\text{OS-II}$, qui est massivement utilisé dans l'éducation et la recherche. Le système d'exploitation client virtualisé résultant est appelé Mini- $\mu\text{C}/\text{OS}$. La quantité de lignes de code (LoC) qui a été nécessaire pour modifier l'OS original $\mu\text{C}/\text{OS}$ est très faible. Il faut compter 139 lignes pour le code source modifié et 223 lignes supplémentaires pour le patch.

La conception de Ker-ONE se traduit par une mise en œuvre légère avec une petite taille de TCB. Le micro-noyau KER-ONE est construit avec moins de 3000 lignes de Code. Ce résultat a été obtenu grâce aux efforts fournis pour éliminer toutes les fonctionnalités inutiles du noyau.

5. Evaluations

5.1 Impact de la virtualisation

Nous avons mené plusieurs expériences afin d'évaluer les performances de Ker-ONE. Les expériences sont basées à la fois sur des mesures et des critères standards. Dans une première étape, nous avons évalué les fonctions de base de l'hyperviseur afin d'obtenir une évaluation générale. Ces fonctions se traduisent par un faible surcoût en termes de temps d'exécution. Ce surcoût temporel a été estimé à $1 \mu\text{s}$.

Ensuite, nous nous sommes concentrés sur la performance du client RTOS. Nous avons effectué des tests qualitatifs sur les fonctions du RTOS, et mesuré l'impact des surcoûts réels sur l'ordonnancement. Finalement, nous avons exécuté des applications embarquées pour estimer une vitesse effective. Sur la base des résultats obtenus, nous avons présenté une analyse approfondie de l'ordonnancement des clients RTOS. Les résultats montrent que notre approche met en œuvre la virtualisation en temps réel avec de faibles coûts et des performances proches de celles obtenues avec un OS natif.

Sur la base des résultats de l'expérience évaluant l'impact de la virtualisation sur le RTOS original, l'impact global que la virtualisation provoque sur le temps de réponse du RTOS $\Delta_{VM}^{Response}$ peut être estimée à $3.08 \mu\text{s}$. L'équation Eq.(5) peut alors être simplifiée comme suit :

$$\Theta_i \approx \lceil \frac{E_i^{Native} + \Delta_{VM}^{ei}}{\Delta^{Tick}} \rceil, \text{ if } \Delta_{VM}^{Response} \ll \Delta^{Tick}. \quad (6)$$

Par conséquent, l'influence majeure sur la configuration de l'ordonnancement du RTOS est causée par Δ_{VM}^{ei} . Pour garantir l'ordonnancement du système, le temps d'exécution réel des tâches doit être mesuré avec précision. Pour une T_k qui fonctionne à l'origine sur un RTOS natif, son ordonnancement est toujours garanti sur Ker-ONE si :

$$\forall T_i \in T_k, \lceil \frac{E_i^{Native} + \Delta_{VM}^{ei}}{\Delta^{Tick}} \rceil = \lceil \frac{E_i^{Native}}{\Delta^{Tick}} \rceil. \quad (7)$$

Si cette contrainte n'est pas respectée, la configuration des paramètres d'ordonnancement pour T_k doit être recalculée pour répondre aux contraintes temps réel.

5.2 Evaluation du coût de la reconfiguration dynamique

Notre évaluation porte sur la latence d'allocation d'une tâche au sein d'une zone reconfigurable. Cette latence provient de plusieurs sources : les défauts de pages, les IPCs, l'ordonnancement des machines virtuelles et l'exécution du *Gestionnaire de périphériques virtuels*. Les coûts globaux sont résumés dans la TABLE 1.

TABLE 1 – Les overheads de l'allocation des DPR

Methods	Overheads (μs)
{Assign}	$3.03\mu s$
{Assign, Reconfig.}	$6.76\mu s + T_{RFCG}$
{Preempt}	$5.10\mu s + T_{preempt}$
{Preempt, Reconfig. }	$9.96\mu s + T_{preempt} + T_{RFCG}$

D'après le tableau, on peut clairement observer que les allocations directes peuvent être efficacement réalisées avec une latence de $3 \mu s$.

6. Conclusion et perspectives

Cette thèse décrit un micro-noyau original permettant de gérer la virtualisation des systèmes embarqués et fournissant un environnement pour les machines virtuelles en temps réel. Nous avons simplifié l'architecture du micro-noyau en ne gardant que les caractéristiques essentielles requises pour la virtualisation, et massivement réduit la complexité de la conception du noyau. Sur la base de ce micro-noyau, nous avons mis en place un environnement capable de gérer des ressources reconfigurables dans un système composé de machines virtuelles. Les accélérateurs matériels reconfigurables sont mappés en tant que dispositifs classiques dans chaque machine. Grâce à une gestion efficace de la mémoire dédiée, nous permettons de détecter automatiquement le besoin de ressources et autorisons l'allocation dynamique.

Selon diverses expériences et évaluations, nous avons montré que Ker-ONE ne dégrade que très peu les performances en termes de temps d'exécution. Les surcoûts engendrés peuvent généralement être ignorés dans les applications réelles. Nous avons également étudié l'ordonnançabilité temps réel dans les machines virtuelles. Les résultats montrent que le respect de l'échéance des tâches du RTOS est garanti. Nous avons également démontré que le noyau proposé est capable d'allouer des accélérateurs matériels très rapidement.

Pour les travaux futurs, nous envisageons de proposer une gestion de la mémoire plus sophistiquée afin que de gros OS généralistes comme Linux puissent être pris en charge. Nous souhaiterions aussi porter notre micro-noyau sur une architecture ARM plus avancée afin d'exploiter pleinement l'extension de la virtualisation matérielle. D'autre part, la politique d'allocation des ressources DPR doit être pleinement étudiée. Il serait intéressant de développer l'algorithme de recherche de solutions plus sophistiquées et de

discuter de l'influence des différents paramètres à prendre en compte pour l'allocation des ressources. Finalement, nous souhaitons considérer la mise en œuvre de scénarios réels afin de pouvoir concrètement évaluer la performance de notre approche par rapport aux solutions existantes dans ce domaine.

TABLE OF CONTENTS

Acknowledgement	i
Résumé de la Thèse	iii
Table of Contents	xvi
Abbreviations	xxi
Introduction	1
1 CONCEPTS AND RELATED WORKS	5
1.1 Basic Virtualization Theories	5
1.1.1 Fundamental Theories of Virtualization	5
1.1.1.1 Machine Model	6
1.1.1.2 Mapping Mechanism	8
1.1.1.3 Instruction behavior	9
1.1.2 Virtualization Approaches	10
1.1.2.1 Hardware Virtualization Extension	11
1.1.2.2 Dynamic Binary Translation	13
1.1.2.3 Para-Virtualization	14
1.2 ARM-based Embedded System Virtualization	16
1.2.1 Micro-kernels	19
1.2.1.1 L4 Micro-kernels	20
1.2.1.2 OKL4 Microvisor	21
1.2.2 Hypervisors	21
1.2.2.1 KVM	22
1.2.2.2 Xen	24
1.2.3 ARM-based Full Virtualization	25
1.3 Real-time Scheduling for Virtual Machines	26
1.3.1 Compositional Real-time Scheduling	27
1.3.1.1 L4/Fiasco Micro-kernel with Compositional Scheduling	29
1.3.1.2 RT-Xen	30
1.3.2 Other Real-time Approaches	31

1.4	CPU-FPGA Hybrid Architecture	32
1.4.1	Dynamic Partial Reconfiguration	34
1.4.2	CPU/FPGA Execution Model	35
1.4.2.1	CPU/FPGA offload model	36
1.4.2.2	CPU/FPGA unified model	37
1.4.3	DPR Resource Virtualization	39
1.5	Summary	42
2	KER-ONE : LIGHTWEIGHT REAL-TIME VIRTUALIZATION ARCHITECTURE	45
2.1	Overview of the Ker-ONE Microkernel	45
2.2	Resource Virtualization	48
2.2.1	CPU virtualization	48
2.2.1.1	CPU Resource Model	50
2.2.1.2	Instruction Emulation	52
2.2.1.3	Virtual CPU Model	54
2.2.2	Vector Floating-Point Coprocessor Virtualization	55
2.2.3	Memory Management	57
2.2.3.1	Memory Access Control	57
2.2.3.2	Address Space Virtualization	59
2.2.3.3	Memory Virtualization Context	61
2.3	Event Management	61
2.3.1	Interrupt Virtualization	62
2.3.1.1	Emulation of Interrupts	63
2.3.1.2	Virtual Interrupt Management	65
2.3.2	Timer Virtualization	66
2.3.3	Inter-VM Communication	67
2.4	Optimization	68
2.5	Real-time OS Virtualization	70
2.5.1	VMM Scheduler	71
2.5.2	RTOS Events	72
2.5.3	RTOS Schedulability Analysis	73
2.6	Summary	75
3	DYNAMIC MANAGEMENT OF RECONFIGURABLE ACCELERATORS ON KER-ONE	77
3.1	Introduction to the Zynq-7000 platform	77
3.1.1	PS/PL Communication	78
3.1.2	Partial Reconfiguration	80
3.1.3	Interrupt Sources	81
3.2	DPR Management Framework	81
3.2.1	Framework Overview	82
3.2.2	Hardware Task Model	84

3.2.3	PRR State Machine	86
3.2.4	PR Resource Requests and Solutions	88
3.2.5	Virtual Device Manager	90
3.2.6	Security Mechanisms	94
3.2.7	Initialization Sequence	95
3.3	Application of Virtual Devices	96
3.3.1	Virtual Device Blocking	97
3.3.2	Virtual Device Preemption	97
3.4	Summary	99
4	IMPLEMENTATION AND EVALUATION	101
4.1	Implementation	101
4.1.1	RTOS Para-virtualization	102
4.1.2	Complexity	103
4.1.3	System Mapping	104
4.1.4	Ker-ONE Initialization Sequence	106
4.2	Ker-ONE Virtualization Performance Evaluations	106
4.2.1	Basic Virtualization Function Overheads	107
4.2.2	RTOS Virtualization Evaluation	109
4.2.2.1	RTOS Benchmarks	109
4.2.2.2	RTOS Virtualization Overheads	113
4.2.3	Application Specific Evaluation	115
4.2.4	Real-time Schedulability Analysis	116
4.3	DPR Allocation Performance Evaluation	117
4.3.1	Overhead Analysis	117
4.3.2	Experiments and Results	119
4.3.3	Real-time Schedulability Analysis	123
4.3.3.1	Response Time	123
4.3.3.2	Execution Time	124
4.4	Summary	125
5	Conclusion and Perspectives	127
5.1	Summary	127
5.2	Perspectives and Future Works	129
	List of Figures	131
	List of Tables	137
	Bibliography	137

ABBREVIATIONS

API : Application Programming Interface
ASIC : Application-Specific Integrated Circuit
CP : Control co-Processor
CPSR : Current Processor State Register
DACR : Domain Access Control Register
DMA : Direct-Memory Access
DPR : Dynamically Partial Reconfiguration
EC : Execution Context
FFT : Fast Fourier Transform
FPGA : Field Programmable Gate Array
GIC : Generic Interrupt Controller
GPOS : General-Purpose Operating System
IP : Intellectual Property
IPC : Inter-Process Communication
IRS : Interrupt Routine Service
ISA : Instruction Set Architecture
IVC : Inter-Virtual machine Communication
LoC : Lines of Code
MMU : Memory Management Unit
OFDM : Orthogonal Frequency Division Multiplexing
OS : Operating System
PCAP : Processor Configuration Access Port
PL : Programmable Logic
PL0/1 : Privilege Level 0/1 **PRR** : Partial Reconfigurable Region
PS : Processing System
PSR : Processor Status Register
QAM : Quadrature Amplitude Modulation
RM : Reconfigurable Module
RP : Reconfigurable Partition **RTOS** : Real-Time Operating System
SoC : System on Chip
SRAM : Static Random Access Memory
TCB : Trusted Computing Base
TTBR : Translation Table Base Register

TTC : **T**riple **T**imer **C**ounter

VD : **V**irtual **D**evice

VFP : **V**ector **F**loating-**P**oint

VHDL : **V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage

VM : **V**irtual **M**achine

VMM : **V**irtual **M**achine **M**onitor

INTRODUCTION

Context

Virtualization has become more and more popular in the embedded computing domain, especially for today's intelligent portable devices such as smart-phones and vehicles. This technology offers the advantage of better energy efficiency, shorter time-to-market cycles and higher reliability. As a result, the exploration of virtualization techniques on embedded systems constitutes a hot topic in the industrial and personal embedded computing domains.

Traditionally, embedded computing systems are used to be relatively simple single-purpose devices, where software is subject to particular constraints, such as power consumption or real-time processing. Only simple operating systems (OS) or bare-metal applications generally run on top of systems. Modern embedded systems, however, are increasingly playing the role of general-purpose systems, due to their growing computing abilities and resources. In this case, software-based systems that were formerly divided to provide different functions are now merging into one platform. As an example, in modern embedded devices, some critical background services may run concurrently with some user-oriented applications, which may be designed for different systems.

Virtualization technology can help device vendors by enabling concurrent execution of multiple operating systems, so that the legacy software stack can be easily ported to contemporary platforms, saving the expensive development and verification efforts. On the other hand, some conventional constraints still remain for embedded systems : limited resources (compared to PCs), real-time processing, and limited battery capacity, which are critical factors in virtualization. Furthermore, in such a mixed-criticality system, it is essential for a virtualization system to guarantee the correctness of critical task timing constraints, since they may significantly influence the system safety. In this case, the system scheduling mechanism is forced to guarantee the real-time tasks deadlines, so that the system manner can be predictable.

One key characteristic of virtualization technology is security, i.e. the isolation and independence among different system components, so that any malfunction or incorrect behaviors are constrained to their own domains. The virtual and physical computing platforms are decoupled via virtual machine (VM), which provides each hosted guest with a separate and secure execution environment. The guest OS runs on the virtual platform, while the actual physical layer is managed by an underlying hypervisor, the virtual ma-

chines monitor (VMM). Virtual machines are temporally and logically independent from each other and cannot affect the system outside its domain. Such systems are also called virtual machine systems. In fact, the issue of security has become increasingly important since OS security is more and more threatened by malicious pieces of code. In order to ensure system security, hypervisors must be kept as small as possible and feature a small trust computing size (TCB).

Besides these software aspects, another challenge for embedded device vendors is that the improvement of computing performance has become more and more difficult to manage with traditional approaches such as IC scaling and ASIC. As an alternative, the interest of vendors have progressively move on to the concept of CPU-FPGA hybrid System-on-Chip (SoC). Currently, FPGA devices are already popular in embedded system designs as a result of their outstanding flexibility, low power consumption and speed-up performance. This is especially true with the Dynamic Partial Reconfiguration (DPR) technique, which has revealed considerable potential in hosting hardware accelerators with less resource consumption. Therefore, the idea of enhancing embedded processor with an FPGA fabric has been considered as a promising solution to improve performances. Xilinx and Altera have both released ARM-FPGA hybrid embedded systems, while Intel's Xeon-FPGA devices are also on their way.

Embedded systems that are based on this type of platforms can benefit from the flexibility of processors and from the FPGA performances. This makes it possible to extend the virtualization technology to the reconfigurable computing technology. In this case, virtual machine users can easily and securely access IP accelerators on the FPGA using the DPR technique.

In this context, a new challenge consists in providing an efficient management mechanism while respecting the independence of virtual machines. The reconfigurable resources have to be virtualized so that virtual machines can access them independently. This feature is critical in virtualization to guarantee the system security. However, though reconfigurable accelerators on conventional embedded systems have been studied in numerous researches, they are mostly implemented in simple applications and OSs, while the usage and management of the DPR technique in a virtualized environment remains an open problem.

This thesis is based on the mentioned issues of current embedded systems. In our research, we focus on the small-scaled embedded ARM-FPGA hybrid systems. These systems are designed with limited resources, and may be used for small devices such as Internet-of-Things (IoT). In this context, we propose an innovative lightweight virtualization approach that is capable of hosting real-time virtual machines and efficiently utilizing reconfigurable FPGA devices. Our approach is implemented with low complexity and high adaptivity, and is suitable for embedded devices that try to take advantage of both virtualization and DPR techniques. In this dissertation we have also run extensive experiments on our system to evaluate the overall performance of the proposed approach.

Manuscript organization

The remainder of this thesis is organized as follows : In CHAPTER 1, we introduce the general techniques and concepts used throughout this thesis. We first focus on the major theories of virtualization technology in embedded systems, dealing especially with real-time constraints. We then describe the concepts and principles of DPR technology. We will also present the state of the art in related researches, introducing their principles and major features. In CHAPTER 2, we describe in details the proposed virtualization architecture. Fundamental structures and designs are presented and explained. We also focus on the real-time virtualization mechanism. In CHAPTER 3, we propose the management of DPR modules in our system, describing both sharing and security mechanisms. In CHAPTER 4, we demonstrate the evaluations taken from standard open-source benchmarks as well as custom experiments. The performance of our system in terms of real-time scheduling and reconfigurable computing are given and analyzed. We conclude this thesis and give the perspectives of our work in CHAPTER 5.

CHAPTER 1

CONCEPTS AND RELATED WORKS

1.1 Basic Virtualization Theories

In the context of computing systems, the "virtualization" term refers to a system virtualization technology, which permits multiple operating systems to be executed simultaneously on the same physical machine. The concept of virtualization dates back to the 1960s and has been initially implemented in IBM's mainframes [Gol74]. In 1974, Popek and Goldberg defined formal system virtualization criteria for computing architectures, as the essential conditions for a computing machine to be virtualized. These criteria constitute the groundwork of the virtualization technology [PG74]. Starting from this model, virtualization experienced decades of research, and has achieved tremendous success in both academic and commercial domains. In this section, we first introduce the fundamental theories of virtualization technology.

1.1.1 Fundamental Theories of Virtualization

In modern computer architectures, a working space is generally divided into several execution layers called *privilege levels* (or *rings*), in order to protect critical resources. While the highest privilege level is authorized to freely control the whole computing machine resources, the lowest privilege level, or the non-privilege level, is only permitted to limited non-critical parts. Accessing high-privilege resources from low-privilege levels may cause an exception, denoted as a *trap*, which will alert the processor of the forbidden behavior. Thus, the highest privilege level is always used by operating systems to hold the most important codes and data to ensure the security, while user applications and processes are running on the non-privilege levels.

In classic implementations, virtualization is achieved by adding a software abstract layer that interfaces the platform resources and the users, which decouples the underlying hardware from the operating systems [Gol74]. The de-privileged operating systems are executed in a virtual and isolated environment that emulates the hardware resources, and are viewed as guest operating systems. A virtual machine (VM) refers to all software components, e.g. guest operating systems and applications that are running within such

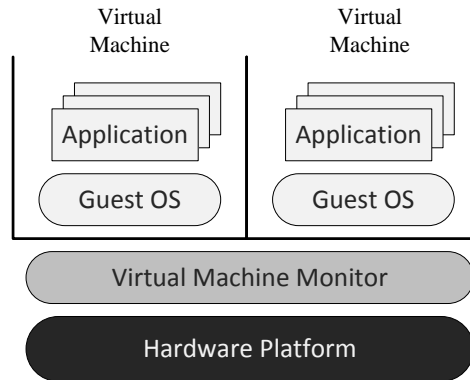


FIGURE 1.1 – Typical virtual machine system architecture

environments. Virtual machines are under the supervision of a virtual machine monitor (VMM). This architecture is also called the virtual machine system. The typical architecture of virtual machine system is shown in FIGURE 1.1.

The Virtual machine monitor is the most privileged component of a virtualization system, and runs directly on the hardware. The VMM provides a basic functionality that is similar to an OS kernel : controlling the hardware platform and managing the components on its top. Additionally, VMM ensures the temporal and logical isolation of virtual machines in the less-privilege levels. Any behavior of a virtual machine is constrained to its own domain, which makes it impossible to influence the outside world. In a sense, VMM holds virtual machines as traditional OS holds processes, except that virtual machines are given the impression that they are granted with the whole hardware platform.

In virtual machine systems, two metrics should be considered : the virtualization efficiency and the *trust computing base* (TCB) size. Virtualization efficiency evaluates the mechanism of virtualization, indicating the adaptivity of a virtualization approach. In most cases, higher virtualization efficiency means lower performance degradation for OS hosted in virtual machine. On the other hand, the TCB is the part of software that runs with the highest privilege and must be trusted. In the context of virtualization, TCB size generally corresponds to the size of VMM. Lower TCB size implies less threats and smaller attack surface from malicious user applications or guest OS that has been compromised. For the consideration of system security, small-sized TCB is always preferred in virtualization techniques.

1.1.1.1 Machine Model

The criteria of virtual machine systems were formally defined by Popek and Goldberg [PG74], which are well known as the conditions for *classical virtualizability*. Architectures that meet these criteria are considered as capable to host virtual machines. The machine model that Popek and Goldberg used was based on the computers available at that time,

such as DEC PDP-10 and IBM 360/67, and was intentionally simplified as :

$$S \equiv \langle E, M, P, R \rangle \quad (1.1)$$

In this definition, the state of the machine model S is composed of the memory E , the processor mode M , the program counter P and the relocation-bounds register R . This model considers the memory space as linearly addressable and ignores the interactions with I/O devices and interrupts, which were rational simplifications according to the state of micro-processors at that time.

However, due to the advances in computing technologies, the model no longer fits current architectures. Some novel characteristics of modern computers have been greatly developed, and have become too significant to be ignored, which are listed as follows :

- Paged virtual memory system is fully exploited in modern operating systems to provide applications with many key facilities and protections, such as dynamic linking, shared libraries and independent virtual address spaces. The details of mapping and translations must be included in the machine state.
- Modern machines are also significantly sensitive to timing. In some cases, timing-dependent behaviors influence not only the user experiences but also the correction of the system.
- Peripherals and integrated controllers also play important roles in contemporary computers, for they help the processor to manage hardware resources. They may include various controllers (e.g. interrupt controller and bus controller), and general I/O devices.

Therefore, some researchers have attempted to extend the model proposed by Popek and Goldberg to adjust it to modern computer architectures. In [VGS08] authors re-defined the *classical virtualizability* model by extending it to include system constructs that do not belong to the original model. The research of [DH10] also tried to include interrupts and I/O in their updated model. However, the complete machine model of current computer architecture was given by Penneman in [PKR⁺13] :

$$S \equiv \langle E, M, P, G, C, A, D^M, D^P \rangle . \quad (1.2)$$

In this model, besides original elements E , M and P , additional objects are included : the general-purpose registers G , the configuration registers C , the address space mapping A , the Memory-Mapped IO (MMIO) device state D^M and the Port-Mapped IO (PMIO) device state D^P . Note that, even though I/O devices may be mapped into the physical memory, the device state D^M/D^P is omitted from E .

When considering timing-dependent systems, this definition of machine model can also be revised to be :

$$S \equiv \langle E, M, P, G, C, A, D^M, D^P, T \rangle , \quad (1.3)$$

where the additional element T stands for the execution timings of computer.

1.1.1.2 Mapping Mechanism

In modern computing machines, paged virtual address translation mechanism has been critical for resource management. Software are compiled and executed according to the virtual address, while the actual physical space is generally hidden from them. Via the translation table, each virtual address page can be assigned with independent physical memory spaces and access permissions. For simplification, we can assume that all memory accesses are performed in the virtual address space.

According to the definition of machine model Eq.(1.2), virtual addresses are accessed by the address mapping A , which can be defined as :

Definition 1. *Address mapping $A(V, P, X)$ maps the physical address space P to a virtual address space V according to the access control policy X . A consists of a set of 3-tuples (v, p, x) in which $v \in V$ is a virtual address, $p \in P$ is the corresponding physical address, and x is the access permission identifier.*

We denote the translation function performed according to translation table A as $Trans_A(v, x)$. Thus, its virtual address space V is valid when :

$$\forall v \in V, \exists p \in P, p = Trans_A(v, x). \quad (1.4)$$

Upon every memory access, the translation function $Trans_A(v, x)$ calculates the physical memory space according to the virtual address v and permission check x . If for an address v , the corresponding translation does not exist or is not permitted, a memory trap will be triggered.

In the context of virtual machine systems, address mapping becomes more complex than traditional physical-to-virtual translation. Operating systems tend to build up its own translation tables to create virtual address space. In virtualization, however, the physical memory space of guest OS needs to be virtualized by the VMM. In this case, a two-stage translation is performed. First, the virtual address space (V_g) of guest OS is translated to the guest OS physical address (P_g). Second, P_g is translated to the actual physical memory (P_h) in the host machine, as shown in FIGURE 1.2.

Therefore, VMM is responsible to manage both the address mappings of guest OS and host machine, respectively denoted as $A_g(V_g, P_g, X_g)$ and $A_h(P_g, P_h, X_h)$. We assume E_N is the physical memory space which VMM allocates to virtual machine VM_N on the host machine, the address space of virtual machine VM_N is valid when it meets the criterion :

$$\forall v \in V_g, \exists p \in E_N, p = Trans_{A_h}(Trans_{A_g}(v, x_g), x_h). \quad (1.5)$$

Maintenance of address mappings is one of the most complicated functionality of VMM. A guest OS is generally permitted to freely map the virtual addresses. During the translation, VMM should respect the guest translation table and accordingly allocate memory resources. For typical memory management units (MMU) that only support one-stage translation, one solution is to compose the guest and VMM mappings into a single address mapping, which directly maps the guest virtual address to the host physical memory. This map is called shadow mapping [AA06], which is shown in FIGURE 1.3. By

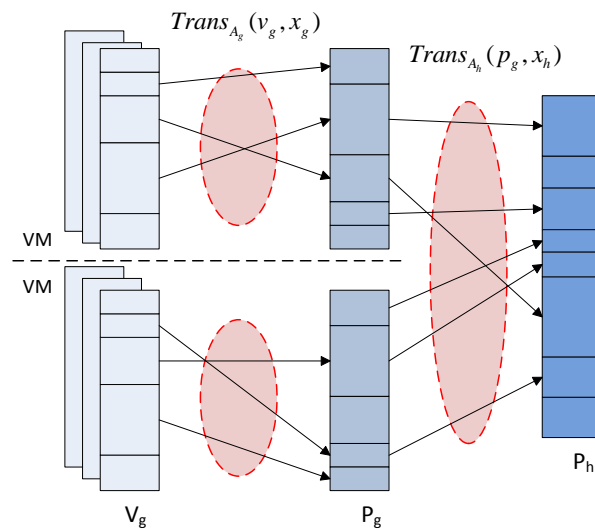


FIGURE 1.2 – Two-stage address mapping in virtual machine systems.

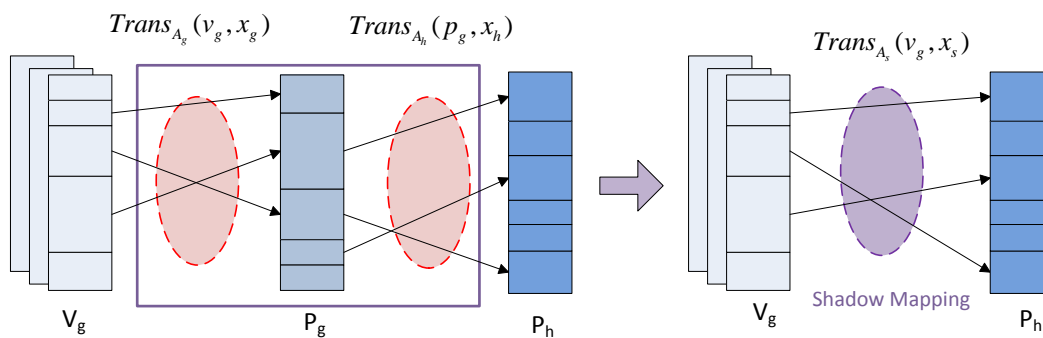


FIGURE 1.3 – Using shadow mapping to perform two-stage address mapping.

denoting the shadow mapping as $A_s(V_g, P_h, X_s)$, the access permission condition X_s must meet the following criteria :

$$\forall v \in V_g, Trans_{A_s} \text{ traps when } Trans_{A_g}(v, x_g) \text{ traps} \cup Trans_{A_h}(p_g, x_h) \text{ traps.} \quad (1.6)$$

As we can notice, the mechanism of shadow mapping requires additional effort from the VMM to monitor and establish protected shadow page table based on the mappings of guest and host machine. Therefore, to accelerate the maintenance of shadow mapping, in some computer architectures (e.g. ARMv7A/v8) MMU is enhanced with dedicated two-stage translation mechanisms, so that the translation from guest virtual address to host physical space is automatically handled by hardware [ARM12].

1.1.1.3 Instruction behavior

In the context of virtualization, instructions are classified in three categories :

- **Privileged instructions** are those that execute correctly only in privilege level, and always trap in non-privilege level.
- **Sensitive instructions** are the ones that may threaten the correction of virtualization. They can be further classified as : (1) *control-sensitive* instructions which manipulate the processor configurations ; (2) *behavior-sensitive* instructions whose results depend on the processor execution mode.
- **Innocuous instructions** those instructions that are not sensitive.

Sensitive instructions must be monitored or prevented in virtual machines, since virtual machines are not allowed to change the configuration of processors, such as the execution mode or physical memory allocations. Furthermore, considering that guest OSs are not running at the privilege levels that they are originally intended for, the results of *behavior-sensitive* instructions are then incorrect.

Base on the classification of instructions, in [PG74], the theorem that defines the condition of instructions in virtualization was proved :

Theorem 1. (Popek and Goldberg [PG74]) *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

According to this theorem, in an instruction set architecture (ISA) that meets the criterion, a trap will be triggered whenever sensitive instructions are executed in virtual machines. In this case, these behaviors are automatically detected and monitored, which is ideal for the construction of VMM, since software based on this ISA can be easily monitored in virtual machines. For this reason, such an instruction set is considered as *virtualizable*. A VMM based on a virtualizable ISA is called an *execute-and-trap* VMM.

In a virtual machine, the behaviors of instructions are constrained to the virtual machine domain, and should not influence the outside world. All sensitive instructions have to be interpreted and properly handled by the VMM. The virtualization of instructions follows the principles :

- **Efficiency** : all innocuous instructions are executed natively without the VMM intervention ;
- **Resource control** : guest software is forbidden access to physical state and resources ;
- **Equivalence** : guest software behaves identically as it runs natively on a system.

1.1.2 Virtualization Approaches

With decades of research, virtualization has been implemented on various platforms and architectures with different solutions. Considering that virtual machines are built on the simulation of physical platforms, the features of instruction set and processor architecture may significantly influence the approaches. Some processor vendors have

considered the demand of virtualization, and provided virtualizable architectures, including virtualizable ISAs and extensional hardware assistance. Such architecture will be ideal for virtualization since a VMM can be easily established. In other cases, for architectures which are not suitable for virtualization, alternate approaches are proposed to implement virtualization.

Generally, while being hosted in a virtualized environment, the performance of guest software will be inevitably degraded, so we wish to reduce the complexity of virtualization mechanisms to minimize the extra workload on the machine. On the other hand, the VMM must guarantee the isolated and secure virtual machine environment. To give more details, the implementation of virtualization should consider the following principles :

- Low complexity or footprint of VMM, so that the resources required for virtualization may be minimized.
- Low reliance on the guest OS source code. Virtualization should minimize the modification of guest software to maintain adaptability. The ideal solution is to host native OS directly in virtual machines without any modification.
- High virtualization efficiency, which means reducing virtualization cost to host guest software with a level of performance close to the native one.
- Strong isolation of system components, so that virtual machines are temporally and logically separated from each other.

While it is not likely that all principles could be emphasized equally, the solution of virtualization should try to maintain balance. For example, an embedded system which has limited resources may prefer a small-sized VMM. On the other hand, a tight-timing system can afford significant modifications on guest OS to obtain a higher performance.

There are two main virtualization approaches : full virtualization and para-virtualization. The full virtualization technique is also called native virtualization [ISM09]. It requires no modification of the guest software, and relies on the VMM to emulate the low-level features of the hardware platform. This feature allows native OSs like Linux or Android to run directly inside the virtual machines. Since it does not rely on the OS code, even close-source software can be easily hosted. Full virtualization relies on supporting technologies e.g. virtualizable ISA and hardware extensions.

Para-virtualization, on the other hand, refers to communication between the guest software and the VMM to implement virtualization. This mechanism is mostly implemented by modifying the guest software codes. Para-virtualization is especially suitable for architectures without hardware assistance or virtualizable ISA.

In this section we will introduce the existing technologies of full virtualization and para-virtualization. We will also overview the characteristics of different approaches.

1.1.2.1 Hardware Virtualization Extension

Considering the potentiality of virtualization, some contemporary computer architectures have introduced hardware support to help building virtual machine systems. This

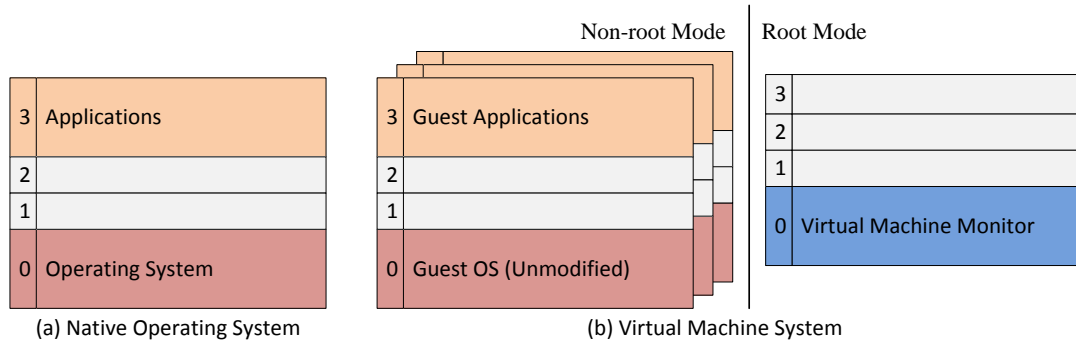


FIGURE 1.4 – Virtualization extension provided by Intel VT-x/VT-i, where the guest software execute in original privilege levels, but in the non-root mode.

is denoted as *hardware virtualization extension*. Typical extensions include additional execution mode, virtualizable ISA, multi-stage MMU translation and virtual interfaces of system registers. Such extensions have been introduced to the traditional PC and server world for a decade, and have been used for the new generations of embedded processors recently. Here, we introduce two typical architectures : Intel VT-x/VT-i and ARM Cortex-A15.

Intel released processors with virtualization technology Intel VT-x and VT-i in 2005, which respectively support Intel architecture 32-bit (IA-32) and Itanium Architecture [UNR⁺05]. The principle extension is the addition of a new processor mode, which we denote as the root mode. Traditionally, Intel microprocessors provide four privilege levels, *Ring 0-3*, using 0 for most-privileged software and 3 for the least privileged. Most software and operating systems use only rings 0 and 3, as shown in FIGURE 1.4(a). With the root mode, four more privilege rings are provided, which have the same features as to the non-root rings except that they are granted with more instructions and resources. Guest software can freely use the four privilege rings that they are originally intended for, but in a non-root mode. In this case, the VMM can be easily built up in the root mode and guest operating systems can be directly hosted, as shown in FIGURE 1.4(b).

ARM proposes a different approach to assist virtualization. Conventional ARM processor runs with two privilege levels : non-privilege level *PL0* and privilege level *PL1*. By default, operating systems utilize *PL1* the secure kernel space, and leave *PL0* for user applications and processes to run, as shown in FIGURE 1.5(a). In the Cortex-A15 architecture, a higher privilege level, *hypervisor mode* (HYP), is added [Lan11]. Note that, while Intel VT-x/VT-i root mode provides replica of four privilege rings, the HYP mode in Cortex-A15 is a completely new execution mode, with a dedicated set of features. The original features of *PL0* and *PL1*, e.g. instructions and hardware resources, remain unchanged. As higher privilege level (*PL2*), HYP mode accesses extra system registers to control the behavior of *PL0* and *PL1*. Thus, guest software can execute as native in *PL0* and *PL1* levels, and the VMM can be implemented at *PL2*. All sensitive instructions are trapped into the HYP mode and handled by the VMM. The mechanism is demonstrated

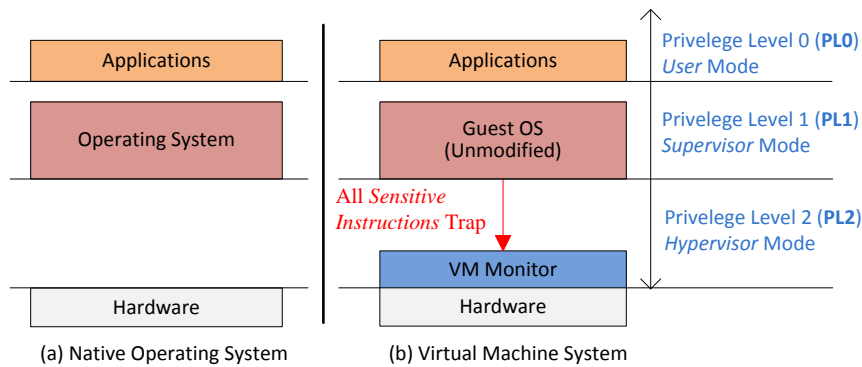


FIGURE 1.5 – In Cortex-A15 with hardware virtualization extension, guest OSs run at the same privilege structure as before so that they can run the same instructions. New HYP mode has higher privilege, where VMM is established to control wide range of OS accesses to hardware.

in FIGURE 1.5(b).

Furthermore, hardware extensions also provide additional hardware mechanism to help virtualize the resources. For example, the ARM Cortex-A15 processor introduces a MMU that supports two-stage translation, which translates a virtual address into *intermediate address space*, before actually having the physical address. This mechanism significantly simplifies the mechanism of shadow mapping, as we mentioned in the Section 1.1.1.2. Cortex-A15 also provides a virtual interrupt controller and virtual timers. The VMM may directly leverage and allocate these virtualized resources to virtual machines. These dedicated hardware features may largely improve the virtualization efficiency [Lan11].

1.1.2.2 Dynamic Binary Translation

Full virtualization can also be achieved without extensions of hardware architecture. This is typically supported by *dynamic binary translation* (DBT), which is also called *software dynamic translation* [Hor07]. This approach translates the instructions of guest software during its execution, on the fly, in order to replace non-virtualizable sensitive instructions with new sequences of instructions that have the intended effect on the virtual hardware. The basic principle of code-rewriting is to execute modified instructions as native instructions, without intervention of the VMM, so that the costs of *traps-and-emulation* are reduced. In this case, the guest OS can be developed as if in native machine, and requires no beforehand modification. In fact, the DBT technique is the only full-virtualization solution that is possible for architectures without hardware assistance or virtualizable ISA. The mechanism of the DBT technique is depicted in FIGURE 1.6.

A major drawback of DBT is the heavy workload that it requires for code interpretation and translation, which takes up considerable resources. Considering the complexity, a DBT-based virtualization normally relies on a host operating system to process the

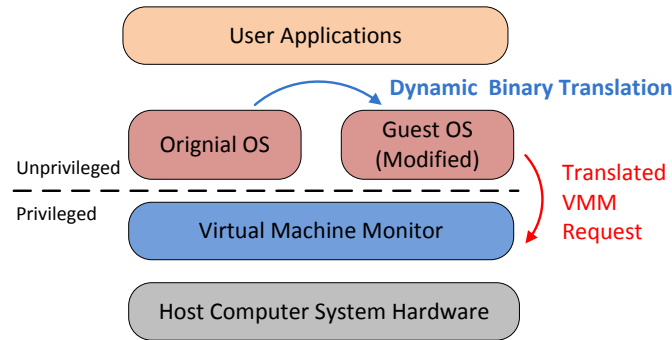


FIGURE 1.6 – With DBT technology, guest OS kernel instructions are rewritten. During execution, revised codes execute directly on processor as native codes.

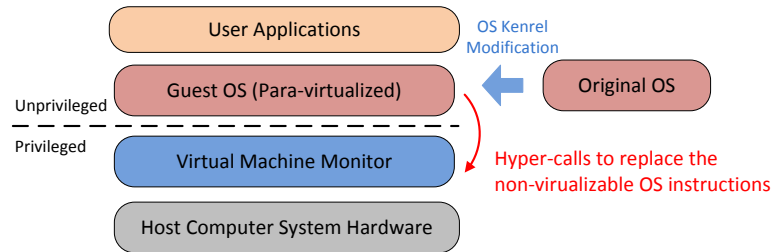


FIGURE 1.7 – Para-virtualization replaces the non-privilege sensitive instructions in OS code with hyper-calls to emulate these behaviors.

guest OS codes. For example, Windows and Linux have been widely used as the host OS for DBT virtualization.

DBT has been used on desktop and server computers, especially on Intel x86 architecture. Products such as VMware, Microsoft Virtual Server, QEMU, VirtualBox have achieved significant commercial success on the market [Hei08][dR14][Res15]. However, subjected to the limitation of on-chip resources, embedded systems did not consider this technique as topic of interest until recent years. As to the best of our knowledge, no DBT-based full virtualization has been constructed for ARM architecture.

1.1.2.3 Para-Virtualization

Though full-virtualization can provide ideal solutions, the limitations are also obvious: it is highly dependent to hardware assistance and only suitable to a determined class of architecture. On the contrast, para-virtualization is able to be built on a wider range of systems.

Para-virtualization refers to communication between the guest OS and the VMM to virtualize the architecture. As shown in FIGURE 1.7, this technique involves modifying the OS kernel to replace non-virtualizable instructions with hyper-calls that communicate directly with the VMM. The VMM also provides hyper-call interfaces for other critical

TABLE 1.1 – Comparisons of existing virtualization solutions.

Solutions	Advantages & Disadvantages
Hardware-assisted virtualization	<ul style="list-style-type: none"> • Original OS kernel can be directly hosted ; • VMM can be implemented with less effort and lower complexity ; • Virtualization efficiency can be improved by dedicated architecture.
	<ul style="list-style-type: none"> • It relies on the hardware virtualization extensions, which are limited to a determined series of processors.
DBT virtualization	<ul style="list-style-type: none"> • Original OS kernel can be directly hosted ; • No reliance on particular hardware architecture ; • Code-rewriting gains a performance benefit over the trap & interpretation mechanism ; • Hybrid and heterogeneous systems can be virtualized since DBT can translate code according to different ISAs [Hor07].
	<ul style="list-style-type: none"> • DBT takes great amount of CPU resources to translate OS codes on the fly, and is unsuitable for embedded systems ; • DBT introduces substantial complexity to VMM.
Para-virtualization	<ul style="list-style-type: none"> • No reliance on particular hardware architecture ; • Hyper-call based communication improves the VMM performance ; • This solution requires a lower VMM complexity than DBT.
	<ul style="list-style-type: none"> • Modification of OS kernel source codes causes extra development costs ; • Potential guest OS is limited to the open-source ones ; • Modified OS requires to be re-certified to avoid unexpected behavior.

kernel operations such as memory management, interrupt handling and time keeping. VMM receives hyper-calls as direct demand from virtual machines which results in a more efficient virtualization mechanism and lower-complexity VMM.

The value proposition of para-virtualization is in lower virtualization overhead, since it avoids the trapping and decoding of instructions, which are inevitable due to the mechanism of full virtualization. Unlike full virtualization, guest OS in para-virtualization is aware of being virtualized. VMM presents to virtual machines a custom interface that is similar but not identical to the underlying hardware [AP12]. Besides, the cooperation between VM and VMM makes it possible to develop more dedicated and flexible virtualization mechanisms.

Para-virtualization requires the availability of OS source code, which narrows its application to several open-source operating systems such as Linux and uC/OS [ISM09]. Another insufficiency in this domain is that there is a lack of a standardized VMM interface for virtual machines. As a consequence, guest OS has to be specifically ported on the target according to different para-virtualization solutions, and will not be supported out of the box.

We overview the current virtualization approaches by comparing their advantages and drawbacks, as listed in TABLE 1.1. Note that, one commonly-considered drawback of

para-virtualization is that it introduces higher VMM complexity than hardware-assisted solutions. In fact, hardware virtualization extensions introduce the similar complexity, but on the hardware level. Although these hardware supports manage to accelerate the virtualization process, thereby obtaining smaller VMM sizes and better security, the related design complexity is tremendous. This is illustrated by the fact that hardware extensions only appear recently and exist in limited products, while being absent in many low-end and mid-range CPUs [PKR⁺13]. In this case, para-virtualization is the most economical solution for small-scaled devices in such a domain.

1.2 ARM-based Embedded System Virtualization

Embedded systems were used to be relatively simple, single-purpose devices whose computing ability was limited. Due to the subject of resources, most embedded systems were dominated by certain hardware constraints, e.g. timing, battery capacity or memory size. Software in embedded systems normally served for dedicated functionality, like device drivers, schedulers and industry control. Real-time constraints were important to guarantee their functionality. As a consequence, simple real-time operating systems (RTOS), rather than general-purpose OSs, are used in such systems. Traditionally the software stack was pre-defined by the device vendors and was unlikely to change during execution [Hei08].

Modern embedded systems, however, are increasingly playing the role of general-purpose computers. With the improvement of computing abilities, the amount and complexity of their software are also growing. For instance, on the contemporary portable devices such as smart phones, vehicles and aircraft, there co-exist software of different types, ranging from critical tasks to high-end user applications. Each day, tons of new applications and devices, in various domains, are created worldwide. For example, as shown in FIGURE 1.8, the smart phone market is expected to grow throughout the forecast period, reaching 1.7 billion unit sales by the year 2018 [dR14]. And the total market of embedded systems is expected to reach 233 billion dollars by 2021 [Res15].

At the meantime, the growing demand on embedded devices also addresses new challenges. We should note that, compared to servers and personal computer, embedded systems are still resource-constrained. For example, battery capacity increases slowly overtime, making energy efficiency a key factor. Their memory sizes also tend to be moderate, since memory is still a cost factor for vendors. Furthermore, as embedded systems are widely used in mission and life-critical scenarios in daily life, there are high requirements on safety, security and real-time scheduling.

In order to define the main requirements and challenges for current and future embedded devices, we refer to the study carried out by European Commission DG CNECT [AP12]. In this study report, according to the involvement of embedded devices in different areas, the required characteristics are divided into four categories : critical requirements, complexity, user acceptance, and technological drivers, as listed in TABLE 1.2.

This table demonstrates that though the device requirements vary according to their usages, virtualization remains universally desired. This is due to the fact that virtual-

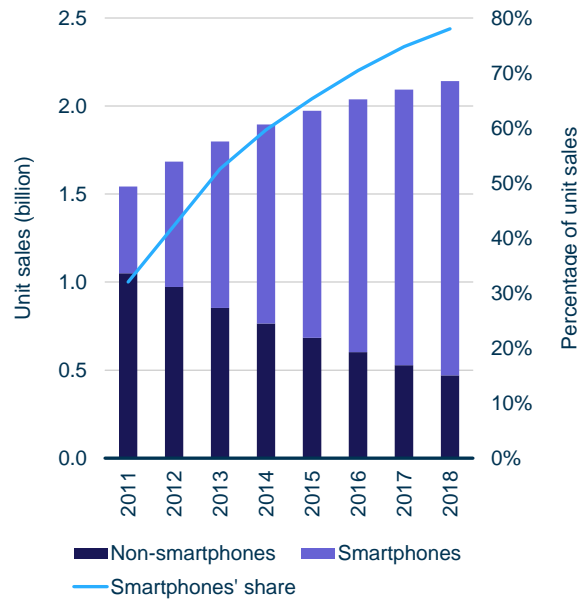


FIGURE 1.8 – Mobile handset unit sales by type, and smartphones share of unit sales, worldwide, 2011-2018 [Source : Analysys Mason, 2014][dR14]

ization can provide attractive benefits that may meet the emerging demands, which are summarized as following :

- **Security.** With the trend of open systems, operating systems are more likely to be compromised by malicious software. Virtualization provides isolated fault-tolerant environments, so that damage of attacks are minimized and limited in their own virtual machines.
- **Mixed criticality.** Virtualization is easier to meet multiple system criteria by deploying heterogeneous operating systems. For example, an embedded device may host an application OS, such as Linux, to provide better human-machine interfaces, while co-hosting a real-time OS for critical missions.
- **Multi-core support.** Virtualization is scalable when running on multiple processors. It can easily allocate processors dynamically to host poorly-scaling legacy OSs, or shut down idle processors to save energy. It also enhances the security of multi-core systems.

Nevertheless, there are still significant limitations on the usage of virtualization. One major cause is that the strongly-isolated computer model, as virtualization proposed, does not fit the environment of embedded systems [Hei08]. By their nature, embedded systems are highly integrated, where software cooperate closely with hardware resources to contribute to the overall performance. Isolating virtual machines and hardware layer may interfere the functionality of these systems.

TABLE 1.2 – Required characteristics of embedded devices for current and future applications.[Source : European Commission DG CNECT Report SMART 2009/0063] [AP12]

Features	Critical requirements			Complexity	User	Technological Drivers	
	Safety	Security	Certification	Distributed architecture	Acceptance	Multi-core & virtualization	Energy efficiency
Automotive	x	x	x	x	x	x	
Aerospace	x	x	x	x		x	x
Industrial automation	x			x		x	x
Energy consumption point		x		x	x	x	
Electricity T&D	x			x	x	x	
Healthcare	x	x	x	x	x	x	x
Communications	x	x		x	x	x	x
Consumer					x	x	

First, the applications on embedded systems require efficient data sharing. Massive data of one component should be efficiently accessed by other components. For example, video files received by a real-time virtual machine can be used by another guest OS to display on the screen. Since VMs communicate with each other through a virtual network interface, sharing bulk data will inevitably cause a waste of processor cycles and battery energy.

Second, as an essential characteristic of embedded devices, real-time scheduling is hard to guarantee in virtualized systems. The VMM schedules virtual machines as black boxes, without knowing the timing constraints within each VM. In this case, real-time activities in guest OS may miss their deadlines.

Third, the heavyweight virtual machines result in a boosting software complexity, which undermines the device robustness and limits the number of users. In fact, when virtual machine hosts a complete OS with its local software, whose complexity keeps increasing, it consumes tremendous resources, especially the memory costs, which is one key constrains of embedded devices. Addressing this issue requires a virtualization framework that is lightweight, small-sized so that it does not add to the overall complexity.

In this thesis, embedded systems are discussed in the context of ARM architectures, because ARM processors have been one of the leading CPU families in the embedded market for their low cost, energy efficiency and high performance. They have been widely applied on embedded devices, especially hand-held devices, robotics, automation and consumer electronics. Currently, ARM family is the most popular embedded 32-bit CPU. For example, according to the embedded market study carried out by UBM Tech (see FIGURE 1.9), most embedded device vendors chose ARM processors as solution for products [Tec14].

In this section, we review the existing ARM virtualization techniques by briefly introducing the principles. Since the ARM architecture is traditionally not virtualizable, we

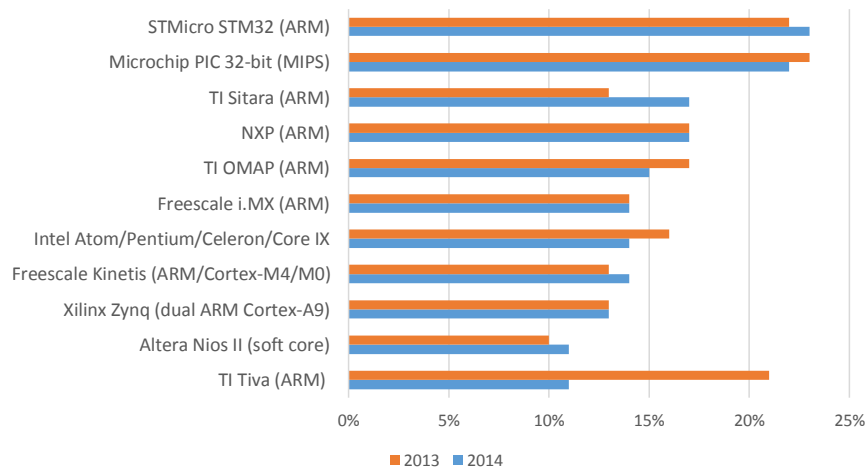


FIGURE 1.9 – Types of 32-bit processors that are preferred by embedded device vendors. The results are listed for year 2013 and 2014, based on a wide-range survey on worldwide embedded developers. [Source : UBM Tech 2014 Embedded Market Study, 2014][Tec14]

first focus on para-virtualization based solutions on conventional ARM processors, and briefly discuss the emerging hardware-supported ARM virtualization at the end.

1.2.1 Micro-kernels

Virtualized hardware abstractions can be provided by micro-kernels or hypervisor systems. Micro-kernels intend to provide a minimized virtualization layer, whereas hypervisors aim to fully replicate and multiplex hardware resources without consideration of the kernel size or complexity. Though some researchers have argued that both systems are sharing similar principles and that the borderline is declining [Hei08], these two approaches should still be analyzed separately. In this part we first introduce the micro-kernel-based virtualization.

The basic micro-kernel concept was proposed by Brinch Hansen [Han70] as a reduced nucleus with fundamental kernel mechanisms. Actual system services were supposed to be implemented in user-level servers. Micro-kernel systems follow strictly the principle of least privilege by providing the minimal set of abstractions. In 1995, based on modern computer models, Liedtke defined the three key abstractions that micro-kernels should provide : address space, threads and inter-process communication [Lie95]. The main idea behind micro-kernels design is the low kernel complexity, which results in better feasibility and security.

A micro-kernel-based VMM typically hosts virtual machines as OS-like concept of processes. It avoids the complex replica of resources. For example, a micro-kernel does not export I/O device interfaces to virtual machines. In contrast, it tends to run device drivers as separate user-level servers. However, moving these services out of the kernel into user-level implies extra communication costs. Therefore, most micro-kernel-based

virtualization works try to propose efficient inter-process communication (IPC) mechanisms.

1.2.1.1 L4 Micro-kernels

One of the initial efforts to build up micro-kernel virtualization on ARM architecture is the Fiasco L4 developed by TU Dresden [Hoh96]. Fiasco L4 was re-implemented from the famous second-generation (2G) L4 micro-kernel, which was proved quite efficient on x86 systems [Lie95]. This kernel follows Liedtke's principles and provides only basic services. Additionally, it offers good isolation characteristics to co-host multiple subsystems. A Linux kernel (L4Linux) is para-virtualized on top of the L4 micro-kernel. Other user applications are running in other virtual machines. From the micro-kernel's point of view, L4Linux and user virtual machines are L4 tasks with independent address spaces.

Fiasco L4 relies on L4Linux to support user threads in a server-client approach, as shown in FIGURE 1.10. L4Linux plays as a Linux server that provides Linux services to user threads. As clients, user thread can only communicate with L4Linux via the IPC mechanism provided by L4 micro-kernel, which is based on system call interface. Through code-rewriting, user threads generate system calls that will be passed to the micro-kernel and be redirected to L4Linux for handling. As illustrated in FIGURE 1.10, such system calls cause heavy context-switch overheads, which is a major drawback of this approach.

Under thorough discussion, the virtual machines hosted in Fiasco L4 are rather groups of user Linux threads than independent guest OSs, since they are considered as different clients sharing the same Linux server. Though virtual machines are strongly isolated from the Linux server, they are running as groups of user Linux applications being supported independently from each other. Therefore, it is unlikely to host heterogeneous OSs in this framework.

One of the major advantages of L4 micro-kernels is low complexity, which makes

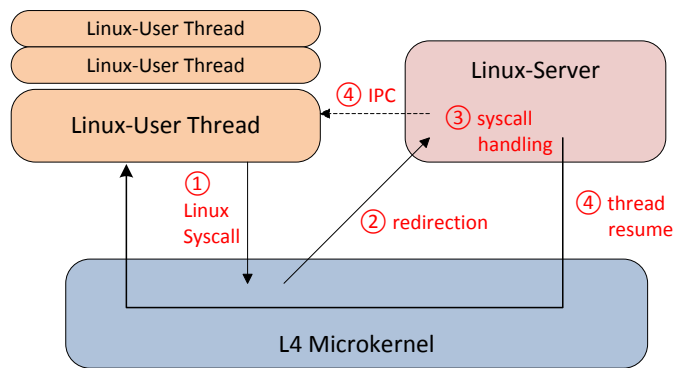


FIGURE 1.10 – Architecture of Fiasco L4 virtualization. Guest user threads are supported by Linux server following the sequence : (1) thread syscall to request Linux service ; (2) L4 micro-kernel takes syscalls and redirects it to Linux server ; (3) syscall handling ; (4) L4 returns to user thread via resuming execution context or inter-process communication.

it easy to be adapted. In [GBL⁺09], a solution dedicated to multi-core smartphones is proposed, named ICT-eMuCo. A combination of an L4 micro-kernel, a load balancer, and virtualization techniques is presented. The ICT-eMuCo solution offers the co-existence of several protocol stacks into the modem subsystem together with a pluggable Rich-OS based applications subsystem. However, ICT-eMuCo was highly customized for phone applications and thus inevitably lacks generality.

1.2.1.2 OKL4 Microvisor

The Open Kernel Lab has proposed another micro-kernel-based virtualization approach, named as OKL4 microvisor[HL10]. OKL4 is a third-generation (3G) micro-kernel of L4 heritage, and was intended for performance-sensitive memory-constrained mobile devices. The developers claim that this solution has achieved commercial success and been adopted in Motorola mobile phones. OKL4 attempts to combine the characteristics of micro-kernel and hypervisor, providing the high efficiency of hypervisors while maintaining the generality and minimality of micro-kernels. Though derived from the L4 micro-kernel concept, OKL4 has been built from-scratch and has significantly simplified the virtualization mechanism from traditional L4 kernels. The evaluation presented in [HL10] shows that OKL4 remains at low code complexity. Furthermore, OKL4 is based on seL4, which is high-a security version of L4 micro-kernel. The seL4 has been formally verified to satisfy strict isolation and information-flow control [KEH⁺09]. The features of seL4 guarantee the security of the OKL4 system.

Unlike Fiasco L4, OKL4 does not rely on L4Linux to manage virtual machines. It provides virtualized CPU and virtual MMU to provide independent execution context and address space. I/O drivers are deployed in user space as processes. VMM schedules guest OSs on virtual CPUs. OKL4 replace the complex system calls in L4 with simple hyper-calls. Guest OS kernel uses hyper-calls to communicate with micro-kernel and requires for services. Since OKL4 aborts the server-client mechanism, guest OS can be hosted independently with their own execution environment. In the actual application of OKL4 kernel on Motorola Evoke QA4 mobile phone, Linux system and the AMSS/BREW baseband stack are co-hosted in virtual machines on top of an ARM9 processor [ISM09]. The virtualization framework of OKL4 microvisor is demonstrated in FIGURE 1.11.

One significant improvement of OKL4 is its efficient IPC mechanism, which consists of virtual interrupts and channels. Virtual interrupts are used for synchronous IPC and are quite simple to implement. Channels are defined as shareable FIFO buffers mapped in user space and can be directly accessed from different users. These policies permit simplified, high performance IPC among virtual machines, without undergoing the redundant system-call-based IPC process in L4 micro-kernel.

1.2.2 Hypervisors

Hypervisors used to constitute the earliest solution for co-hosting multiple operating systems when virtualization first went into fashion in the 1970s [PG74]. Initially, hypervisors were intended to reuse legacy software stack and to execute multiple tasks

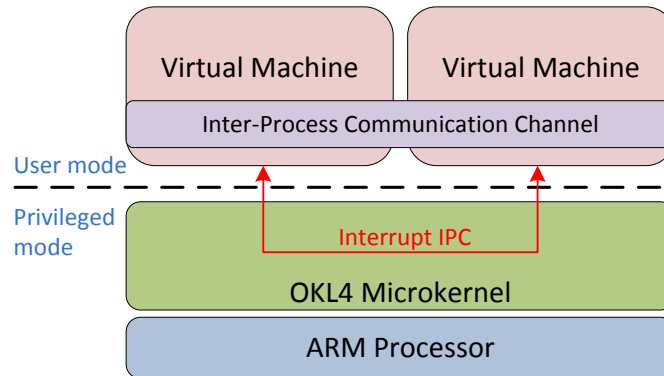


FIGURE 1.11 – The virtualization framework of OKL4 microvisor on ARM architecture.

concurrently. Since the renaissance of virtualization technology in 1990s, hypervisors have been widely employed on desktop and server computers. They pursue to provide a full abstraction of the hardware layer, which should resemble the physical platform as closely as possible. Generally, the size and complexity of kernels are not the main concerns of hypervisors. For example, hypervisors traditionally provide virtual networks as IPC for virtual machines, which are running standard network protocols. Virtual machines access the virtual network interfaces as if they were actually visiting networks. Also, hypervisors prefer to export virtual device interfaces that emulate the actual peripheral drivers (but maybe simplified).

Therefore, a large but fully-functional kernel may serve perfectly as hypervisor. However, porting hypervisors to embedded systems implies that the factors of kernel size and complexity can no more be ignored. To meet this challenge, hypervisors are driven to move closer to micro-kernels in terms of lower kernel complexity. Many hypervisor solutions on ARM architecture choose to move part of their functionality into an existing embedded OS e.g. embedded Linux, to relieve their burdens. In this part we introduce the works of hypervisor-based ARM virtualization.

1.2.2.1 KVM

Kernel-based Virtual Machine (KVM) is a commonly adopted open-source virtual machine monitor, which was originally designed for x86 hardware virtualization extensions. Based on Intel and PowerPC processors, KVM provides full-virtualization on Linux kernels. In 2011, the authors of [DJ11] released a revised KVM version that was adapted to ARMv5 architecture, which is denoted as KVM/ARM. The motivation of this work is to provide para-virtualization solutions to the increasing Linux-based distributions targeting embedded ARM-based devices.

KVM resides as an additional loadable module in Linux kernel, so that it can take advantage of the existing Linux functions to host virtual machines. The virtualization execution path is illustrated in FIGURE 1.12. KVM maintains the execution context of virtual machines, and provides an abstraction layer to emulate physical resources.

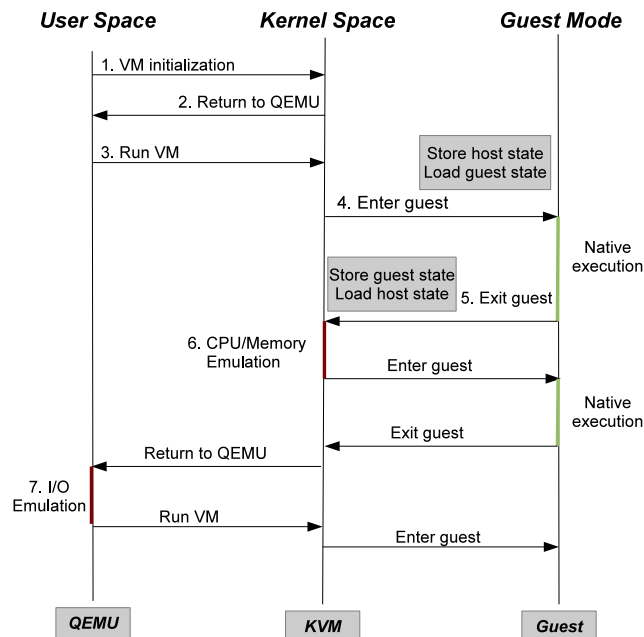


FIGURE 1.12 – The virtualization path of KVM virtual machines. KVM plays as an intermediate layer that receives the request of emulation from virtual machines, and redirects them to either QEMU or to the Linux kernel [DLC⁺12].

Meanwhile, Linux is in charge of most VMM functionality. Linux kernel performs the VM scheduling, manages the memory allocation of VMs, and provides the actual physical device drivers. Furthermore, KVM utilizes QEMU, in Linux user space, to create the virtual machines and to emulate the I/O drivers. In summary, KVM monitors virtual machines, collects their traps and relies on Linux kernel for emulation. As a consequence, KVM always keeps a low complexity.

KVM's low complexity and high integration with the Linux kernel provides the advantage that it can easily be included in current embedded Linux kernels on ARM. This is also the reason why numerous researches have been carried out to augment KVM/ARM. For example, in [DLC⁺12] another variant of KVM, the ARMvisor, was ported to ARM Cortex-A8 architecture. It focused on a simplified lightweight memory virtualization model to replace the traditional costly model in KVM. The evaluation presented in [DLC⁺12] showed a significant performance increase.

Nevertheless, since KVM and the Linux kernel are inseparable as VMM, the size of privileged components can be huge. A large-sized trust computing base (TCB) may expand the threats that system is exposed to, which will undermine the security level. Besides, it might be too expensive to implement this solution on memory-constrained embedded systems.

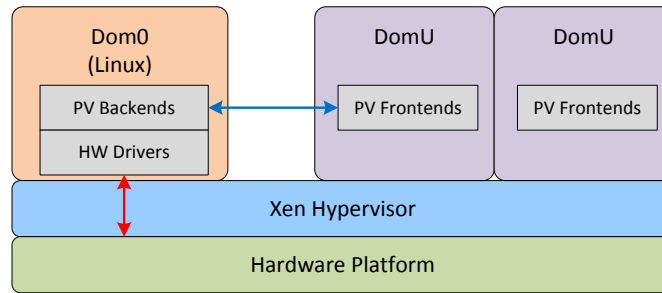


FIGURE 1.13 – The architecture of Xen-on-ARM hypervisor.

1.2.2.2 Xen

The Xen hypervisor is another para-virtualization technology that was originally designed for x86 platforms. As a popular virtualization solution on traditional computers, Xen’s application to ARM architecture is a popular topic in the domain of embedded systems [Seo10]. The mainline ARM-based Xen hypervisor was proposed by Hwang et al. [HSH⁺08], which was denoted as Xen-on-ARM, targeting the ARM-based secure mobile phones. Unlike KVM, Xen does not leverage the existing operating system as host OS. Instead, the Xen VMM runs directly on the hardware and manages virtual machines, including IPC, memory management and scheduling. Xen on ARM also removes the need of QEMU because it does not do any emulation. It accomplishes the goal by using para-virtualized interfaces for I/O devices [XEN14].

The architecture of Xen-on-ARM hypervisor is depicted in FIGURE 1.13. Xen VMM is at the most privilege level, while everything else in the system is running as a virtual machine on top of Xen, including Dom0, the first virtual machine. Dom0 is created by Xen, is privileged and drives the devices on the platform. Xen virtualizes CPU, memory, interrupts and timers, providing virtual machines with one or more virtual CPUs, a fraction of the memory of the system, a virtual interrupt controller and a virtual timer. Xen keeps peripheral devices in Dom0, which is typically Linux, but could also be FreeBSD or other operating systems. Dom0 runs the same device drivers for these devices that would be used on a native execution. Other virtual machines, called DomU in Xen terminology, gets access to a set of generic virtual devices by running the corresponding para-virtualized front-end drivers, which will pass the demand to the para-virtualized backend drivers in Dom0 for handling.

Xen-on-ARM is significantly simplified from its x86 version, and ends up with a cleaner and small framework. Several further researches on Xen focus on improving its adaptivity and performance. One well-known work is EmbeddedXEN proposed by Rossier [Ros12]. This novel kernel is able to support third-party guest OS with full privileges, instead of the limited authority in classic virtual machine systems. In this approach guest OS can be para-virtualized with less modification and lower overheads. One of its drawbacks is that it can host only two virtual machines, Dom0 and DomU. And it is based on the assumption that the third party OS can be fully trusted, which may

threaten the system in actual application.

1.2.3 ARM-based Full Virtualization

In the previous discussions, we provided an overview of the existing mainstream para-virtualization solutions for traditional ARM architectures, i.e. ARM v5/v6/v7. In this part, we introduce several researches that have explored the ARM-based full virtualization technology based on the hardware extensions.

Beginning from ARMv7A, the hardware virtualization extensions brings new possibilities of virtualization. One of the earliest attempts was the revised OKL4 released in 2011 [VH11], which was capable of full-virtualization based on new ARM features. However, due to the lack of actual device, OKL4 was developed and evaluated by using the ARM Fast Models simulator. Therefore only estimated performance evaluations were obtained from this work. Another early research of full virtualization solution, CASL-Hypervisor [LCC13], was also developed based on System C simulations. The actual device-based development began after the release of Cortex-A15 processor. Both KVM/ARM and Xen have updated their technologies to provide full virtualization supports by exploiting new features [XEN14][DN14]. Evaluation results from these works have proved that by exploiting the extensional ARM features, better virtualization efficiency and lower complexity can be achieved.

Other researchers take advantage of the special security technology of ARM, *TrustZone*, to implement full-virtualization. The *TrustZone* technology [Xil14a] refers to security extensions implemented in recent ARM processors e.g. Cortex-A5/A7/A8/A9/A15 and the newly-released Cortex-A53/A57. This hardware extension virtualizes a physical core as two virtual cores, providing two completely separated security domains, the *secure world* and the *non-secure world*. A new *monitor mode* is introduced. The *TrustZone* technology is shown in FIGURE 1.14. This feature is leveraged in [POP⁺14] to support virtual machines by hosting a guest OS in each security domain. It utilized the *secure world* for critical real-time OS (FreeRTOS) and the *non-secure world* for general-purpose OS (Linux). VMM is implemented in the monitor mode and has full view of the processor. In this case, guest OSs execute on original privilege modes and requires no modification. Furthermore, the dedicated secure world facilitates the VMM to manage virtual machine resources, and results in small VMM size. However, this approach can only host two virtual machines concurrently, limited by the number of separate worlds. Therefore, in [Bau15] the authors employed this technology on Cortex-A15, by using the virtualization extensions to host multiple virtual machines by using *TrustZone* to host security/safety applications, as shown in FIGURE 1.15.

However, we should note that traditional ARM processors are still employed in most embedded devices currently, and may remain as mainstream for their relatively low cost. Especially in the domain of CPU-FPGA SoC, to the best of our knowledge, the hardware virtualization extensions are still unavailable. Therefore, in our work, we have decided to only focus on para-virtualization solutions on conventional embedded systems. Further details of ARM-based full virtualization technologies will not be presented in this paper.

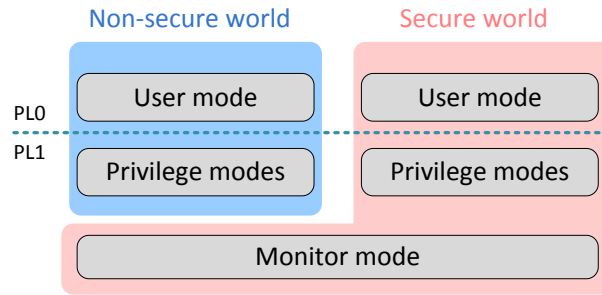


FIGURE 1.14 – The TrustZone technology in ARM processors.

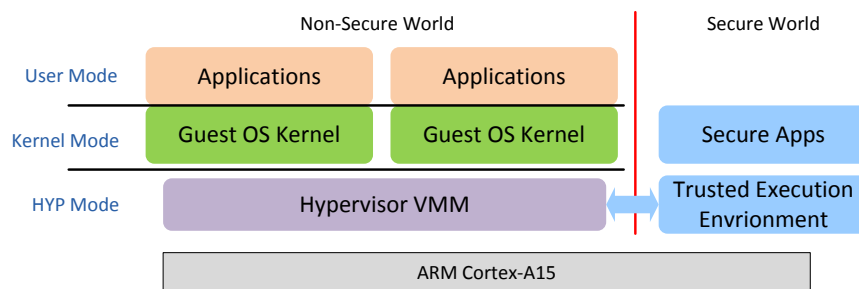


FIGURE 1.15 – Using hypervisor mode and security extensions to build up virtual machine systems.

1.3 Real-time Scheduling for Virtual Machines

For embedded system virtualization, the support for real-time systems has been in high demand, since real-time missions are critical for safety and security. However, the timing constraints of real-time systems are not easily applicable to virtualization. In typical virtual machine systems, a VMM considers virtual machines as black boxes. In this case, the VMM may fail to allocate the processor to guest OSs satisfying their real-time requirements if it has no specific knowledge about tasks inside virtual machines. For example, in round-robin inter-VM scheduling, a guest OS with unfinished real-time tasks may be scheduled out by other virtual machines and therefore miss the deadlines. Hence, to respect real-time constraints of virtual machines, it entails an extensional VMM scheduling framework which is aware of the urgency of individual guest OSs. To simplify the problem, the related researches are based on the assumption that there is no dependency among guest OSs or real-time tasks.

The scheduling of virtual machines preserves typically the characteristics of hierarchical scheduling framework (HSF) [FM02], which supports CPU resource sharing by introducing the concept of components, which are clusters of internal applications. Scheduling is divided into two layers : the system scheduler that schedules components, and component schedulers that are in charge of local scheduling within application clusters. In the context of virtualization, the system scheduler refers to the VMM's scheduler ; and

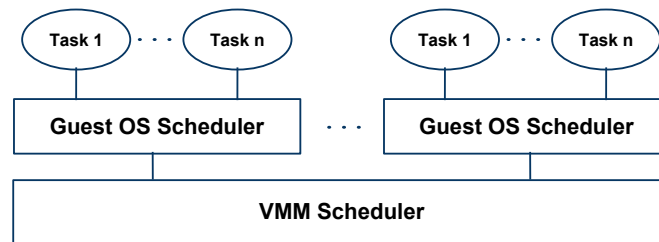


FIGURE 1.16 – Hierarchical scheduling framework in virtual machine systems.

each component corresponds to a virtual machine running on top of VMM, generally a guest OS, which hosts several local internal tasks or threads. The HSF in virtual machine systems is depicted in FIGURE 1.16. We define the scheduling performed by VMM as the *inter-VM scheduling*, and the scheduling of tasks inside virtual machines as *intra-VM scheduling*.

The real-time schedulability for hierarchical systems is addressed theoretically in numerous research works [MA01][LB05][CAA09][KZ09]. Many of these researches have focused on the schedulability with minimal CPU bandwidth. To achieve this goal, in their proposed scheduling strategy, the details of component scheduler are monitored and interfered by the higher-level scheduler, ending up with integrated scheduling models. However, such approaches are not suitable for virtual machine system, since it requires for more generalized scheduling where different layers are more decoupled. In this section we take an overview of current real-time scheduling strategies in virtualization.

1.3.1 Compositional Real-time Scheduling

Presented by Shin and Lee [SL04] in 2004, the *Compositional Real-time Scheduling Framework* (CSF) has been widely exploited in real-time virtual machine systems. This scheduling model manages to meet the desired features of VMM scheduling : (1) the high-level scheduler should not access the component internals and should operate only on component interfaces ; (2) schedulability of a component’s workload should be analyzed independently and be informed to the system ; (3) high-level scheduler analyzes the real-time constraints based on component urgencies.

This compositional scheduling framework is based on the *periodic resource model* (PRM) proposed by the same authors in [SL03]. In this model the authors define real-time application as a combination of periodic tasks that exhibit a periodic behavior. Therefore, by abstracting the real-time application with a workload model, it can be considered as a single periodic task. Based on this periodic task model, schedulability can be directly analyzed using classic scheduling algorithms. The periodic model is characterized as $\Gamma(\Pi, \Theta)$ to describe a partitioned resource whose availability is guaranteed for Θ time units in every Π units. In other words, Π describes the resource period, and Θ defines the demand bound of this resource.

FIGURE 1.17 reveals how PRM is employed in a hierarchical scheduling framework.

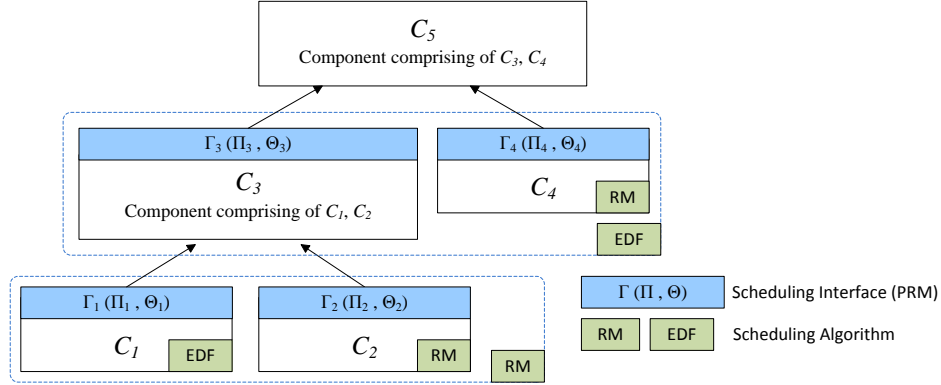


FIGURE 1.17 – Compositional scheduling framework in hierarchical systems. Periodic resource model is used to abstract component real-time constraints and establish scheduling interface between layers. Different scheduling algorithms can be used for different components and layers.

As described in the earlier section, we define components as basic schedulable units. Each component is composed of internal workloads and schedules them according to independent internal schedulers. Denoting a component as C , it can be described as a triple (W, R, A) . W stands for the workloads (tasks or missions) included in the component. R is a resource model describing the available resources. A is the scheduling algorithm which component scheduler uses to schedule workloads. By using PRM, the collective real-time requirements of component C are abstracted to be a single real-time requirement Γ , which is called the *scheduling interface*. C is then described as $C(W, \Gamma, A)$. An optimal scheduling interface Γ of a component can be obtained by minimizing the resource bandwidth for W [SL04]. Once a child component C_1 calculates its *scheduling interface* Γ_1 , it passes Γ_1 to its parent component, which treats the scheduling interface as a single workload T_1 . By satisfying the timing constraints of T_1 , it is guaranteed that the resource demand of child component C_1 can be met. In this case, the child component can be scheduled without revealing its internal details, e.g. the number of tasks and local scheduling algorithm.

Since each component schedules independently, compositional scheduling framework is feasible to implement multiple scheduling strategies concurrently. For example, *hard real-time*¹ tasks can be collected into a component with the *Earliest Deadline First* (EDF) strategy so that all the deadlines can be met, while *soft real-time*² tasks can be scheduled in another component with *Rate-monotonic* (RM) scheduling.

1. Hard real-time and soft real-time systems are distinguished by their criticality and policy [Kri99]. Hard real-time tasks are generally safety-critical, whose overrun in response time leads to potential loss of life and/or big financial damage, e.g. ABS controllers in vehicles. For hard real-time systems, a missed task deadline results in the failure of the whole system.

2. Soft real-time tasks are the ones whose deadline overruns are tolerable, but not desired. There are no catastrophic consequences of missing one or more deadlines. Missed tasks can be delayed to execute

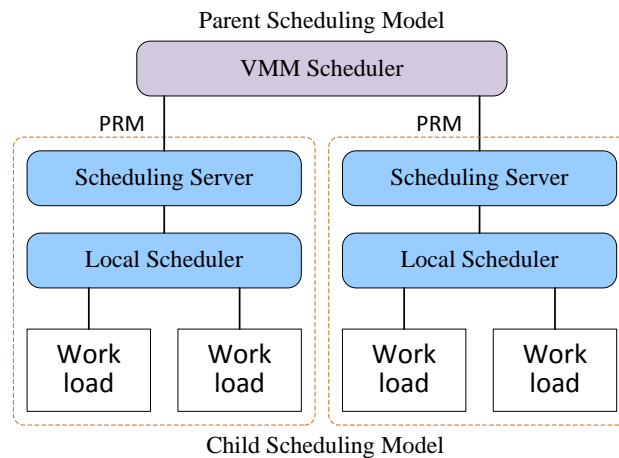


FIGURE 1.18 – General implementation of compositional scheduling framework in virtual machine systems.

In FIGURE 1.18 we present a general idea of implementing CSF in real-time virtual machine systems. The local scheduler inside guest OS is generally extended with a *scheduling server* that calculates the PRM for its tasks or threads according to the scheduling algorithm. This periodic interface should then be passed to the VMM via hyper-calls or IPC, as a collective timing requirement of guest OS.

Another issue in the implementation is the calculation of scheduling parameters. In most embedded systems where scheduling is based on timer ticks, parameters are expressed in integer numbers, which undermines the accuracy of scheduling. This factor, denoted as the quantization overheads, should be fairly discussed. In the following, we introduce several implementations of CSF in existing virtualization technologies.

1.3.1.1 L4/Fiasco Micro-kernel with Compositional Scheduling

Compositional scheduling framework has been explored on the Fiasco L4 micro-kernel in [YKP⁺11]. The authors revised the L4Linux server to support the periodic resource model in real-time scheduling. Linux servers, on which real-time tasks are running, collect the internal tasks' timing requirements and calculate the PRM interface. This interface is then passed to the L4 micro-kernel for scheduling.

As described in Section 1.2.1.1, L4 hosts guest applications in a client-server mechanism, where L4 micro-kernel schedules the L4Linux threads as clients. To facilitate the scheduling, L4Linux internal tasks are mapped to corresponding *shadow threads* that can be directly waken up by L4 micro-kernel. Furthermore, several L4Linux kernel features are also implemented as threads, such as the Linux kernel thread, the timer interrupt

or aborted according to different policies. One typical soft real-time system is the user interface in smart phone OS, whose response time directly determines the user experience and should be answered as fast as possible. Soft real-time scheduling is usually evaluated by the criterion *Quality of Service* (QoS).

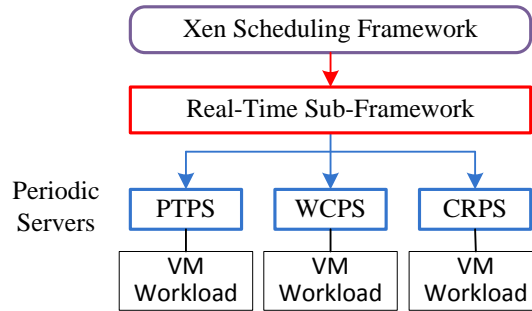


FIGURE 1.19 – The scheduler architecture of RT-Xen.

thread, and the idle thread. Thus, from the viewpoint of L4 micro-kernel, one L4Linux is an assembly of multiple L4 threads (and *shadow threads*). The L4Linux server, as the collection of L4Linux threads, calculates the PRM scheduling interface and communicates with the L4 micro-kernel. In this case, multiple L4Linux servers are scheduled as periodic interfaces.

This approach significantly benefits from the homogeneous architecture of L4 systems. By utilizing shadow thread mechanisms, real-time tasks in virtual machines can be interpreted as L4 threads and directly scheduled by the L4 micro-kernel. This also makes a contribution to the overall performance.

1.3.1.2 RT-Xen

The original Xen-ARM provides support for the real-time guest OS by the *Simple Earliest Deadline First* (SEDF) scheduler. To use SEDF, each guest OS, at first, has to provide its own scheduling parameter (period, execution slice, relative deadline) to Xen VMM, then VMM sorts the guest OSs by their given deadlines and selects the guest OS that has the earliest deadline. So, SEDF guarantees real-time scheduling among guest OSs with inter-VM schedulability. However, SEDF cannot guarantee task execution inside a guest OS, i.e. the intra-VM schedulability.

Therefore, in [XWLG11] and [LXC⁺] an extended Xen-ARM hypervisor, RT-Xen, was proposed. RT-Xen is complemented with a *compositional scheduling architecture* (CSA), which is built on the CSF model to support hierarchical real-time scheduling. The contributions of RT-Xen can be concluded as : first, it implements CSF in Xen hypervisor scheduler by introducing PRM scheduling interface. Second, it proposes several novel periodic servers to implement PRM interfaces. Third, it provides an algorithm for computing the optimal PRM interface for quantum-based platforms under RM scheduling.

FIGURE 1.19 describes the scheduler architecture of RT-Xen. The Xen hypervisor scheduler, as the root scheduler, schedules virtual machines according to their PRM interfaces. PRM interfaces can be built up based on three different periodic server policies : *Purely Time-driven Periodic Server* (PTPS), *Work-Conserving Periodic Server* (WCPS) and *Capacity Reclaiming Periodic Server* (CRPS). These policies differ in their strategies on the handling of idle budget during execution. PTPS simply focuses on the consump-

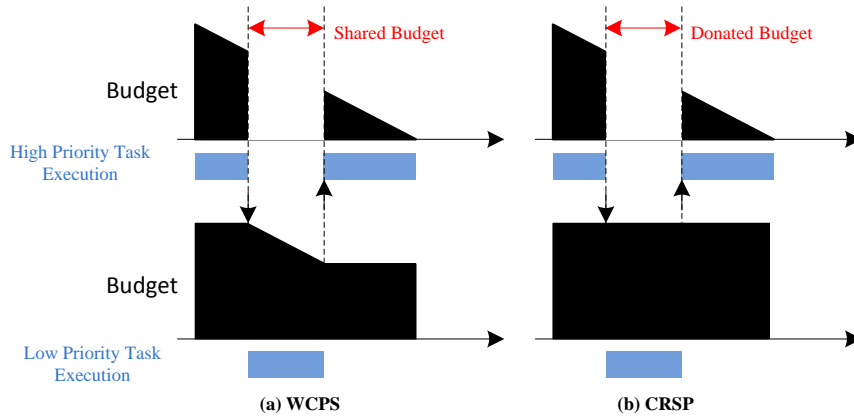


FIGURE 1.20 – Execution of servers with the WCPS and CRSP policies.

tion of budget, not caring whether it is at idle state. On the contrast, WCPS allows its idle budget to be used by lower priority components, as shown in FIGURE 1.20(a). CRSP resembles WCPS, excepting that it donates the idle budget to other component as extra resource, as shown in FIGURE 1.20(b). In this case, the receiver can potentially finish its task earlier, resulting in an overall improvement in task response time than the other two policies. To unify the operations on different servers, an intermediate layer, *real-time sub-framework* is inserted as general interfaces.

The intention of these server policies is to provide a better software real-time performance. All scheduling parameters in the PRM interface are given in quantized numbers of timer ticks according to the scheduling timer in Xen implementation. To simplify the problem, RT-Xen assumes that the timer tick is small enough to perform accurate scheduling, and ignores the changes of actual CPU bandwidth allocation caused by the quantization overheads.

1.3.2 Other Real-time Approaches

Besides solutions with compositional scheduling framework, there are several hypervisors that focus on alternative approaches. Most of these works are attempting to extend the existing virtualization technologies to real-time domain.

In [YY14], the Xen-ARM hypervisor is enhanced via another approach to compensate the drawback of CSF, which is the quantization overheads caused by the implementation of scheduling ticks. The authors proposed a novel scheduling algorithm, denoted as *SH-Quantization*, which specially takes the quantization overhead into account. This algorithm calculates the scheduling parameter pair (Π, Θ) that meets the intra-VM timing constrains. The parameters (the period Π and the execution slice Θ) are selected for the minimum *effective CPU bandwidth* (EBW) which measures the integer quantized execution demand in real implementation. In implementation, the authors virtualized a real-world RTOS, $\mu\text{C}/\text{OS-II}$, as guest OS on Xen-ARM. The scheduler of $\mu\text{C}/\text{OS-II}$ is modified as the following : first, a dedicated interface is provided to $\mu\text{C}/\text{OS-II}$ to request

real-time scheduling using the the *SH-Quantization* algorithm; second, $\mu\text{C}/\text{OS-II}$ has to change its scheduling parameters via additional hyper-calls. According to the evaluation results, *SH-Quantization* algorithm managed to improve the hypervisor’s ability to support real-time OS. However, we should note that this technology involves heavy workload for the modification of the guest OS source code, for in this approach the guest OS scheduler has a strong dependency for the VMM hypervisor.

In [BSH11], Heiser presented the L4-type third generation micro-kernel seL4 on ARM Cortex-A8, which has been through a formal machine-security check. SeL4 not only proposes strong-isolated software stacks, but also defines several behavior patterns to eliminate potential threads. For example, it never stores function pointers at run-time, so call jumps can be resolved statically. According to the authors, seL4 is extensively proved as it will never crash or perform unsafe operation based on its well-controlled behaviors. Based on this feature, seL4 was then further claimed to be efficient enough to support hard real-time systems. To confirm this conclusion, the authors proposed a simple pessimistic pipeline model to simulate the Cortex-A8 instruction latency. Given that the software behaviors are quite deterministic in seL4, with the pipeline modeling, the *worst-case execution time* (WCET) of tasks can be accurately estimated. Furthermore, since the analysis is sound and statistic, the computed overheads can be confidently used as safe upper bound for pre-determined real-time scheduling. This feature is denoted as protected hard real-time. However, currently there is no actual implementation of real-time virtualization based on this concept, which makes this approach hard to evaluate.

Besides the technologies introduced above, there are other real-time researches targeting other hypervisors like KVM [CAA09] and PikeOS [Kai08]. However, all these frameworks rely on the modification of the guest OS scheduler so that intra-VM scheduling can be taken into account. This approach undermines the generality of the framework, for the developers have to deal with the inner algorithm of guest RTOS. Furthermore, most of these works require for extra computations of scheduling models. For example, in the widely-used CSF scheduling, computation of PRM interfaces or extra scheduling servers have to be added. In our work, we attempt to propose a scheduling framework which respects real-time OS constraints with minimized additional mechanism, so that RTOS can be supported without any modification on their original scheduler.

1.4 CPU-FPGA Hybrid Architecture

On conventional computing systems, *Application Specific Integrated Circuit* (ASIC) are preferred by embedded device vendors for their higher performance and power efficiency for specific scenarios. However, since the processor complexity and computing workloads are significantly increasing, so is the cost of designing and manufacturing ASICs. Driven by this fact, an appropriate substitute is in high demand by device vendors.

Meanwhile, Field Programmable Gate Arrays (FPGA), as classical programmable device, has proved to be advantageous for accelerating complex computing. More importantly, FPGAs permit the developers to freely implement and optimize their algorithms.

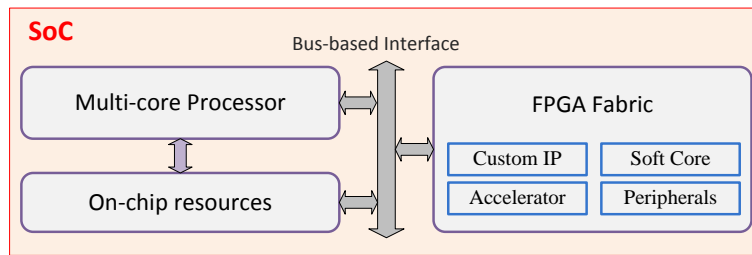


FIGURE 1.21 – The general architecture of CPU/FPGA hybrid processors, with CPU and FPGA being implemented in independent packages.

In fact, FPGA devices continue to gain popularity as the cost of reprogrammable logic gradually declines. However, what prevents FPGA to be widely employed as conventional ASIC processors is that FPGA is unable to use legacy software stack. Most FPGA devices meet this challenge by shipping synthesized CPU core to the programmable-logic fabric as *hard processors*. In past years, Altera and Xilinx, the leading FPGA vendors, have offered a few FPGAs with hard CPU cores baked into the chip, such as PowerPC inside the Xilinx Virtex family [Fle05]. The variety of these hybrid devices is quite narrow, however, and the performance of hard processors is comparatively poor, even in the fastest and most expensive FPGAs.

Nevertheless, to meet the demand for an ASIC substitute, companies keep striving to find the convergence point of conventional CPU and FPGA computing. The latest attempts are from Xilinx, Altera and Intel, who are crafting new ways to combine CPU cores with programmable logic. Instead of using synthesized CPU core implemented in FPGA fabric, these approaches provide System on Chip (SoC) architectures where CPU and FPGA domains are independently implemented and tightly connected by on-chip bus and inter-connections. CPUs dedicated for embedded system are chosen in these devices. Xilinx released such CPU/FPGA hybrid platforms in the Zynq-7000 series, where a dual-core ARM Cortex-A9 processor is integrated with 7-series FPGA fabric [Xil14c]. ARM was also introduced in Altera family of Cyclone-V and Arria-V FPGAs [Alt15]. Intel's solution is the Intel Atom processor E600C Series, which pairs an Intel Atom processor SoC with an Altera FPGA in a multichip package [Kon12]. In this approach, a single-core dual-threaded Intel Atom processor SoC and an Altera midrange FPGA, Arria-II, are bonded side-by-side and linked over a PCI interface. Recently, Intel has taken a further step by releasing a Xeon/FPGA platform dedicated for the Data Center [Gup15]. In FIGURE 1.21 we briefly depict the general architecture of these platforms.

CPU-FPGA hybrid processors inherit the advantages of both sides. On one hand, the high-end general purpose processors are capable of establishing complex computer systems, and existing software stack can be directly shipped without obstacle. On the other hand, the adoption of FPGA accelerators offers a compelling improvement in performance when performing intensive computations. Moreover, with the help of CPU processing, FPGA accelerators can be managed much more efficiently with higher-complex

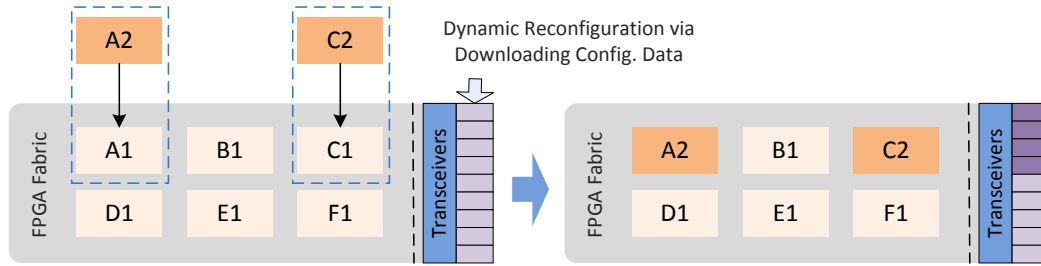


FIGURE 1.22 – Dynamic Partial Reconfiguration permits users to modify a given area of FPGA circuit on-the-fly, while the rest fabric functions normally.

strategies, which will inevitably enhance the acceleration. Considering that FPGAs are already playing important roles in enterprise, cloud computing, datacenter, network, and *high performance computing* (HPC) markets, it can be foreseen that a significant portion of CPU workload in these domains will be shifted from CPUs to FPGAs over the coming years, which may result in a performance jump.

Furthermore, the software stacks for programming FPGAs have greatly evolved over years to simplify the FPGA programming. For example, Altera in particular has grafted support for FPGAs onto the OpenCL development environment [OBDA11] to ease the design flow, and Xilinx has also released Vivado *High-Level Synthesis* (HLS) for a faster path to IP creation [Xil14b]. With these technologies, it is even easier for software workloads to be shipped to programmable logic for acceleration.

While being considered as quite promising, CPU-FPGA systems also bring up new challenges. One major challenge is how to efficiently offload software tasks to the FPGA side. FPGA resources can either be accessed as flexible input/output peripherals or be considered as co-processors with local workloads. With different strategies, scheduling, sharing and security mechanisms should be carefully discussed. There have been researches attempting to extend the existing traditional CPU-only techniques to CPU/FPGA hybrid systems, to fully exploit the mainstream FPGA computing. In this section, we will first introduce the main characteristic of current mainstream FPGAs, the *dynamic partial reconfiguration* (DPR) technology, and then we will discuss the possibility of building virtual machine systems on this architecture.

1.4.1 Dynamic Partial Reconfiguration

The DPR technology has been a trending topic during the last decade [BHH⁺07], which has been included in the recent mainstream FPGA vendor devices, such as Xilinx Virtex family and Altera Stratix family. DPR is a technology breakthrough for FPGA devices. For traditional FPGA reconfiguration computing, one of the major drawbacks is the lack of flexibility, because the whole fabric is required to be reconfigured even when modification is required for part of FPGA. As a consequence, even a partial update or modification of hardware functionality ends up with enormous time overhead and power consumption. As a solution, DPR permits users to reconfigure particular areas of an

FPGA while the rest continues executing, as shown in FIGURE 1.22. By pre-compilation, certain areas of the FPGA fabric can be defined as reconfigurable. In other words, gate arrays in these areas can be re-programmed by receiving commands and synthesis information of alternative modules. The replaceable modules are synthesized during the hardware compilation. Depending on different device families, different reconfiguration ports and synthesis tool chains are employed.

This technique is proved to be quite prospective for embedded systems. Compared to static fabric, DPR benefits from the following major advantages :

- **Enhanced dynamic resource allocation and re-utilization** : Users can implement more complex algorithms breaking them down into smaller mutually exclusive modules. In this case, constraints regarding the chip size can be easily met.
- **Improved develop efficiency** : The time-consuming synthesis and mapping are only required for the modified function, while the remainder of the design is fixed.
- **Security** : Reconfigurable accelerations are isolated from each other, with unified interface, which contributes to a higher error tolerance.
- **Lower power consumption** : DPR reveals a better power efficiency in terms of performance per watt than traditional solutions [TCL09].

However, DPR technology still suffers from expensive reconfiguration overhead, which remains a crucial issue in practice [McD08]. In modern high-end FPGAs which may have tens of millions of configuration points, one reconfiguration of a complex module will be very time-consuming. Especially in a computing-intensive system, where several mutually exclusive components are sharing reconfigurable resources, the time lost on reconfiguration will severely degrade the overall performance [HD10]. Therefore, a dedicated efficient management associated to a high data transfer bandwidth for configuration is essential in DPR systems.

Exploitation of DPR has been under massive researches for the last decade. Though these researches covered various domains e.g. efficient reconfiguration framework [LKLJ09] [HGNB10] and power consumption optimization, the major effort of society is made to provide efficient management of DPR resources in computing systems. In the following part we introduce the related works focusing the DPR management in the context of CPU/FPGA architecture.

1.4.2 CPU/FPGA Execution Model

Due to the nature of heterogeneous architecture, one major challenge for CPU/FPGA systems is the cooperation and coherency between software applications and hardware accelerators. With software/hardware applications executing in parallel, the classical software-related issues such as task scheduling and resource sharing are extended to the FPGA side. In fact, from the viewpoint of software developers, these challenges can be abstracted as how to map reconfigurable resources to software space. In FIGURE 1.23 we have classified the model types of existing approaches according to their strategies,

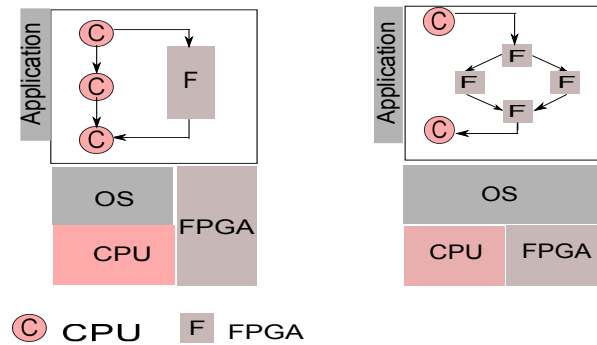


FIGURE 1.23 – The concepts of two CPU/FPGA models : *offload model* and *unified model*. In *offload model*, application is executed on CPU cores and FPGA is used as an accelerator. In *unified model* the application is divided in parts and mapped to CPU and FPGA. OS helps in seamless execution of application parts in the hybrid system.

namely *offload model* and *unified model* [BPS15].

In the *offload model*, OS or bare-metal software applications are executing in CPU and DPR resources are used as accelerators. In this case, DPR resources can be fully exploited since they are directly accessed and programmed by applications. However, it also undermines the generality as the processing of DPR accelerators are exposed to software users.

In the *unified model*, on the other hand, CPU and FPGA are unified via well-defined interfaces. The OS is in charge of the workload allocation and migration between both resources. In some cases, a middleware is implemented inside the OS. Thus, with the help of library calls, applications can make use of both resources as OS services, without knowing the actual implementation at the physical layer. This model abstracts the CPU/FPGA platform so that user applications have better performance, but also require complicated scheduling and allocation mechanism.

1.4.2.1 CPU/FPGA offload model

Offload model is often used for bare-metal applications or simple OS, whose usage scenarios are relatively simple and single-purposed. These systems require applications to fully control the behaviors of accelerators, including computation and reconfiguration. Researches for these system focus on faster reconfiguration path [HGNB10] and efficient partial reconfiguration controller, as they are critical for the overall performance. One typical approach is ZyCAP [VF14] based on ARM/FPGA system, which was proposed as an efficient partial reconfiguration controller. This controller provides to software users an interface that permits the overlapping of software execution and hardware partial reconfiguration. Furthermore, this approach proposed a high reconfiguration throughput, by enhancing the ICAP interface with high-bandwidth *Direct Memory Access* (DMA). The measured reconfiguration throughput turned out to be much higher than the default

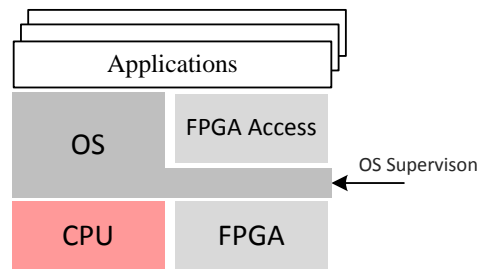


FIGURE 1.24 – With DPR resources used by OS or multiple users, the offload model should be modified. Because applications/threads still access FPGA accelerators as peripheral devices, the OS kernel has to supervise their accesses as an intermediate layer.

Processor Configuration Access Port (PCAP).

Offload model is also adopted by some well-developed OSs e.g. Linux, to support DPR resources. This is because the *offload model* considers DPR modules as separate accelerators that can therefore easily extend an OS as devices without heavy-cost source code modification. In this case, researchers only need to focus on the mechanism to share and reconfigure DPR accelerators as peripherals. In the work of [HH09], an embedded PetaLinux kernel was extended to include DPR modules into the device tree, and was mapped to the user applications through device nodes. From the user viewpoint, DPR resources are acting as peripheral devices, whose sharing and allocation, however, were actually controlled by the Linux kernel. However, since an OS often hosts multiple users, one common problem is the sharing of FPGA accelerations among different clients. In this case, the OS should be in charge of the allocation of accelerators, which should be implemented either via the mutex/semaphore mechanisms or via accelerator virtualization. As a consequence, the execution model is no longer the classical *offload model* since software applications must use accelerators under the supervision of the OS kernel, as shown in FIGURE 1.24. This requires a more complex mechanism, for instance, the hardware virtualization technology. This will be explained in the later section.

1.4.2.2 CPU/FPGA unified model

Unified model is based on the theory that in DPR systems, the programmable logic fabric can be interpreted as several engine containers where multiple hardware accelerators can be hosted in a time-multiplexed sharing strategy. In this case, the model of *multi-kernel multi-threads* [RVdlTR14] is often used to study DPR architecture [PG11][KBT08]. This model defines a FPGA as a group of computing agency with multiple hardware threads, as shown in FIGURE 1.25. Based on this concept, a CPU can process hardware computations as schedulable hardware tasks.

This model requires dedicated OSs that manage the hardware threads and hide technical details from the users. As introduced previously, the OS should be platform-specific or a deeply-customized version of existing OSs. With uniform interfaces and APIs, multi-tasking between hardware/software tasks can be supported on OSs such as Rainbow OS

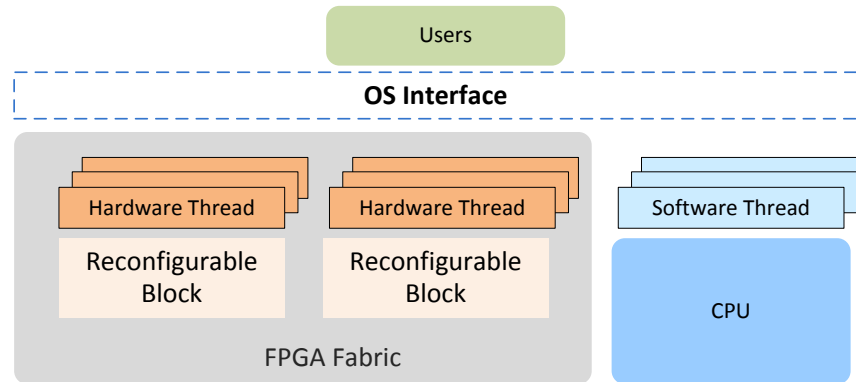


FIGURE 1.25 – Dynamic Partial Reconfiguration architecture modeling as a group of computing agents with multiple hardware threads.

[JHE⁺13] and Hthreads [APA⁺06]. One well-known approach in this area is ReconOS proposed in 2009 [LP09]. ReconOS was built as a revised version of the widely used real-time OS eCos. In this approach, the functionality of an application is divided into software threads on the CPU side and parallel hardware threads, which are mostly data computation tasks. To provide symmetry between software and hardware tasks, for each hardware thread, a new dedicated eCos thread, denoted as the *delegate thread*, is dynamically created and connected to the corresponding OS interface. Delegate thread is hidden by ReconOS and provides hardware threads with equal access to OS services as software threads, so that they can behave as software threads. For the user application, there is no need to know whether its threads are implemented with software or hardware. ReconOS was released on the Xilinx Virtex platform, which lacks partial reconfiguration. Therefore, this work focused on the pre-defined static FPGA accelerators, which, though undermines its value in context of the DPR management, still provides a classical solution for modeling hardware threads in OSs.

The multi-thread hardware accelerator model is also suitable for parallel high performance computing architectures, such as GPU, where concurrent multithread execution is achieved. The DPR feature permits a dynamic resource management strategy to optimize resources usages while meeting other requirements, e.g. power budget or working conditions. *ARTICo*³ proposed in [RVdlTR14] is an embedded architecture that is based on the NVIDIA CUDA execution model. It is able to calculate the optimal hardware resource allocation according to real-time conditions. The multiprocessors in traditional CUDA devices are substituted by hardware accelerators in dynamically reconfigurable slots. A dedicated unit *Resource Manager* is in charge of allocating application computations with a changeable amount of DPR slots (or threads). The resource of DPR slots used by one application can be modified on the fly. FIGURE 1.26 shows a possible resource allocation schedule that serves for two parallel applications. Note that, since the work is intended to migrate CUDA-like GPU execution model to *ARTICo*³, each DPR slot corresponds to a CUDA microprocessor thread. The major advantage of using DPR in

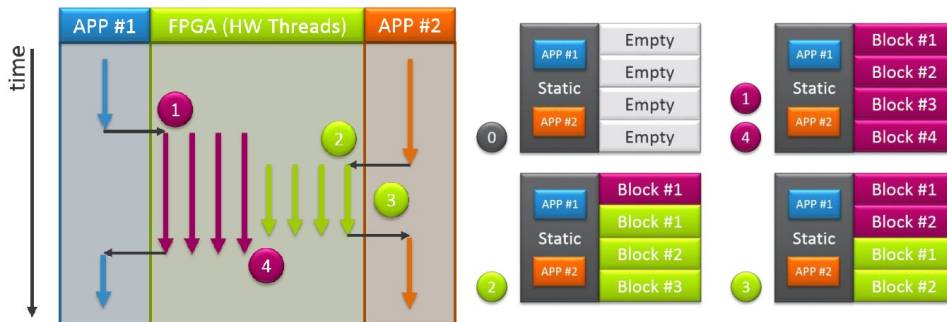


FIGURE 1.26 – *ARTICo³* handles the requests of two parallel applications APP#1 and APP#2. While APP#1 arrives first, the Resource Manager allocates all available DPR slots to accelerate the computation. Then a higher-priority application APP#2 arrives, which drives the *Resource Manager* to reconfigure slots of APP#1 to work on the new computation.

this case is that hardware slots are application-specific and can be allocated for different applications.

1.4.3 DPR Resource Virtualization

While CPU/FPGA architectures permit more complex software stack, general-purpose operating systems and even virtualization start to be employed, especially for the platforms requiring computation accelerations such as embedded systems and cloud computing environment. On these platforms, DPR resources are shared by multiple clients, whose execution is mostly independent. Thus, the allocation of DPR resources and coherency of hardware tasks are critical problems to be solved. In conventional systems, the accelerator must be exclusively used, that is, must be released by one of the clients and then claimed by another one. However, such manipulations to support different clients by continuously releasing and claiming the accelerators may lead to additional time overheads. Furthermore, such mechanism is not suitable for virtualization since virtual machines are isolated from each other, making the inter-VM DPR allocation even more expensive.

In this case, DPR resource virtualization turns out to be an ideal solution since it deceives the software clients by providing virtual accelerator accesses, and hence significantly simplifies the software development. To virtualize DPR resource, the classical hardware virtualization challenges have to be taken into consideration, such as the CPU/FPGA communication bandwidth, memory sharing, security and hardware computation data preservation, which have been under enormous research in recent years [WBP13][APL11][DRB⁺10]. Additionally, a dynamic reconfiguration model should also be included since the FPGA accelerators are no longer static anymore. In the following we introduce some research works which successfully employ DPR virtualization in their OS or virtual machine systems.

DPR resource virtualization was implemented in the Linux OS in [HH09] by pro-

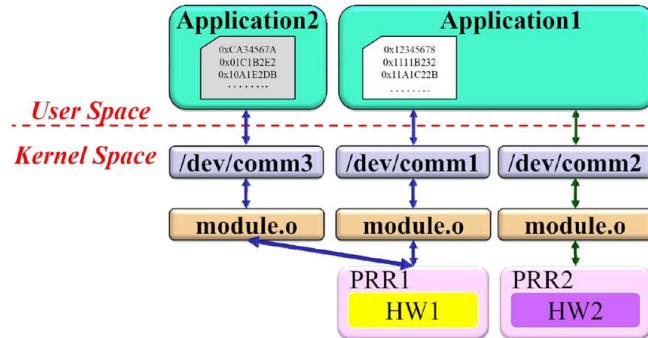


FIGURE 1.27 – Hardware resource virtualization in the OS4RS framework. Logic resources, i.e. device node modules, can be linked to hardware functions as many-to-one manner, so that one hardware function can be shared by multiple clients. On the other hand, one device node can be linked to alternate hardware modules so that it can always access available resources.

viding the framework called *operating system for reconfigurable systems* (OS4RS). The virtual hardware permits the same hardware devices and the same logic resources to be simultaneously shared between different software applications, that is, a reconfigurable hardware function can be virtualized to support more software applications. This mechanism is realized in the Linux kernel via leveraging the kernel modules, i.e. the device nodes. The virtualization mechanism is shown in FIGURE 1.27. Device nodes can be dynamically linked to different DPR accelerators on the application’s demand. While device nodes are linked to DPR modules in the many-to-one manner, the hardware resource is thus virtualized to software applications. Meanwhile, applications only need to deal with the device nodes, as OS4RS kernel links available DPR resources to them in a transparent manner. OS4RS is also in charge of reconfiguring DPR modules when necessary. Furthermore, the acceleration computation result is automatically stored in the module of device nodes. Since device nodes are continuous for applications, the computation can be easily picked up later by other hardware accelerators accessing the former results.

The above research focuses on the multitasking level on a single OS. In some researches, the technology of FPGA resources virtualization moves a step forward to the virtual machine systems, where guest OS or clients are sharing FPGA resources. Compared to multiple tasks on single OS, virtual machines are more isolated and independent, and the required hardware functions can be more diverse. Researches in this domain tend to consider FPGA accelerators as a static coprocessor that servers for multiple virtual machines. For example, in the research of pvFPGA [WBP13], one of the earliest researches in this domain, authors try to extend the Xen hypervisor to support FPGA accelerator sharing among virtual machines. However, this research focuses on the efficient CPU/FPGA data transfer method, with a relatively simple FPGA scheduler that provides a FCFS (*first-come, first served*) sharing on the accelerator, without including the partial reconfiguration technology.

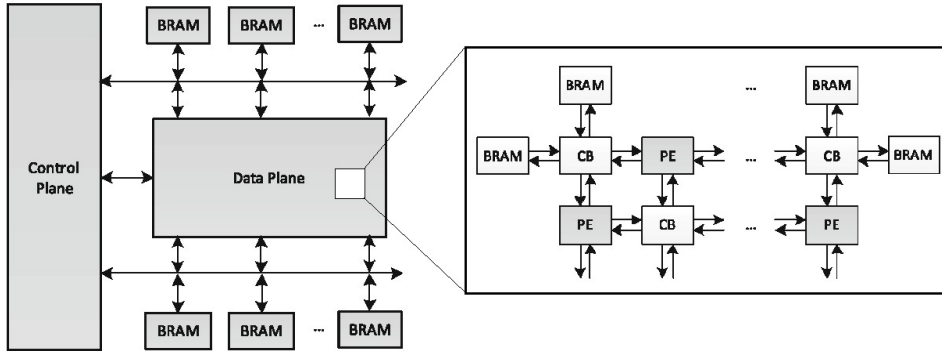


FIGURE 1.28 – An intermediate fabric region is composed of a data plane that hosts the programmable hardware computing components, and of the control plane that holds and defines the behavior and context frame of hardware tasks.

Meanwhile, DPR virtualization is more popular on cloud servers and data centers, which normally have a higher demand for computing performance and flexibility. Researches of this area are focusing on the integration of hardware accelerator resources in cloud computing system. By leveraging DPR technology, the concept of virtual FPGA (vFPGA) is provided as a virtual device for user custom hardware logic. For example, in [BSB⁺14] the authors use partial reconfiguration to split a single FPGA into several reconfigurable regions, each of which is managed as a single Virtualized FPGA Resource (VFR). In effect, this virtualizes the FPGA and makes it a multi-tenant device, although still requiring the external control of the Agent. A user can now allocate one of these VFRs and have their own custom designed hardware placed within it. Based on the similar idea, the work of RC3E [KS15] provides selectable vFPGA models, which permits users to require for DPR resources as full FPGA, virtual FPGA or background accelerators. The solutions mentioned above suit cloud server well since they virtualize FPGA as a programmable resource for clients, facilitating the user to implement their own IP cores. However, the target of our research is to implement an efficient DPR accelerator sharing among virtual machines. We tend to isolate users with the development and implementation of hardware accelerators. Thus, we will not discuss these researches in details in this thesis.

One research that is relatively close to our research is the framework of hardware task virtualization on a hybrid ARM-FPGA platform proposed in [JPC⁺14]. In this approach the authors modified the CODEZERO hypervisor to host guest OSs, including the real-time OS, $\mu\text{C}/\text{OS-II}$. The hypervisor is able to dynamically schedule hardware tasks according to two scheduling strategies : non preemptive and preemptive hardware context switching. Hardware resource virtualization is used to preserve and resume the hardware task state in switching, so that FPGA resources can be shared by hardware tasks. However, the classical DPR technology is not employed in this work for hardware reconfiguration. Instead, reconfigurable computing components in this framework are implemented by *intermediate fabric regions* (IF), which are built of built of an intercon-

nected network of coarse-grained *processing elements* (PE) overlaid on top of the original FPGA fabric. FIGURE 1.28 presents the programmable hardware computing components whose behaviors are defined by the configuration registers that controls the PE and interconnections. PEs are distributed across the fabric in a grid inside the *data plane* block. A PE is connected to all of its 8 immediate neighbors using programmable crossbar (CB) switches. The operation of the PEs and CBs is set by downloading configuration information to PE and CB configuration registers, which are referred to as *context frame registers* for they hold the execution context of hardware tasks. These context frames can be saved and resumed by the *control plane* block, in which way hardware tasks can be scheduled in and out safely.

Since the workload of managing context frame registers is significantly less than traditional DPR netlist information, this approach presents low reconfiguration and management overheads, e.g. hardware tasks can be switched within several microseconds. However, the application of intermediate fabric is limited to relatively simple computing functions, since its possible configuration space and circuit scale are much smaller. Besides, mapping circuits to IFs is less efficient than using DPR or static implementations as the IF imposes a significant area and performance overhead. As concluded by the works, this approach is more appropriate to systems with light but frequently-switched computations, while DPR is more suitable for heavy workload computations.

1.5 Summary

In this chapter we have introduced some of the typical existing technologies for embedded virtualization, real-time virtualization and DPR management on ARM-FPGA FPGA platforms.

In our work, the target platform is the Xilinx Zynq-7000 SoC, which includes a dual-core Cortex-A9 processor and 7-series FPGA fabric with the dynamic partial reconfiguration (DPR) technology [Xil14c]. This is currently one of the mostly used ARM-FPGA system. In this platform, what we require is a lightweight virtualization solution that is appropriate for simple IoT devices. In this case, the existing virtualization technologies are not ideal for our purpose, since their complexity is mostly higher than we expect and most of them lack the support for our target platform. Furthermore, several technologies e.g. OKL4, remain close-source and are impossible for further studies. We also find the widely-used compositional real-time scheduling algorithm unsuitable for our intention as it requires the computation of PRM interfaces or extra scheduling servers.

Therefore, in our research we intend to propose a micro-kernel approach that supports real-time virtualization with minimal software complexity. In this paper, we focus on the design and evaluation of the real-time virtualization micro-kernel, KER-ONE, whose properties are a small-sized Trust Computing Base (TCB) and an accurate scheduling for real-time tasks. We also orient our work to guarantee minimal modifications of guest operating systems running on top of our kernel.

Meanwhile, we extend our framework to focus on the application of classical DPR technology being supported by virtual machine systems in CPU/FPGA architectures.

Efforts have been made to develop a dedicated virtualization architecture that provides efficient DPR resource sharing among virtual machines, while meeting the potential challenges. In the proposed framework, we attempt to improve the hardware task security, to minimize the extra performance loss caused by hardware reconfiguration, and to provide a unified user-transparent interface for DPR resource accesses.

CHAPTER 2

KER-ONE : LIGHTWEIGHT REAL-TIME VIRTUALIZATION ARCHITECTURE

In this chapter, we introduce our research on real-time embedded system virtualization. The proposed virtualization framework, Ker-ONE, is a microkernel-based virtual machine system that supports real-time virtual machine scheduling. The subject of this research is to provide light-weight, flexible and real-time virtualization solutions for small-scaled ARM-FPGA hybrid embedded system. By eliminating unnecessary components and strictly following the principle of least privilege, the Ker-ONE microkernel achieved high virtualization performance with low complexity. In this chapter, we will focus on the virtualization mechanism of Ker-ONE microkernel, including the virtual machine mechanism and the real-time scheduling strategy that supports guest RTOS. Since Ker-ONE is also extended to support FPGA fabric, it should also be noted that the mechanism concerning the management of partially reconfigurable FPGA accelerators will be presented in the next chapter.

2.1 Overview of the Ker-ONE Microkernel

In the work of Aichouch [Aic14], one of the precedent researches of this thesis, a simplified micro-kernel, Minimal NOVA, was revised from the NOVA hypervisor [SK10] in order to study real-time scheduling configurations on the x86 platforms. The author only preserved the functionality of threads in NOVA to implement a user-level library that allocates real-time scheduling as a middleware on top of a microkernel-based OS. The user-level library of Mollison and Anderson [BA07] was used in this research to develop and evaluate scheduling and locking techniques for the proposed real-time system.

Nova is an interesting architecture which is very suitable to embedded systems. Its low kernel complexity facilitates the integration of new components, making it easier to implement custom virtualization mechanisms. However, whereas both NOVA and Mini-

mal NOVA are intended for the x86 architecture, the object of our research is ARM-based embedded systems. Considering the significant architecture difference between x86 and ARM, and the different applications between desktop computers and embedded systems, we had to re-design from bottom-up a novel micro-kernel, for the ARMv7 architecture. This new micro-kernel has been entitled Ker-ONE. Like NOVA, Ker-ONE features a small TCB size and low complexity. However, most designs of the framework are no longer connected to NOVA hypervisor.

The design of Ker-ONE follows some basic principles. First, we would like to avoid complex software implementation and by considering very simple guest OS systems such as μ C/OS or FreeRTOS. Second, we require a simple but reliable real-time scheduler. Third, the hypervisor needs to be scalable and easily-adaptable to extension mechanisms. One of these mechanisms consists in dynamically managing hardware tasks among multiple virtual machines with the help of partial reconfiguration.

Virtualization on ARM-FPGA platform has some limitations. Though full-virtualization of ARM processors is now possible by using ARM virtualization extensions such as in Cortex-A15, this technology is currently unavailable on ARM-FPGA platforms. Therefore, Ker-ONE is developed using para-virtualization technology. The design of Ker-ONE following the principle of low complexity, we decided to focus only on critical virtualization functionality and eliminate non-mandatory features. Ker-ONE provides a small TCB size, and a RTOS-support mechanism which is proposed to handle the real-time constraints of most applications. Currently our research is based on the following assumptions :

- In a first step, We have only considered single-core architectures, leaving multi-core systems to future prospects, which is also the case in other state-of-the-art researches such as RT-Xen and L4.
- We focus on the virtualization of simple OSs, instead of heavy-weight OSs such as Linux, since para-virtualizing these complex OSs would be quite expensive and goes against the purpose of our work.
- In order to realize real-time virtualization with less-complex scheduling, we assume that all critical real-time tasks are held in one specific guest RTOS, while tasks with lower priorities are held in general-purpose OSs (GPOSs). Thus, Ker-ONE is designed to co-host one guest RTOS and several GPOSs.

Ker-ONE is based on the ARMv7 architecture. In order to present the virtualization mechanisms of Ker-ONE, it is essential to understand the basic features and terms of ARMv7.

As described in section 1.1.2.1, an ARMv7 processor runs with two privilege levels : non-privilege level (PL0) and privilege level (PL1). Critical system resources, e.g. configuration registers, interrupts, are excluded from PL0 and are only accessible in PL1, which guarantees the security of privileged codes. In details, ARMv7 offers 6 main execution modes for different execution scenarios : PL0 is exclusively occupied by the user mode (USR), and at PL1 exist five modes i.e. supervisor (SVC), interrupt (IRQ), fast

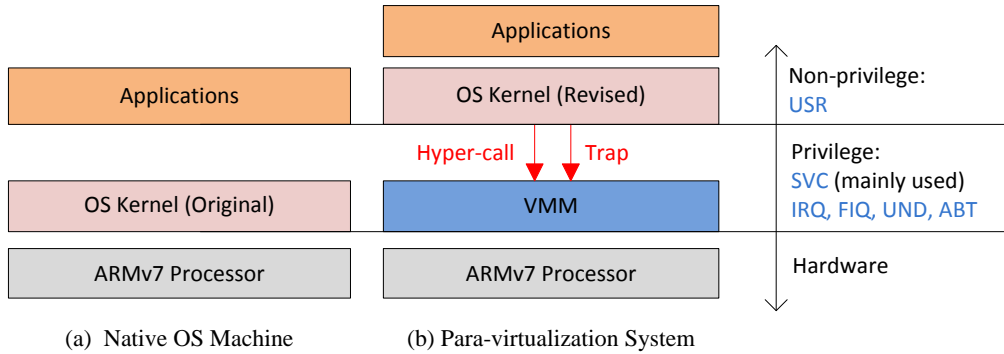


FIGURE 2.1 – Para-virtualization systems on ARMv7 architecture.

interrupt (FIQ), undefined instruction (UND) and data-abort (ABT) modes. In most operating systems, the OS kernel executes in supervisor mode, and manages all hardware resources, while user applications/threads/processes run in the user mode, as shown in FIGURE 2.1(a). Other execution modes, on the other hand, are reserved to receive and handle corresponding system exceptions. Whenever an exception occurs, the processor automatically switches to the corresponding exception mode for handling. TABLE 2.1 lists the execution modes and descriptions in ARMv7.

TABLE 2.1 – Description of ARMv7 execution modes.

Mode	PL	Description
USR	0	Execute user applications and processes
SVC	1	Major mode for privileged software, e.g. OS kernel or VMM
IRQ	1	Handle Interrupt Exceptions
FIQ	1	Handle Fast Interrupt Exceptions
UND	1	Handle Undefined Instruction Exceptions
ABT	1	Handle Data-Abort Exceptions

The two-level structure provides the OS kernel with a safe environment to run by isolating the user space and the kernel space. However, in para-virtualization, guest OSs should be ported to a non-privileged level whereas the privileged level is occupied by VMMs. According to the mechanism of para-virtualization, the source code of a guest OS must be modified to properly run in USR mode and sensitive instructions have to be replaced with hyper-calls, as shown in FIGURE 2.1(b).

Ker-ONE consists of both the host micro-kernel and a user-level environment. In the Ker-ONE virtualization framework (see FIGURE 2.2), the microkernel is the only component that runs at the highest privilege level, mainly in supervisor mode. According to the smallest TCB policy, only the basic features that are security-critical remain in the microkernel, including the memory management, the inter-VM communication, the exceptions handling, and the scheduler. The VMM runs on top of the basic microkernel

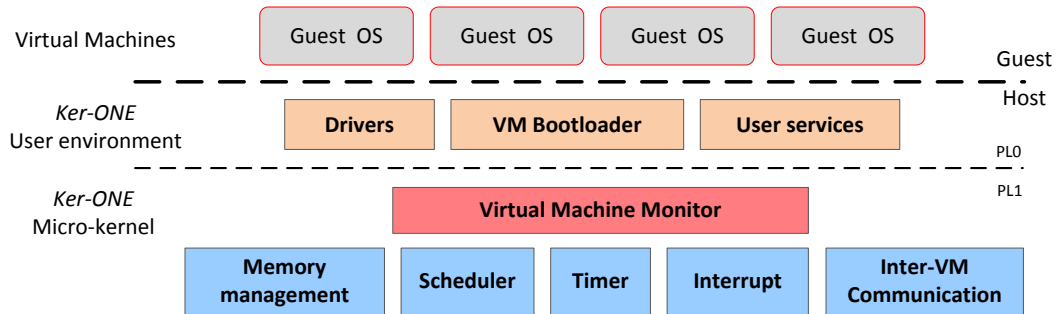


FIGURE 2.2 – Ker-ONE consists of the micro-kernel and virtual machine monitors in privileged level and User environment in non-privileged level.

functions to support the execution of guest operating systems in the associated virtual machine. It provides a virtual hardware layer, emulates sensitive instructions and handles virtual machine’s hyper-calls. The user environment runs in the user mode (PL0) and is composed of additional system applications, for example, device drivers, file systems, VM bootloaders and special-purpose services (as the Hardware task manager which controls the hardware accelerators in the partially reconfigurable FPGA fabric). A virtual machine can execute a para-virtualized operating system or a software image of user application. Each virtual machine has an independent address space, and executes on the virtual hardware provided by the VMM.

While guest OS is de-privileged to the un-privileged level, VMM is in charge to virtualize the resources of CPU computing resources and provide virtual machines with complete and virtual execution environments. In the following we present the detailed implementation of VMM mechanisms in Ker-ONE.

2.2 Resource Virtualization

In virtual machine systems, guest OSs run on top of abstract physical resources. To guarantee the correctness of guest OSs execution, the VMM has to establish an execution environment that abstracts and virtualizes computing resources, including the CPU computing resources, the instruction architecture set, the memory spaces, etc. In this section we introduce the virtualization mechanism of these resources.

2.2.1 CPU virtualization

As described above, ARMv7 processor runs in six execution modes. The computing of CPU is based on system registers, which are divided into different categories according to their functionality. While some general-purpose registers are shared by all modes, some specific registers are respectively banked for different modes to facilitate the mode independency and context preservation.

R0 – R7	R0 – R7 (Shared)				
R8 – R12	R8 – R12 (Shared)				R8 – R12 (Banked)
R13 (SP)	SP_svc	SP_abt	SP_und	SP_irq	SP_fiq
R14 (LR)	LR_svc	LR_abt	LR_und	LR_irq	LR_fiq
R15 (PC)	PC (Shared)				
User	Supervisor	Data-abort	Undefined Instruction	Interrupt	Fast interrupt

FIGURE 2.3 – The allocation of general-purpose registers (R0-R14) and PC (R15) register for different ARMv7 execution modes.

FIGURE 2.3 depicts the usage of the R0-R15 register group in different execution modes. R0-R14 are 32-bit general-purpose registers that are accessible in all modes. R0-R12 are frequently used for CPU computing and are shared by most modes, except the FIQ mode, which possesses an independent R8-R12 bank for a faster interrupt handling process. R13 (or SP) is always used to store the software stack pointer. R14 (or LR) is used as a link register to store the return address when a subroutine call is made. SP and LR registers are used by most C/C++ compilers by default. R15 (or PC) is defined as a program counter which always points to the address of next executable instruction. Note that SP and LR are respectively banked for each mode. Whenever CPU switch from one mode to another, the corresponding register bank of successor mode will be automatically used by instructions. With this mechanism software of different mode can easily maintain their execution context, e.g. stack position and program pointer.

Additionally, the CPU execution also relies on the processor state register (PSR). PSR is an assembly of processor flags that control the state and condition of CPU, such as the execution mode, the interrupt mask, etc. In most cases, the configuration of PSR may influence the correction of execution. Therefore, PSR is designed to be partially accessible for non-privilege level, and is fully-accessible for privilege level. In other words, some critical flags in PSR are reserved read-only (RO) to user mode. In TABLE 2.2 we describe the attributes of some PSR states that will be frequently-used in this thesis.

Note that only the APSR flags are accessible in all modes, since these flags are in charge of storing computation results and help for the conditional execution in user mode. On the other hand, the critical flags in PSR are accessible only in privileged software execution. User mode software is allowed to read values from these flags, while any writing operation on them is automatically ignored by the system.

As PSR plays such an important role, ARMv7 proposes banked PSR registers to help maintaining the coherency of PSR during execution. FIGURE 2.4 presents the management of PSR in ARMv7 processors. The *Current Program Status Register* (CPSR) is shared by all modes to hold the current execution status. The *Saved Program Status Register* (SPSR) is used to store the current value of the CPSR when an exception is taken so that it can be restored after handling the exception. Each exception handling mode can access its own SPSR. User mode does not have an SPSR because it is not

TABLE 2.2 – Description of ARMv7 execution modes.

Domain	Access	Description
<i>PSR.APSR</i>	PL0 : RW PL1 : RW	The Application Program Status Register (APSR) holds copies of the <i>Arithmetic Logic Unit</i> (ALU) status flags. They are also known as the condition code flags. They are used to determine whether conditional instructions are executed or not.
<i>PSR.I</i>	PL0 : RO PL1 : RW	<i>Interrupt Mask</i> indicates if interrupts are masked. It is used to mask/unmask all interrupts.
<i>PSR.F</i>	PL0 : RO PL1 : RW	<i>Fast Interrupt Mask</i> indicates if fast interrupts are masked.
<i>PSR.M</i>	PL0 : RO PL1 : RW	<i>Mode Filed</i> determines the current mode of the processor. Changing the value of this filed will switch CPU to the corresponding mode immediately.

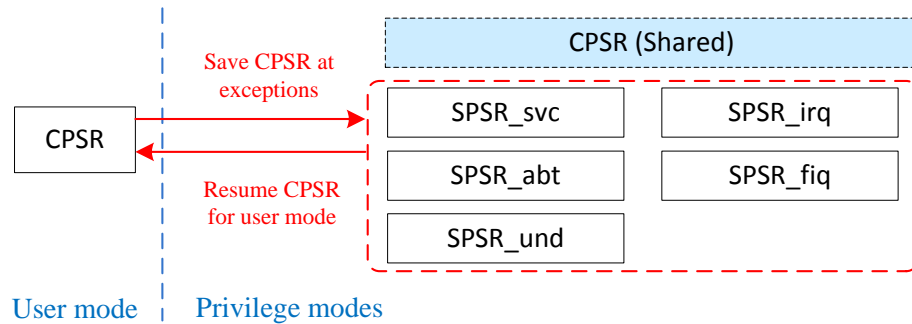


FIGURE 2.4 – The management of Program Status Registers in ARMv7 architecture.

exceptions handling.

ARMv7 also provides coprocessors to extend the functionality of CPU. In classical implementation of ARMv7 architectures, two coprocessors are available for software : CP14 and CP15. These coprocessors are designed for specific purposes. For example, Coprocessor 15 (CP15) provides system control functionality, which includes architecture and feature identification, as well as control, status information and configuration support. The status of these coprocessor registers should also be properly virtualized to successfully host virtual machines.

2.2.1.1 CPU Resource Model

In this part we propose the ARMv7 CPU resource model on which our virtual machine are executing. In order to achieve a simpler modeling, we focus on the resources that are logically and physically inseparable to processor itself, decoupling less-related components for later discussion. From this viewpoint, the ARMv7 processor can be expressed as in FIGURE 2.5, where only the states of processor and co-processor are included in the

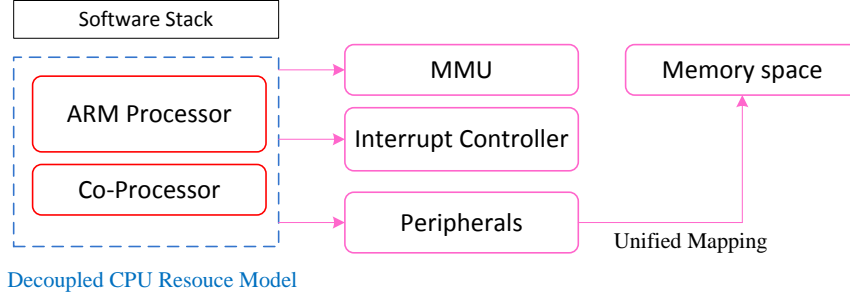


FIGURE 2.5 – ARMv7 processor modeling by decoupling integrated functionalities.

CPU model. Meanwhile, the management of memory space and interrupts are performed by independent components MMU and interrupt controller, and are out of the CPU’s range. Moreover, since all I/O interfaces are memory-mapped, the states of I/O devices are managed via custom implementations and are not part of the CPU functionality.

Here, we review Penneman’s resource model [PKR⁺13] that is required to fully support a bare ARMv7 processor, which is expressed as the concept of *Machine State* in Equation (1.2) :

$$S \equiv \langle E, M, P, G, C, A, D^M, D^P \rangle \quad (2.1)$$

In our system, after eliminating memory space, peripherals and interrupts from the CPU execution resource (as shown in FIGURE 2.5), the model of CPU resources can be simplified from the complete *Machine State* S . to be a subset of S :

$$S_C \equiv \langle M, P, G, C \rangle . \quad (2.2)$$

G refers to the system general-purpose registers (R0-R14). P is the program counter (R15). C consists of an ARM Processor State Register (CPSR/SPSR) and a CP14/CP15 coprocessor, and $PSR.M$ bit holds the status of processor mode. Note that the registers of the coprocessor are not necessarily completely included in the C set, since some registers are read-only or are ignored in the CPU execution. Thus, coprocessors have to be carefully analyzed to determine which of their registers belong to C .

In terms of virtualization, the execution of virtual machines should be provided with a virtualized CPU resource abstraction that meets the S_C model. These virtualized resources are supposed to be available for each virtual machine independently. As a solution, we propose a virtual CPU (vCPU), which emulates all the processor resources that are needed for virtual machines. FIGURE 2.6 demonstrates the mechanism of CPU resource virtualization in virtual CPU. In our mechanism, CPU resources are considered as the context of virtual CPU execution, denoted as *Execution Context* (EC), which holds and virtualizes values and configurations of necessary resources in the S_C model.

The Execution Context is virtualized according to different policies. The state of the $\langle M, C \rangle$ set, as the Processor State Register (CPSR/SPSR) and co-processor CP14/15, is critical for system security and should be supervised by the VMM. Moreover, some of these resources are physically inaccessible in user mode, making it mandatory to manually

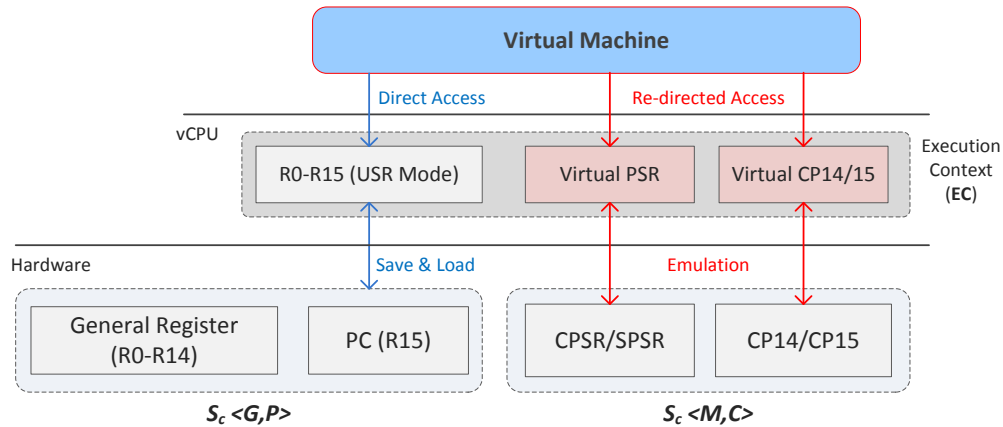


FIGURE 2.6 – The mechanism of CPU resource virtualization. Resource $\langle G, P \rangle$ is managed in a simple save/restore method, and resource $\langle M, C \rangle$ have to be emulated as virtual resources for virtual machines.

emulate them for virtual machines. As a solution, an emulation interface is used to handle virtual machine’s accesses to these resources by re-directing them to the corresponding virtual states rather than in the physical ones. For example, configuring the $PSR.I$ field in CPSR, which is not permitted in user mode, is a frequent operation in most guest OS kernels. In our system, this operation is forwarded to the *virtual CPSR* in EC and has no influence on the physical CPSR register. Another typical example is when guest OS switches between kernel space and user space. In this case, the *virtual Mode* of vCPU should be switched accordingly. Via this mechanism, virtual machine can fully access virtual resources while the physical resources are always controlled by the VMM.

On the other hand, the $\langle G, P \rangle$ set, as general-purpose registers (R0-R14), and the program counter (R15), is commonly used by software parts. Accesses to these resources make no threat to the system and can be totally trusted. The virtual CPU maintains the consistency of these registers for each virtual machine via the simple Save & Load mode. Thus, a virtual machine uses these resources as on a native machine. However, it should be noted that LR and SR have virtual banked values in virtual modes. Whenever the *virtual Mode* of vCPU is changed, the corresponding LR and SR virtual values should be loaded to the physical registers immediately.

2.2.1.2 Instruction Emulation

The ARMv7 ISA is initially non-virtualizable, meaning some sensitive instructions may run in non-privileged user mode without triggering any exceptions and cause unexpected results. These sensitive instructions involve accesses to critical system resources (i.e. set C in S_C), and are originally supposed to execute in privilege software such as OS kernels. When running in user mode, these instructions will attempt to read/write values at privilege registers, and will end up with incorrect results. The summary of

non-virtualizable sensitive instructions in ARMv7 is depicted in TABLE 2.3.

TABLE 2.3 – Sensitive non-privilege 32-bit ARM instructions.

Instruction	Functionality
LDC, STC, MCR, MRC	Coprocessor Access
LDM(exception return)	Exception handling and context switch
LDM/STM (user mode registers)	
RFE	
SRS	
CPS, MRS, MSR	System register (PSR) Access
WFE, WFI	Low-power Mode

In TABLE 2.3, sensitive instructions are categorized according to their functionality and operation objects, as explained in the following :

- **Coprocessor access** : these instructions are used to interact with coprocessors, which will be denied when executing from user mode. For example, a user mode MCR instruction that attempts to write data to a coprocessor register will be automatically ignored without triggering any exception.
- **Exception handling** : these instructions are intended for handling exceptions, especially for fast returning from exception mode to user mode. Instructions of this type involve loading values to the user mode registers, such as LDM that is always used for resuming CPSR from SPSR at the end of exception handling.
- **System register access** : some instructions directly read or write to system registers such as the CPSR or SPSR. In OS kernels, these instructions are frequently used for CPU control, e.g. changing mode, or manipulating interrupts. In practice, they are actually the sensitive instructions that are the frequently used by virtual machines.
- **Low-power mode** : two special instructions, *Wait-For-Event* (WFE) and *Wait-For-Interrupt* (WFI), are also defined as sensitive instructions and are designed to put the CPU into low-power state until an interrupt/exception occurs. They are used when the CPU is idle in order to save energy. In our system, these instructions should be detected by the VMM when a virtual machine is in IDLE state, so that the VMM can schedule another virtual machine to run. Details of this scheduling strategy will be explained in Section 2.5.

The virtualization of sensitive instructions includes the detection and emulation of instructions. In the source code of guest OSs or applications, these sensitive instructions are manually replaced by corresponding hyper-calls or macros as extensional virtualization patches, so that they can be detected and re-directed to the virtual resources instead of the physical ones. VMM properly emulates the behavior of these instructions by manipulating the virtual resources in the virtual CPU.

2.2.1.3 Virtual CPU Model

The virtual CPU (vCPU) is the fundamental component of the VMM that directly hosts virtual machines. The aim of the vCPU is to provide virtual machines with an abstracted layer that mimics the behavior and resources of the physical processor.

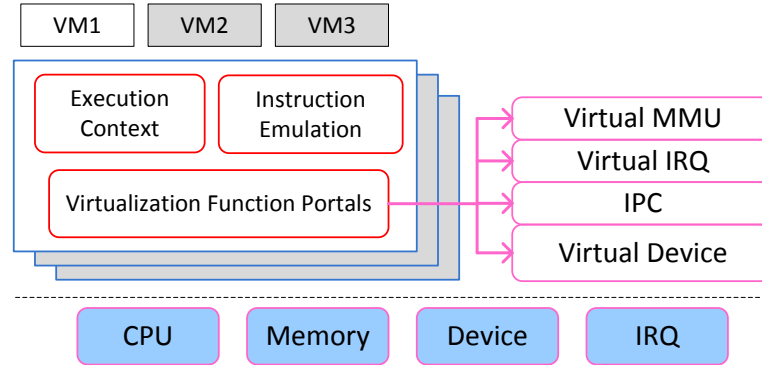


FIGURE 2.7 – a VCPU has three major properties : (1) holds the virtual machine execution context (EC); (2) emulates sensitive instructions; (3) provides access to other system resources/functionality. Virtual machines run on independent vCPUs that play the role of an intermediate layer between virtual machines and the rest of the VMM.

We built up the vCPU model based on the simplified decoupled processor model (see FIGURE 2.5) introduced in Section 2.2.1.1. In this model, the CPU is responsible not only for instruction execution, but also for connecting software with other system services, e.g. memory management, scheduling, interrupt handling and inter-process communication. Therefore, whereas the vCPU is in charge to hold CPU resources and handle instruction emulation, it should also provide function portals that links to the decoupled resource or services, as shown in FIGURE 2.7. Based on this model, the VMM may easily establish virtualized execution environments by initializing independent vCPUs to each virtual machine. In this case, a vCPU acts as an intermediate layer between virtual machine software and physical resources. The VMM performs the switching of different virtual machines via activating the corresponding vCPU execution context.

The behavior of virtual machines are monitored through vCPUs by means of hyper-calls or exceptions (or traps). A vCPU is in charge of gathering all exceptions (including hyper-calls, as special exceptions) that may occur in a virtual machine, and is responsible for distinguishing and dispatching them. Based on the nature of exceptions, it provides them to different functional components for proper handling. A VCPU also includes a local handler for hyper-calls and traps, and several capability portals for other VMM functions. Normally, trapped instructions and hyper-calls that access CPU resources (i.e. resources held by EC) should be handled locally. On the other hand, other exceptions should also be re-directed. For example, when a data-abort exception is trapped by a virtual machine, the vCPU will analyze it and forward this exception to the memory management for further processing.

2.2.2 Vector Floating-Point Coprocessor Virtualization

The Vector Floating-Point (VFP) co-processor in ARMv7 architecture is an important component to speed up certain types of applications, especially in the communication domain where intense floating-point computing is involved. However, due to the huge size of the register bank that VFP is using (32 64-bit registers), the save/restore of the VFP context heavily degrades the overall performance. Hence, an efficient virtualization mechanism of VFP is required. Given the heavy cost of a context switch, we apply the *lazy switching* mechanism for VFP virtualization, which means that the VFP context is only switched when necessary and not at every switch. *Lazy switching* has also been proposed for OKL4 kernels in [VH11], but authors only theoretically estimated the overhead. In [YYY13], an implementation of VFP *lazy switching* was proposed as an external patch to the Xen-ARM hypervisor. In this section, we propose another approach of *lazy switching* to be used in Ker-ONE.

There are two key features when dealing with lazy switching : trapping and consistency. In the ARMv7 architecture, VFP instructions are not privileged operations and will not be trapped if executed in user space. To trap VFP operations, the VFP configuration register *FPSCR.EN* bit has to be cleared in order to disable the VFP coprocessor so that any VFP instruction generates an *undefined instruction* (UND) exception. Also, VFP relies on its register bank for computation, which is denoted as the *VFP context* (FPC). With lazy switching, the current context in the physical VFP bank may not correspond to the currently running virtual machines. In this case the VFP context must be correctly saved to ensure its consistency with virtual machines.

FIGURE 2.8 shows the mechanism of *VFP context* (FPC) virtualization. For each virtual machine, a vCPU is created initially without FPC. When a virtual machine attempts to use the floating-point engine, this request is passed to the VMM, which will then create a corresponding FPC and associate it to the vCPU (as in FIGURE 2.8(a)). When multiple virtual machines are using VFP, the physical VFP register bank is always loaded with the FPC of virtual machine that is currently using it. The process of changing the VFP register content from one virtual machine to another is called VFP context switch. The current active FPC in the physical VFP register bank is indicated by a global pointer called *FPC_Current*.

It should be noted that a virtual machine is exclusively linked to a FPC. If a guest OS executes multiple processes which are sharing VFP, there should be local floating-point contexts (denoted as *Local FPCs*) which will be locally handled by the guest OS, as shown in the VM3 in FIGURE 2.8(a).

FIGURE 2.8(b) presents an example of the process flow of VFP lazy switching. In this example, VM 1 and VM 3 are executing processes that performs floating-point computations. To detect the floating-point instructions in a virtual machine, the *FPSCR.EN* state register of VFP is cleared at every virtual machine switch, so that the VFP instructions are disabled whenever a virtual machine is scheduled. When the virtual machine runs, any VFP operation would be trapped to the VMM as UND exception, indicating that the current virtual machine attempts to use the VFP. Then, the VMM will enable VFP

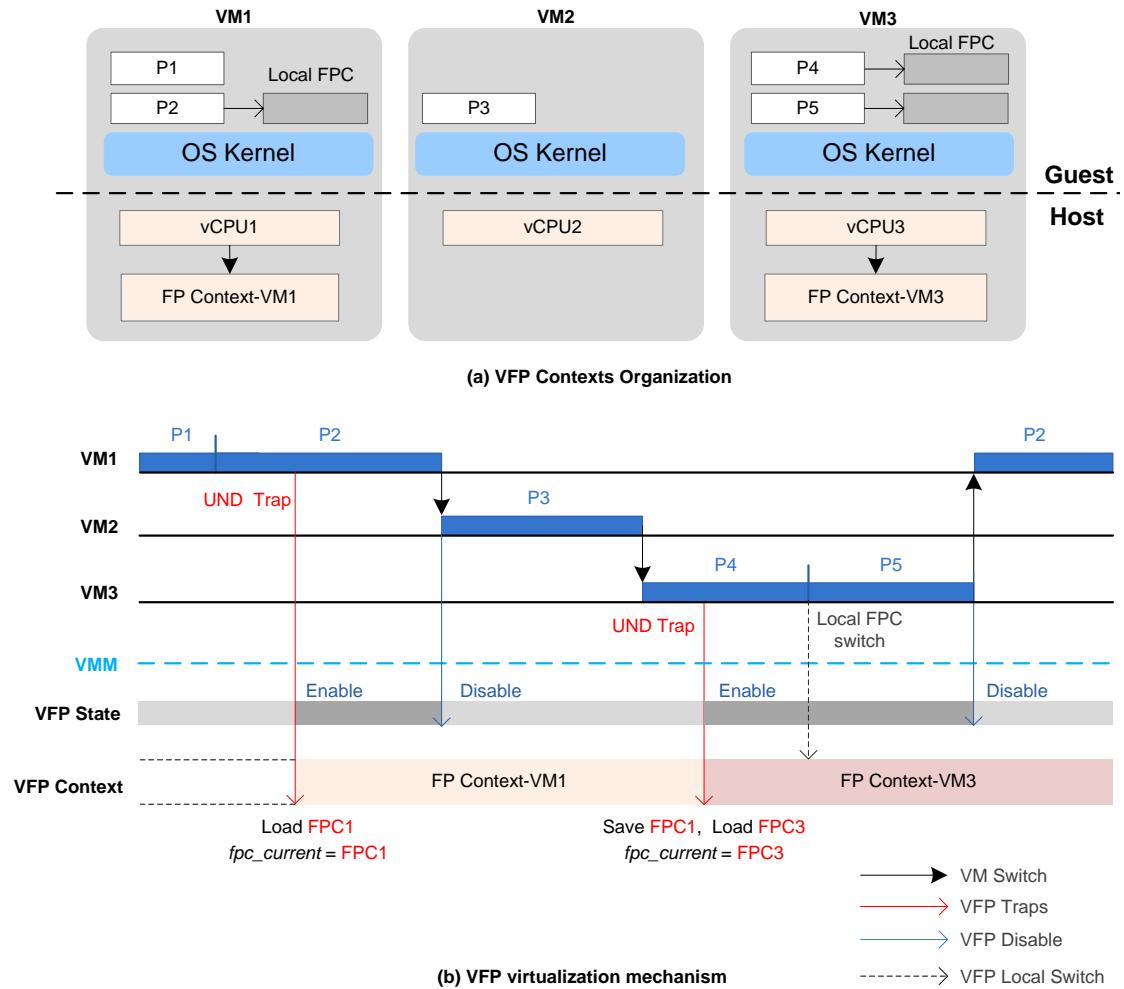


FIGURE 2.8 – VFP virtualization mechanism via lazy virtualization. (a) *VFP Contexts* (FPC) are created for VM1 and VM3 respectively since they have processes that use VFP resources. Guest OS are responsible of the preservation of Local FPCs. (b) VFP is disabled at each VM switch, and its register contents are switched only when necessary.

by setting the *FPSCR.EN* bit, and performs VFP context switch to load the register bank with corresponding FPC of current virtual machine. During the context switch, the current VFP register values would be saved to the FPC of the previous virtual machine client (i.e. VM 1 in FIGURE 2.8(b)), and resumes the VFP context of the current running vCPU (i.e. VM 3) and updates the *FPC_Current* pointer. In this mechanism, *VFP Context Switch* is performed only when the usage of VFP has to be preempted by another virtual machine. From FIGURE 2.8(b) we can notice that the occurrence frequency of the VFP context switch is significantly reduced.

2.2.3 Memory Management

The memory space in ARMv7 architecture is mainly managed by the Memory Management Unit (MMU), which performs virtual address translation, page table configuration and page access control. By creating page tables, the MMU is able to map physical memory to virtual address space according to two sizes of pages : 1M or 4K size page. A two-level cache system is provided to accelerate memory access, including instruction/-data independent L1 Cache (ICache and DCache) and unified L2 Caches. Besides, a Translation Look-aside Buffer (TLB) is used to record the performed address translation so that these translation results can be directly reused in future accesses.

Ker-ONE leverages a simplified mechanism of *shadow-mapping* for memory management. Note that, the technology of *shadow-mapping* is used by some other virtualization technologies, such as KVM [DJ11][DLC⁺12]. They seek to support multiple user-level independent address spaces (i.e. user-level processes with independent page tables) to host complex desktop OSs such as Linux or Android. However, as mentioned before, Ker-ONE is intended to work with simple OSs. In this case, the support of multiple user-level protection domains are not mandatory since the guest OSs that we have considered are mainly OSs with single-domain page tables, e.g. $\mu\text{C}/\text{OS-II}$, FreeRTOS, etc. Moreover, the implementation of such a technology is likely to greatly increase the complexity of the micro-kernel. In this context, we focus on the memory virtualization of single guest protection-domains.

2.2.3.1 Memory Access Control

The traditional memory access control is realized by tagging memory pages with different access permission (AP) flags, which can be categorized as three types : *privilege access*, *full access* and *real-only access*. Pages with privilege access can only be accessed from privilege modes, whereas full access tagged pages are open to all execution modes. Whenever software parts attempt to access the memory space, the MMU checks the AP flags of these pages and generates a page fault exception (i.e. *data-abort* (ABT) exception) if the access is not permitted. Thus, the memory space with a privilege access flag is protected from less privilege software. This mechanism is widely used by OS kernels to establish isolation between the user space and the kernel space.

In virtualization, the address space in virtual machine systems is organized in a two-stage hierarchical structure. The first stage maps the guest virtual space to the guest physical space, and the second stage maps guest physical space to the host physical space. In other words, memory space in virtualization requires the access control on different privilege levels : *host kernel* (HK), *guest kernel* (GK) and *guest user* (GU), respectively for VMM, guest OS kernels and guest OS applications, as shown in FIGURE 2.9. To guarantee the system security, software must be forbidden to access resources at a higher privilege layer. To create protected memory space for host kernel space (i.e. VMM), we take advantage of the default AP mechanism of the MMU, by configuring host kernel resources with the *privilege access* flag, and the virtual machine space (including guest kernels and guest users) with the *full access* flag.

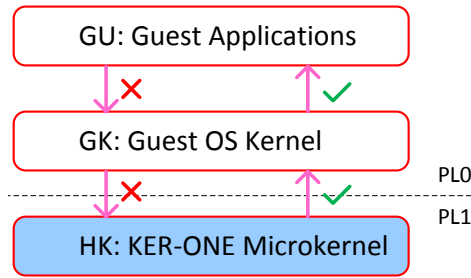


FIGURE 2.9 – The access control among different address space privileges in virtual machine systems : host, guest kernel and guest user.

Additionally, we proposed an additional mechanism to separate the guest kernel and guest user space, to protect guest OS kernels. In Ker-ONE, we exploit the domain access control register (DACR) of the MMU, which offers 16 access control domains (D0-D15). Each memory page can be linked to one of these domains. Pages that belong to the same access control domain are controlled by the corresponding bits in the DACR. Each domain has three possible states : *no access (NA)*, *client* and *manager*. The NA domain rejects any access and generates a Domain fault. The *client* domain accesses are always checked against the access permission. For the *manager* domain accesses, the access permission are not checked and are always permitted.

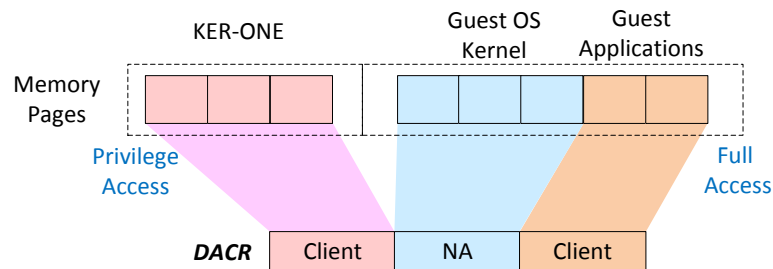


FIGURE 2.10 – By associating memory pages to different DACR domains, the memory space of virtual machines is divided into several domains with different access policies.

As shown in FIGURE 2.10, guest kernels and guest users are associated to different DACR domains, denoted as GK domain and GU domain respectively. When the CPU is running in the GU space, the GK domain is set as NA so that it is forbidden to the guest user. On the other hand, when a guest OS enters the kernel space, the VMM switches the DACR GK domain state to *client* so that the entire VM space is accessible for OS kernel software, as in the native machine. Via this mechanism, guest kernels are isolated and protected from guest user software. The overall mechanism of memory access control is summarized and listed in TABLE 2.4.

TABLE 2.4 – The configuration of *Access Permission* Flag and DACR Domain State for three different privilege levels : *guest user*, *guest kernel* and *host kernel*.

Memory Space Domain	AP Flag	DACR Domain State		
		GU Level	GK Level	HK Level
Guest user	Full Access	client	NA	client
Guest kernel		client	client	client
Microkernel	Privileged	client		

2.2.3.2 Address Space Virtualization

The VMM provides isolated virtualized address spaces by manipulating the translation page tables for virtual machines. As introduced previously, the MMU in ARMv7 performs a single-stage virtual-to-physical address mapping, making it unsuitable for the two-stage hierarchical mapping structure in virtualization. As a result, the VMM is responsible for interpreting the virtual address space in a virtual machine and for transforming this mapping into an actual ARMv7 one. The principle of *shadow-mapping* is used in our approach.

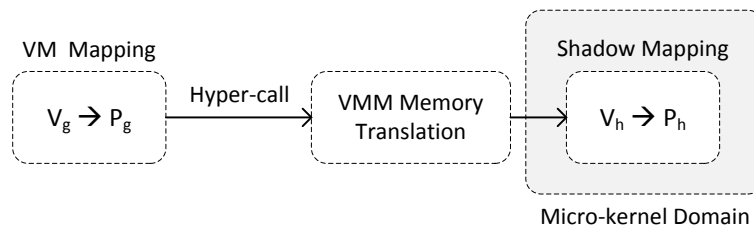


FIGURE 2.11 – The mechanism of address space virtualization by using shadow mapping mechanism and hyper-calls.

FIGURE 2.11 depicts the mechanism of address space virtualization for virtual machines. For each independent virtual machine address space, a translation page table is created by the VMM. This page table is referred to as the shadow page table. Shadow page tables are held in host kernel space and cannot be directly accessed by guest OS. Instead, guest OSs have to use hyper-calls to create and manage mappings between the guest virtual and physical address spaces. For example, to require a mapping from a guest virtual address v_g to a guest physical address p_g , a guest OS uses the hyper-call interface :

```
hypercall_create_page(guest_virt, guest_phys, page_attributes).
```

During the hyper-call, the VMM translates the guest physical address into the host physical address p_h and creates the page table entry that directly maps v_g to p_h . Meanwhile, the argument of *page_attributes* passes the attributes of the created page, e.g. the page size and the privilege level (OS kernel or guest user). For guest OS kernels, any operation on page tables must be performed via hyper-calls, including the creation,

deletion and remapping of page tables.

By default, each virtual machine is allocated with a determined size of physical memory space, whose usage is under the supervision of VMM. Any mappings or accesses outside this area will be rejected by the VMM. In this way, each virtual machine is strictly isolated from each other.

We have introduced the criterion of memory virtualization for shadow mappings in Section 1.1.1.2, Eq.(1.6). Assume that the mapping of VM and VMM are denoted as $A_g(V_g, P_g, X_g)$ and $A_h(V_h, P_h, X_h)$ respectively. And the shadow page table is referred to as $A_s(V_g, P_h, X_s)$. Then the access permission condition of shadow mapping X_s is the combination of guest mapping access permission X_g and host mapping access permission X_h , which can be proved as the following :

1. Shadow mapping A_s strictly corresponds to the guest mapping A_g that guarantees :

$$\forall v \notin V_g, Trans_{A_s}(v, x_s) \text{traps.} \quad (2.3)$$

2. Additionally, within the memory space of Shadow mapping A_s , guest kernels and guest user spaces are managed as native machines. Any accesses that is forbidden in the original mapping A_g remains forbidden in the shadow mapping. Therefore :

$$\forall v \in V_g, Trans_{A_s}(v, x_s) \text{traps when } Trans_{A_g}(v, x_g) \text{traps.} \quad (2.4)$$

3. Virtual machines and VMM are separated with access permissions. During the VM execution, any access to the VMM memory space will trap with an access-permission-fault exception. Therefore :

$$\forall v \in V_g, Trans_{A_s}(v, x_s) \text{traps when } Trans_{A_h}(p_g, x_h) \text{traps.} \quad (2.5)$$

Based on the above characteristics, the mapping in virtual machines can be fully virtualized in our approach. The VMM depends on the page-fault traps to emulate both access conditions X_g and X_h respectively. Specifically, access-permission failures are caused by illegal accesses outside the VM domain, and must be handled by the VMM itself. On the other hand, if traps are caused by DACR domains, they are mostly conflicting the guest mapping X_g and should be passed to the guest OS kernel.

Another classical challenge for address space virtualization is the coherency of cache memory and TLB when switching from one address space to another. In the ARMv7 architecture, the cache consistency is protected by hardware, since both instruction and data caches are physically-tagged. Thus, without the duplicate mapping that shares the physical memory, the memory space switch is spared of the expensive cache flush. ARMv7 also provides the *Address Space Identifier* (ASID) register to simplify the management of the TLB. Translations with different ASIDs are respectively labeled in TLB. Each address space is associated with one unique ASID value. The VMM reloads the ASID register whenever address spaces switch.

2.2.3.3 Memory Virtualization Context

The VMM emulates the virtual memory space for guest OS by maintaining the address space for each virtual machine. ARM processors use the CP15 coprocessor to control the behavior of the MMU, including enabling/disabling, loading the position of a translation page table in the Translation Table Base Register (TTBR), configuring the DACR register, etc. For each virtual machine, the VMM holds a replica of these resources, denoted as the memory space context. This context includes necessary resources to virtualize the memory space of the guest OS, i.e. (1) corresponding page tables, (2) TTBR, (3) DACR, (4) ASID. The virtual MMU (vMMU) emulates the behavior of the physical MMU. It manages the shadow mapping page tables and handles hyper-calls that guest OSs use to access page tables.

The VMM switches VM address spaces by saving and resuming the memory space context in the physical registers. The switching of address spaces may result in heavy overheads, since the cache miss rate and TLB miss rate will inevitably increase after the switching. In fact, such a cost is one of the most significant virtualization overheads, which will be discussed in Section 4.2.1.

2.3 Event Management

In general computing systems, an event is an action or occurrence that may be handled by software. Sources of events include the user, who may interact with software via I/O devices (for example, UART port, keystrokes on the keyboard, etc.). Another source is on-chip hardware peripherals such as system timers. Software can also trigger its own set of events, e.g. to communicate with other processes or other computers. Most modern OSs rely on events to schedule tasks and allocate resources. Software that changes its behavior in response to events is said to be *event-driven*, which often exists in scheduling algorithms or real-time systems. For example, system timer is commonly used by OS kernels to count time slices and perform scheduling, and real-time OS depends on the external events to perform real-time computation. In embedded systems, events can be categorized according to their purpose as the following types :

- **System events** are used to help the execution of OS, which are normally generated by software as synchronous notifications such as the timer tick for scheduler or the completion of certain tasks.
- **Peripheral events** are the ones that peripherals or I/O devices use to interact with the processor.
- **Communication events** are used to convey notifications to other systems, mostly in multi-core systems where one CPU can raise events to other CPUs for communication purposes.

Most events are implemented as interrupts, which will be raised to the processor and be handled via dedicated event handlers in OS. These handlers directly interact with

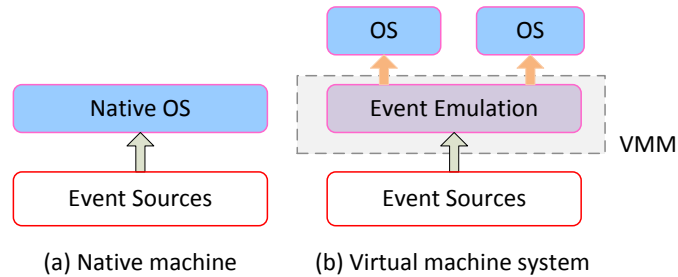


FIGURE 2.12 – Event management in native machines and virtual machine systems.

the sources of event by processing their requests and making responses, as shown in FIGURE 2.12(a).

In virtual machine systems, the management of events is much more complicated. Hardware resources, e.g. timers and peripherals may be shared by multiple virtual machines concurrently. As the intermediate layer between virtual machine and physical resources, the VMM must emulate these events to virtual machines, while guaranteeing that the events of each virtual machine are completely independent from other VMs, as shown in FIGURE 2.12(b). The virtualization policy of events may differ based on their applications in the system.

Events can be emulated following different principles. In existing technology such as XEN hypervisor, the VMM emulates the virtual machines events with additional mechanism. Guest OS receives software-emulated events with new ID numbers and attributes, which are handled through a dedicated mechanism called the *event_channel*. This channel wraps all physical events indifferently and re-generates virtual events. Such a mechanism forces the guest OS to abandon its original handlers and to adjust itself with the *event_channel*, and results in extra complexity in both VMM and guest OSs. Moreover, interrupts of different natures are under the same policy, which inevitably increases the efficiency.

Ker-ONE addresses this issue by an alternative approach. We attempt to emulate the physical events as the original ones, so that the local handling process in guest OS is respected. Interrupts with different purposes are distinguished and classified, so that we can propose dedicated mechanisms accordingly.

In this section we introduce the event emulation in Ker-ONE. First, we will present the mechanism of interrupt virtualization. Then we will propose the emulation of two special mechanisms, i.e. virtual timer and inter-VM communication.

2.3.1 Interrupt Virtualization

From the processors viewpoint, events arrive as interrupts, which are triggered by hardware sources. In Ker-ONE, the VMM manages all hardware interrupts. When running in a virtual machine, all interrupts are trapped to the VMM. The VMM first handles the physical interrupt by acknowledging the interrupt and clearing the source. Then, the

VMM sends a corresponding virtual interrupt to the targeted virtual machine if necessary. In this case, the hardware interrupts management is performed in the host space, so that the VMM remains in complete control of the hardware resources. Meanwhile, virtual machines receive and handle the emulated interrupts as if in native machine. This technique is called the interrupt virtualization.

2.3.1.1 Emulation of Interrupts

The ARMv7 architecture provides the Generic Interrupt Controller (GIC) to control interrupts [ARM13]. The GIC logically splits into an *Interrupt Distributor* block and one or more *CPU Interrupt Interface* blocks. The functionality of these two blocks is as follows :

- The ***Interrupt Distributor*** performs interrupts' prioritization and distributes interrupts to the target CPU Interrupt Interface blocks. The Distributor also holds the interrupt state and configurations (such as enable/disable, sensitive type, etc.).
- The ***CPU Interrupt Interface*** performs priority masking and preemption of interrupts. This interface raises the highest priority interrupt to CPU. Software interrupt handlers directly interact with the CPU Interrupt Interface for proper handling of an interrupt.

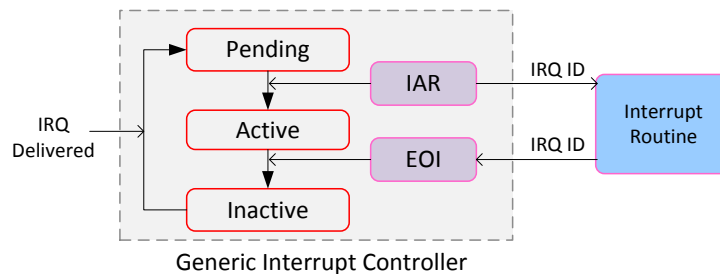


FIGURE 2.13 – The behavior and states of hardware interrupts managed by the Generic Interrupt Controller (GIC).

In a GIC, the behavior of each interrupt is controlled as a state machine (see FIGURE 2.13), in which three major states exist : *Inactive*, *Pending* and *Active*. Generally, interrupts are in the *Idle* state by default. when an interrupt arrives, it is distributed to the target *CPU Interrupt Interface* (changing state to *Pending*), and raises a hardware interrupt to the CPU. Then the software Interrupt Service Routine (ISR) is responsible for acknowledging the reception of an interrupt by accessing the Interrupt Acknowledge Register (IAR) (changing state to *Active*). After proper handling, the ISR should set the End of Interrupt register (EOI) to mark the end of the interrupt processing (changing state to *Inactive*). The typical software ISR is demonstrated in FIGURE 2.14(a).

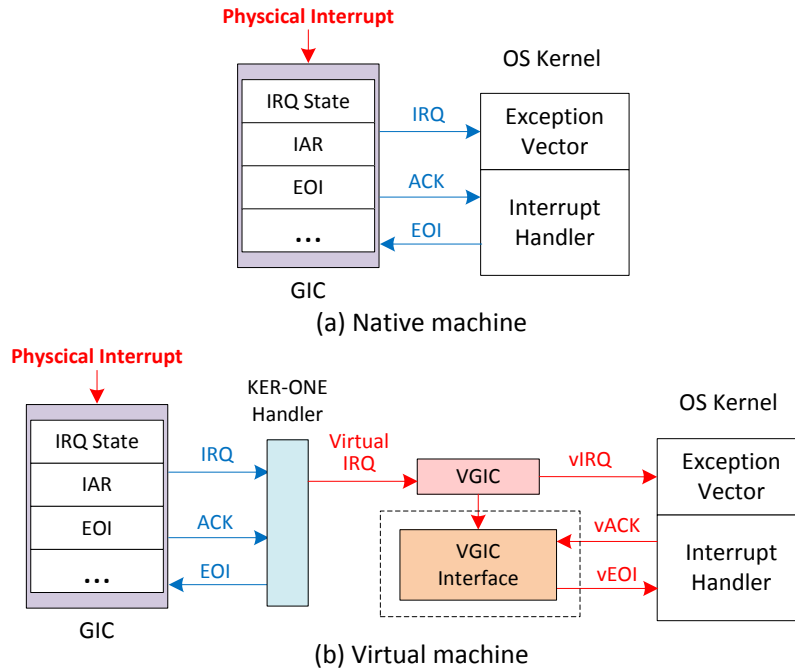


FIGURE 2.14 – The process of physical interrupts being handled and virtualized through the virtual GIC.

For virtual machines, virtual interrupts are raised by the VMM. In order to simplify the adjustment of the guest OS IRQ handler, we have decided to respect the guest OS original interrupt handling routine by providing a virtual interface of the GIC. The virtual content is identical to the physical GIC registers.

As shown in FIGURE 2.13(b), a custom virtual GIC (vGIC) is designed to emulate the management of interrupts. The vGIC holds the state of virtual interrupts for each VM, and emulates the behavior of the GIC by manipulating virtual interrupt states. A data structure, denoted as vGIC Interface, is implemented to hold the vGIC context. This structure holds virtual GIC registers including IAR, EOI, IRQ enable register, etc. Note that guest OS kernels may configure the vGIC and control virtual interrupts by directly accessing the vGIC Interface registers.

The flow in FIGURE 2.14(b) describes the processing of virtualized interrupts in guest OSs. When a physical interrupt occurs, the exception vector of Ker-ONE redirects them to the VMM for handling. The VMM handler completes the physical IRQ processing and raises the corresponding virtual interrupt in the vGIC. The vGIC then updates the virtual interrupt state and sends it to the VM by forcing it jump to its local exception vector. Then the guest OS kernel interacts with vGIC Interface to complete the handling. Note that, with this mechanism, the original OS IRQ handler only requires light modification of several code lines to re-direct GIC accesses. Thus the IRQ handler in the OS can be easily adapted.

Based on the vGIC, the states of virtual interrupts are consistent and independent in each VM. For example, a virtual interrupt can be disabled or masked by one VM, while the corresponding physical interrupt can still be collected by other VMs. A vGIC can also be programmed to directly send virtual interrupts into a specific virtual machine. This functionality is normally used by the VMM as a method of inter-VM communication (IVC), by allowing one virtual machine to raise interrupts to another one. This point will be detailed in Section 2.3.3.

2.3.1.2 Virtual Interrupt Management

In Ker-ONE, interrupt sources are categorized and divided into different priority layers according to their importance, as listed in TABLE 2.5. The highest interrupt priority level concerns the critical system interrupts. These are security-related interrupts that are only used by the VMM. An example of such an interrupt source is the VMM scheduling timer. Critical IRQs are always protected. Each VM is always associated to one interrupt layer, and is not allowed to manipulate interrupts in higher layers, so that events of higher significance remain unaffected.

TABLE 2.5 – List and description of interrupt priority layers.

Layer	Description	Priority Level
Micro-kernel	System critical interrupts	0-16
RTOS	RTOS Interrupts	16-64
GPOS	Non-critical interrupts	64-128
IVC	Inter-VM interrupts for communication	128

Interrupts used by guest OSs are divided into two levels : RTOS and GPOS. A RTOS is given a higher priority level than a GPOS, so that interrupts of a particular RTOS can neither be disabled nor be blocked by GPOSs, which guarantees that these events can be received by the RTOS even during the execution of GPOSs. This is quite important for ensuring the schedulability of real-time tasks, since RTOSs generally rely on real-time events, such as timer ticks or external interrupts to perform scheduling.

IVC refers to interrupt resources that are reserved for inter-VM communication, which are a group of software-generated interrupts in Ker-ONE. They are allocated with the lowest priority level given that inter-VM interrupts do not have to overtake guest OS local interrupts.

For each virtual machine, the context of virtual GIC is held in the *vGIC Context* data structure. This structure allows VMM to maintain the coherency of virtual interrupts when scheduling VMs.

2.3.2 Timer Virtualization

In many operating systems, reading and writing from (to) timers are common operations to schedule tasks and measure performances. For example, an OS often relies on timer ticks to determine if a specific task is ready to execute. Application workloads also often leverage timers for various reasons. In traditional virtualization, the physical timer resource is controlled by the hypervisor, and VMs are provided with software virtual timers, which have to be accessed by traps or hyper-calls. However, this method is likely to cause several problems. First, trapping to the hypervisor for each timer operation may incur noticeable performance overheads [DN14]. Second, the resolution of the VM timer is limited by the timer period of the hypervisor. For example, on a hypervisor whose timer period is set to $10ms$, guest OS with $1ms$ timer accuracy may work incorrectly.

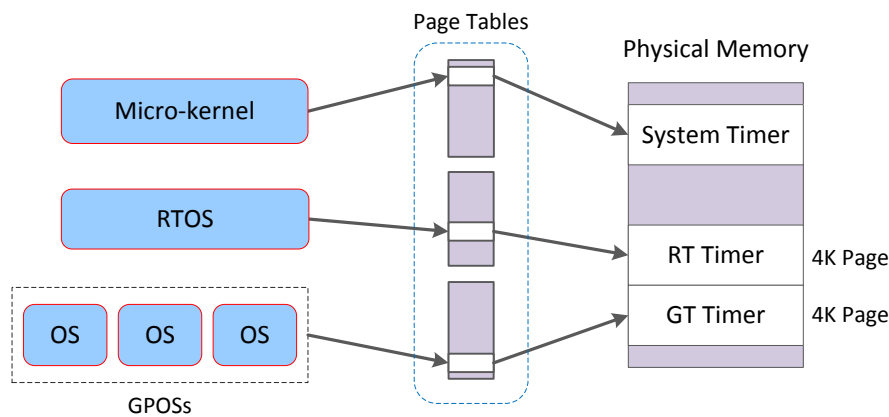


FIGURE 2.15 – Three independent physical timers are provided to the micro-kernel, RTOS and GPOSS respectively, which can be accessed directly. For one guest OS, only one timer interface is mapped in the page table.

As a solution, Ker-ONE allows VMs to directly access and program the timer without being trapped into the VMM. As shown in FIGURE 2.15, we use three independent physical timers for VMM, RTOS and GPOSS respectively, denoted as system timer, RT timer and GP timer. The ARM page table mechanism is leveraged to create independent timers for VMs. Note that each timer interface corresponds to a 4k memory page in the address space. For each VM, only one timer interface is mapped in its page table, so that it can only have access to the allocated timer. As depicted in FIGURE 2.15, system timer is reserved in the host space and can only be accessed by the micro-kernel. The RT timer is exclusively used by the RTOS VM. The GP timer is shared by multiple GPOSS VMs. The timer sharing is implemented by saving and restoring its register values for each VM. A guest OS is free to configure its timer, e.g. clocking period, interval value, interrupt. These configurations will be restored whenever a VM is scheduled.

One concern about this mechanism is the protection of the timer state when it is programmed by multiple users independently. In our system, the VM timer is implemented with the Triple Timer Counter (TTC), which is a peripheral timer provided in the ARM

architecture. The programming of the TTC is performed by discrete read/write operations on configuration registers. Therefore, the programming sequence can be interrupted anytime and continued later as long as the registers are properly resumed.

Compared to conventional software virtual timer solutions, our mechanism would slightly increase the VM switch overhead, since the timer registers must be reloaded at each VM switch. In our system, this involves a total number of 7 TTC registers. However, this overhead is negligible, considering that it also avoids frequent hyper-calls or traps, and simplifies the VM timer emulation mechanism.

2.3.3 Inter-VM Communication

An efficient communication mechanism inter virtual machines (IVC) is essential for most virtualization micro-kernels. The issue of IVC can be interpreted as the classical Inter-process communication (IPC) problem in micro-kernels. Since the work of L4 micro-kernels, great efforts of research have been given to minimize the IPC overheads for micro-kernel, which results in a greatly simplified synchronous IPC model, denoted as *fast IPC model* [Lie94], and has achieved high IPC performance. However, the synchronous IPC model has been aborted by some embedded micro-kernels because of its complexity, as in [Hei08]. In Ker-ONE, we use simple asynchronous communication methods instead of synchronous IPC model to achieve lower complexity. An IRQ-based IVC mechanisms is implemented in our system.

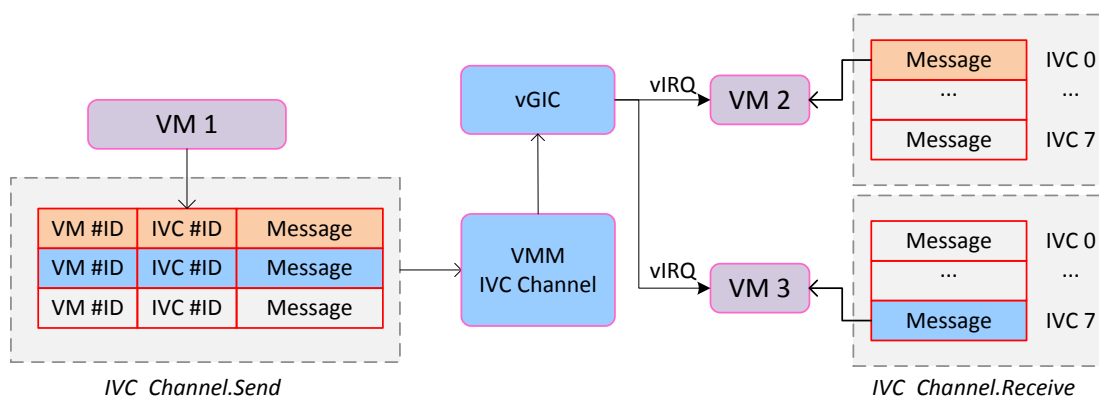


FIGURE 2.16 – The IVC mechanism leveraging VM/VMM shared memory region.

Ker-ONE leverages the VMM/VM shared memory region to facilitate asynchronous IVC. For each virtual machine, a shared memory page is created that can be accessed from both VMM and VM sides. A shared structure, *IVC_Channel*, is implemented in this region. This channel is composed of *IVC_Channel.Send* and *IVC_Channel.Receive*, which is depicted as in FIGURE 2.16. A total number of 8 IVC signals is reserved for the channel. To post IVC messages to other VMs, a virtual machine needs to create an entry in *IVC_Channel.Send* by indicating the target VM, the IVC number, and the message. Whenever the sender VM is scheduled out, the messages from *IVC_Channel.Send*

will be dispatched to the `IVC_Channel.Receive` of the target VMs. So that whenever these VMs are scheduled, Ker-ONE invokes the vGIC to send virtual interrupts to the target to acknowledge the arrival of IVC messages. In our case, these interrupts are reserved software-generated interrupts (SGI), which are configured with the lowest priority (TABLE 2.5). They will not preempt the VM local interrupts.

Note that, the sending and receiving of IVC mechanisms are performed with only several lines of read/Write instructions on the shared memory. Therefore, this approach is shorter and lightweight compared to the simplified *fast IPC model* in L4 micro-kernels.

2.4 Optimization

In the previous sections, we have described the mechanisms of virtualization that have been proposed in our system. As we can notice, to provide a fully-virtualized environment, it is essential to emulate physical resources by manipulating virtual resources : guest OS would access to these virtual resources frequently when executing kernel functions. Thus, the management of these resources may noticeably influence the performance of guest OS. In this section we discuss the management policies of virtual resources.

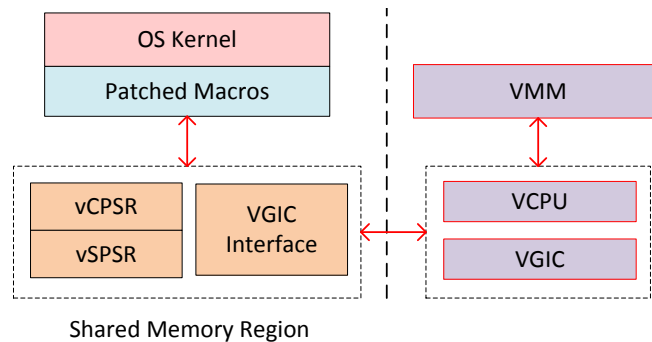


FIGURE 2.17 – Implementation of virtual PSR and vGIC interface in a VM/VMM shared memory region.

Here we focus on the most frequently used resources during guest OS execution. PSR registers, including CPSR and SPSR in an example of such resources. As already presented, this register configures the execution mode ($PSR.M$) and interrupt mask ($PSR.I$) of CPU, which will be frequently accessed and changed by OS to enter or exit the kernel space, or to execute critical functions. Moreover, CPSR/SPSR is saved/restored whenever a task is saved or resumed, which is a common operation in OS. Another virtualized resource that deserves special attention is the virtual GIC Context, which contains virtual registers emulating physical GIC. Each IRQ handling involves several read/write on these registers. In an *event-driven* OS, e.g. RTOS, the access to these resources may influence the IRQ handling overhead, which then increases the response time.

One simple and conventional solution for these resources is to hold them within the micro-kernel domain and make sure that they are exclusively manipulated by the VMM.

Guest OS accesses them via hyper-call interfaces, which calls the VMM to handle their requests. This approach is easy to implement since guest OS only need to replace the original store/load instructions with corresponding single-line hyper-calls. Nevertheless, compared to the original access process which is performed by a single instruction, trapping to the VMM is quite expensive and heavyweight.

An alternative policy is to implement virtual resources that are frequently used in the VM/VMM shared memory region. Guest OS access the virtual resources directly, and the VMM performs emulations according to the shared resource asynchronously. One obvious advantage of this policy is that a guest OS can perform operations on these resources without generating hyper-calls, which greatly reduces overheads. However, since this approach holds virtual resources in the virtual machine domain, it requires extra mechanisms at the VM side to properly access and protect these resources. This results in a considerable amount of functions and macros being added to the original OS code. Besides, the security of a VM is also undermined since these resources are no longer protected by the VMM, and thus can be threatened by malicious software. Any compromised task is able to freely modify the values of these resources. In TABLE 2.6, we compared the pros and cons of both policies.

TABLE 2.6 – Advantages and drawbacks of two policies for virtual resource management.

Policies	Performance	Source code modification	Security
Micro-kernel domain	low	low	high
Shared memory region	high	high	low

Despite that the use of shared memory region requires more effort to virtualize the guest OS, it is able to considerably shorten the execution path to access resources, which makes it a preferred optimization for virtualization technologies. To address this issue more clearly, we take the example of a common task-switch function in the OS kernel, during which the context of an old task is stored in the stack and a new task’s context is resumed. We have listed the pseudo assembly code in TABLE 2.7. Note that hyper-calls are relatively heavy operations because it involves entering and exiting the micro-kernel. With the virtual CPSR implemented in the shared memory region, we can avoid hyper-calls. We can also notice that the first approach causes only slight changes to the source code by simply replacing MCR/MRC instructions with hyper-calls, whereas the second approach has to introduce extra assembly macros to fulfill these operations.

In our system, virtual resources of PSR and vGIC are held in shared memory region to optimize the overall performance, whose implementation is shown in FIGURE 2.17. A shared data structure is created in VM domain, and registered in the VMM. The VMM can have knowledge of the current state of both virtual CPSR and vGIC by checking the shared structure when necessary. Dedicated macros (as in TABLE 2.7) are used to patch the source code of guest OS. This optimization results in a significant increase

TABLE 2.7 – Pseudo codes of the task context save/resume process with two policies. *Policy 1* : Virtual PSR is held in VMM domain and is accessed via hyper-calls. *Policy 2* : Virtual PSR is held in VM domain and accessed directly.

Function : OS Context Switch (OSCtXSw)		
Native Machine :		
STMFD	SP!,{R0-R12, LR}	@ Push current registers
MRS	R0,CPSR	@ Get current CPSR
STMFD	SP!,R0	@ Push CPSR value
...		@ Load new task stack
LDMFD	SP!,R0	@ Pop new task's CPSR
MSR	SPSR_cxsf, R0	@ Move CPSR to SPSR
LDMFD	SP!,{R0-R12, LR, PC}^	@ Pop new task's context, including CPSR
Virtualization Policy 1 :		
STMFD	SP!,{R0-R12, LR}	
Hypercall_MRS		@ Use hyper-call to get current virtual CPSR
STMFD	SP!,R0	
...		
LDMFD	SP!,R0	
Hypercall_MSR		@ Use hyper-call to resume virtual CPSR
LDMFD	SP!,{R0-R12, LR, PC}	
Virtualization Policy 2 :		
STMFD	SP!,{R0-R12, LR}	
Macro_MRS		@ Macro to get virtual CPSR : @ - Load vCPSR value ; @ - Get physical CPSR.aprs ; @ - Update vCPSR.aprs ;
STMFD	SP!,R0	
...		
LDMFD	SP!,R0	
Macro_MSR		@ Macro to resume virtual CPSR : @ - Change vCPSR value ; @ - Resume physical CPSR.aprs ;
LDMFD	SP!,{R0-R12, LR, PC}	

of virtualization efficiency. In Chapter 4 we will give extensive experiment results to demonstrate the performance of these two policies.

2.5 Real-time OS Virtualization

This section introduces the real-time support of our system. In terms of schedulability, a virtual system is a typical two-level hierarchical architecture, involving both intra-VM and inter-VM scheduling. Since real-time OSs demand that real-time tasks always meet their deadlines, the VMM scheduling mechanism must guarantee the execution of RT VM, while the application developers are responsible for defining a schedulable task set according to the proposed RTOS scheduling algorithm.

Generally, a RTOS hosts a set of periodic tasks T_k that can be characterized by a

periodic task model $T_k = T_i(e_i, p_i, d_i)$, where e_i is the worst-case execution time (WCET) with its period p_i ($p_i > e_i$) and relative deadline d_i with ($d_i > e_i$). The task set T_k is considered as hard real-time schedulable if for every period p_i , T_i can complete its execution e_i , within its deadline d_i . Otherwise the scheduling fails. For a hard real-time system, missed task deadline results in the failure of the whole system.

For a virtualized RTOS, being hosted in a virtual machine causes extra overheads to e_i , and the CPU bandwidth of a RTOS is influenced by the VMM scheduling. This may result in the collapse of a tasks set that is actually schedulable on a native machine. Therefore a dedicated VMM scheduler is required to support RTOS tasks in virtual machines.

2.5.1 VMM Scheduler

The Ker-ONE VMM scheduler is intended to support real-time virtual machines with low complexity. In this work, we mainly focus on two characteristics : the scheduling accuracy for real-time tasks and the algorithm complexity for implementation. As introduced at the beginning of this chapter, we focus on a VMM which hosts one RTOS and several GPOSs. The RTOS tasks are considered as critical with soft real-time constraints.

For the Ker-ONE scheduler, we have discarded some well-known algorithms such as the compositional scheduling algorithm, since we would like to avoid complex algorithms that require additional computation for PRM models [YKP⁺11] or modification of the OS original scheduling interface [YY14]. The purpose of the Ker-ONE scheduler is to allow RT tasks to be scheduled by the original RTOS scheduler with negligible virtualization overheads.

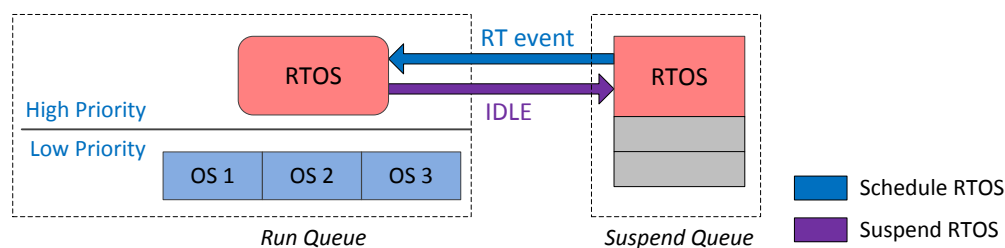


FIGURE 2.18 – RTOS Priority-based scheduling mechanism with independent physical timers. An RTOS is added to the run queue and preempts the other OS when RTOS events occurs.

The VMM scheduler is based on a priority-based preemptive round-robin strategy. It includes two scheduling lists : a *run queue* and a *suspend queue*. The *run queue* contains all currently executable components, and the *suspend queue* holds the idle components. For every scheduling decision, the VMM scheduler always selects the highest-priority VM in the *run queue* to execute. Among VMs that have the same priority level, the CPU is shared according to a time-slice-based round-robin policy. The VMM allocates each VM a determined quantum of CPU time denoted as time slice, which expires during the VM

execution. Whenever the budget is consumed up, the VM is replenished with full time slice and moved to the end of the round-robin circle. The higher priority VMs can always preempt the lower ones as long as they are included in the *run queue*.

The proposed scheduling mechanism is demonstrated in FIGURE 2.18. The VMM always removes the currently running RTOS from the *run queue* when it is IDLE, allowing the lower-priority non-RT OSs to run. In this case, the non-RT OSs budgets are consumed. The non-RT OSs continue their execution in a round-robin way until any event occurs in the RTOS. In this case, the VMM adds the RTOS back into the *run queue* and reschedules all VMs. With this mechanism, the workload of higher-priority VM (i.e. RTOS) is not affected because only its IDLE time is used by the other VMs.

In order to implement this scheduling strategy, we have added a hook routine to the IDLE task of RTOS, which is composed of the hyper-call that acknowledges the VMM about the idle state of the RTOS virtual machine, so that lower priority VMs are rescheduled to run. Note that in modern ARM processors, the instructions of Wait-for-Interrupt (WFI) or Wait-for-event (WFE) are normally used in the IDLE task, in order to lower the CPU power consumption. In Ker-ONE, these instructions are no more valid since the idle time is donated to other software.

2.5.2 RTOS Events

RTOS rely on events to perform tasks scheduling. Events include timer ticks that help keeping track of the execution time and other exceptions or interrupts. For most RTOS schedulers, timer ticks are important to determine whether a task is ready to run or not. In virtualization, a guest OS executes in virtual time instead of physical time. For VMs, the virtual time is measured by counting the virtual timer ticks that it receives. Therefore, in a VM, the difference between physical time and virtual time can be quite significant if the VMM generates virtual timer ticks only when this VM is scheduled.

However, in real-time virtualization, a guest RTOS should be aware of the physical time (i.e. actual execution time) to guarantee the scheduling of its tasks. In section 4.4 we have introduced that an independent physical timer, RT timer, is allocated to RTOS with direct access. Recall that we have divided IRQ resources into several priority levels (as in TABLE 2.5). RT timer has higher priority level than GPOSs. Thus, even when RTOS is not scheduled in CPU and GPOSs are running, RT timer interrupt can still be received by the VMM and get delivered to the RTOS. In this case, RTOS can set RT timer as its scheduler requires without worrying about inter-VM scheduling, and the original OS scheduler remains unchanged.

For example, at some time t_0 , the RTOS scheduler sets the RT timer to trigger after T seconds and goes into the IDLE state, which causes the VMM to reschedule GPOS A into the RUNNING state. Then after T seconds, the RT timer triggers an interrupt while GPOS A is still running. However, since the RT timer IRQ has a higher priority, it will preempt the execution of GPOS A so that the RTOS's real-time scheduling is respected.

2.5.3 RTOS Schedulability Analysis

Though Ker-ONE is designed to minimize the extra latency cost on guest RTOSs, the virtualization environment will inevitably degrade performance. Therefore, regarding the schedulability of the RTOS tasks, it is crucial to take into consideration the cost model that formalizes the extra virtualization-related overheads.

Normally in a complete cost model, the actual execution time that a task T_i needs, consists of the WCET e_i and the latency of its being released or scheduled. Thus, the extended model for the actual execution time E_i is :

$$E_i = e_i + relEv, \quad (2.6)$$

$$\text{and } relEv = \Delta^{event} + \Delta^{rel} + \Delta^{sched} + \Delta^{cxs},$$

where e_i is adjusted with the tasks' *Release Event* overhead $relEv$, which consists of the event latency Δ^{event} , the release overhead Δ^{rel} , the scheduling overhead Δ^{sched} and the context switch overhead Δ^{cxs} .

Virtualization overheads have three sources. First, when an event (e.g. RT timer tick) targets an RTOS, it may not be delivered immediately. This is typically the case if the VMM is in critical state such as handling hyper-calls. In this case, the arrival of an event is delayed for the RTOS. Second, if an RTOS is to preempt other VMs, the VMM scheduler takes additional time to reschedule the RTOS and to switch between VMs. Third, the release of a task on a RTOS causes more overheads than on a native machine due to the extra costs of the environment virtualization, e.g. the instruction emulation and interrupts virtualization. Even the execution time e_i is affected. FIGURE 2.19 describes how virtualization affects the execution time. The negative effect of virtualization can be included into the tasks practical execution time : E_i^{VM} (see Eq. (2.7)).

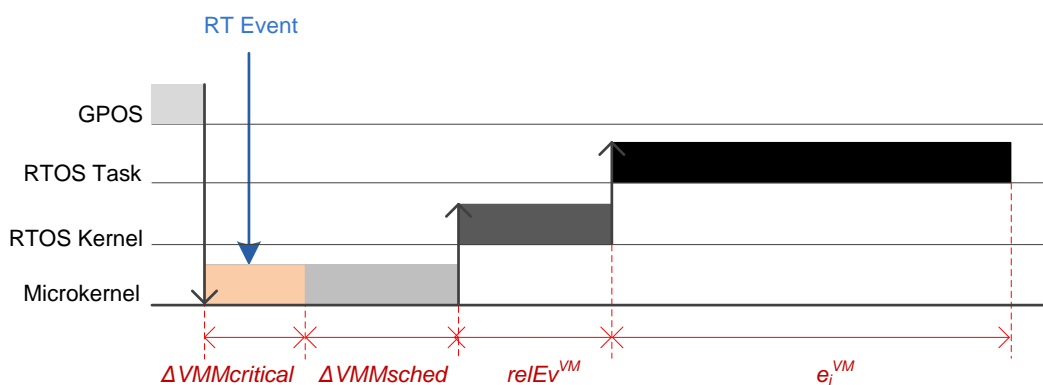


FIGURE 2.19 – Overhead of an RTOS task execution in VM, which is composed of VMM critical overheads, VMM scheduling overheads and RTOSs intrinsic overheads.

$$\begin{aligned}
E_i^{VM} &= e_i^{VM} + relEv^{VM} + \Delta VMMsched + \Delta VMMcritical, \\
\text{and } e_i^{VM} &= e_i^{Native} + \Delta_{VM}^{ei}, \\
\text{and } relEv^{VM} &= relEv^{Native} + \Delta_{VM}^{relEv},
\end{aligned} \tag{2.7}$$

where e_i^{VM} and $relEv^{VM}$ are actual T_i execution time and Release Event overheads in a VM, which are respectively inflated with the Δ_{VM}^{ei} and Δ_{VM}^{relEv} overheads. $\Delta VMMsched$ is an extra overhead required for RTOS scheduling. And $\Delta VMMcritical$ is the delay caused by the VMM critical execution. In fact, the VMs' response time for RTOS events can be represented as :

$$Response^{VM} = relEv^{VM} + \Delta VMMsched + \Delta VMMcritical. \tag{2.8}$$

From $Response^{VM}$, the extra VM overhead caused to the task response time can be derived as :

$$\begin{aligned}
\Delta_{VM}^{Response} &= Response^{Native} - Response^{VM} \\
&= \Delta_{VM}^{relEv} + \Delta VMMsched + \Delta VMMcritical,
\end{aligned} \tag{2.9}$$

And the overall virtualization overheads then can be obtained as :

$$\begin{aligned}
E_i^{VM} &= E_i^{Native} + \Delta_{VM}, \\
\text{and } \Delta_{VM} &= \Delta_{VM}^{ei} + \Delta_{VM}^{Response},
\end{aligned} \tag{2.10}$$

where E_i^{Native} refers to the native execution cost E_i . Therefore, for the task set T_k that was verified as schedulable in the original RTOS, the adjusted model $T_k = T_i(E_i^{VM}, p_i, d_i)$ has to be re-verified for virtualization according to Δ_{VM} . In fact, the value of Δ_{VM} evaluates the influence of virtualization on the original tasks. As we can notice, such an influence is caused by the extra overheads of response time and execution time. This model is depicted in FIGURE 2.20.

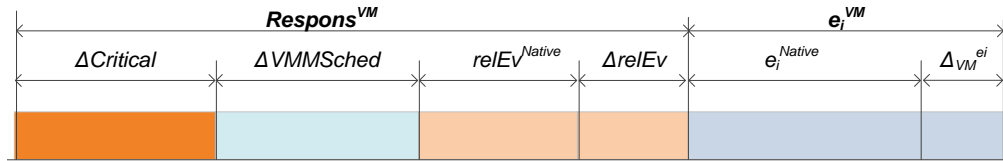


FIGURE 2.20 – Composition of overall execution time in the virtual machine context, which is composed by the overheads of response time ($Response^{VM}$) and execution time (e_i^{VM}).

Moreover, in a real implementation, the quantization issue has to be discussed. Given that RTOS scheduling is performed according to scheduling ticks, all schedule parameters such as e_i and p_i are expressed as numbers of ticks. In practical scheduling, we define the

effective execution time E'_i for T_i , as a number of ticks Θ_i . This represents the minimal tick-quantized execution time of E_i^{VM} , as follows :

$$\begin{aligned}\Theta_i &= \lceil \frac{E_i^{VM}}{\Delta^{Tick}} \rceil \\ &= \lceil \frac{E_i^{Native} + \Delta_{VM}^{ei} + \Delta_{VM}^{Response}}{\Delta^{Tick}} \rceil,\end{aligned}\tag{2.11}$$

where Δ^{Tick} stands for the schedule tick interval of the RTOS. From our overhead measurement, the cost of $\Delta_{VM}^{Response}$ is estimated to be less than few microseconds. Since an RTOS normally sets the schedule ticks to several milliseconds, the extra response overhead for RTOS tasks can be negligible due to the significant timing difference (several orders of magnitude). The Δ_{VM}^{ei} cost, on the other hand, depends on the type and workloads of tasks. The detailed evaluation and measurement results are given in Section 4.2.

Some researchers also pay attention to the quantization overhead in an RTOS, i.e the difference between the RTOS execution time based on the number of ticks, and the physical time. However, since Ker-ONE uses an independent real time timer, the quantization overheads for virtualized RTOS's remain the same as in the native situation, and will then not be discussed in the thesis.

Another interesting issue deals with the schedulability of multiple RTOSs. Currently Ker-ONE only hosts a single RTOS. With multiple RTOSs run with the same priority level, RTOS events may not be processed in real-time. Therefore the scheduling strategy has to change to respect the deadlines of real-time tasks.

2.6 Summary

In this chapter we introduce the details of the Ker-ONE micro-kernel. We first explained the motivation and scenario of our research, which is intended for small-scaled embedded systems. Based on this scenario, we have claimed the assumptions and limitations of our micro-kernel, and gave an overall introduction of the kernel architecture.

Then we described the virtualization approach from two aspects. We introduced the virtualization mechanism for resources of the processor, the floating-point co-processor and the virtual machine memory space. The major concern of these resources is to provide secure and continuous maintenance so that virtual machines have consistent and independent execution environments. On the other hand, in events virtualization, we pay much attention to the emulation and control of behaviors since they influence the behavior of virtual machines. We described our emulation policies for virtual interrupts, timers and IVCs. We also discussed the optimization approach by introducing shared memory regions. Both pros and cons were fully discussed in this part.

The last part of this chapter focuses on the real-time support of Ker-ONE. We proposed a real-time virtualization mechanism with an independent RT timer and preemptive scheduling. Based on a simplified task model, we have analyzed how our approach

would influence the schedulability of guest RTOS in Ker-ONE. A complete model was built to express this impact.

In the next chapter, we demonstrate the extension of Ker-ONE, which focuses on supporting DPR accelerators in virtual machines. We will discuss the problems of DPR implementation, resource allocation, virtual machine synchronization and software coding that occurs during the sharing of DPR resources among independent virtual machines.

CHAPTER 3

DYNAMIC MANAGEMENT OF RECONFIGURABLE ACCELERATORS ON KER-ONE

In the last decade, the research on CPU-FPGA hybrid architectures has become a hot topic. One of the main challenges in this domain consists in efficiently and safely managing dynamic partial reconfiguration (DPR) resources. In this chapter we introduce the management framework of the reconfigurable accelerators by the Ker-ONE micro-kernel. Note that one of the strongest motivation of using virtualization is the isolation among components. Based on this feature, virtual machines access resources independently, being unaware of the existence of other VMs. The purpose of our framework is to provide an abstract and transparent layer for virtual machines to access reconfigurable resources. The underlying infrastructure of partial reconfiguration management is hidden from the virtual machines, so that the software developers do not need to consider the implementation details. In this section, we first introduce the hardware platform of Zynq-7000, including its architecture and reconfiguration features. We then propose a framework where DPR accelerators are presented as virtual devices, and are universally mapped in each VM space as ordinary peripheral interfaces. The framework automatically detects VM's request for DPR resources and handles these requests dynamically according to a preemptive allocation mechanism. In this chapter we also address another issue that is the security when sharing PR resources among VMs. Our framework guarantees that VMs and DPR modules are exclusively connected, and that DMA accesses are restricted in isolated VM domains. In the end of this chapter, we will discuss the employment policy of DPR accelerators for guest OS applications.

3.1 Introduction to the Zynq-7000 platform

The Zynq-7000 family is based on the Xilinx All Programmable SoC (AP SoC) architecture. This platform integrates a feature-rich dual-core ARM Cortex-A9 MPCore

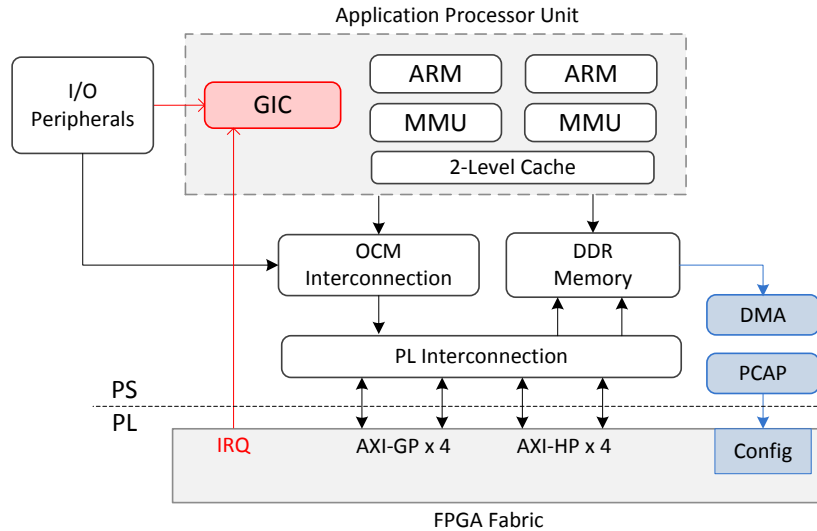


FIGURE 3.1 – The overview architecture of Zynq-7000 ARM-FPGA hybrid platform.

based processing system (denoted as PS) and Xilinx programmable FPGA fabric logic (denoted as PL) in a single device. ARM CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals. The PL includes mainly the FPGA fabric which can be completely reconfigured or used with partial, dynamic reconfiguration (PR). PR permits the configuration of a portion of the PL. This enables optional design changes such as updating coefficients or the time-multiplexing of PL resources by swapping in new algorithms as needed. This capability is analogous to the dynamic loading and unloading of software modules.

In FIGURE 3.1, we present the architecture of this hybrid platform with the functional blocks of the system. The PS and the PL are on separate domains, being connected with various interfaces and signals. The PS part includes mainly the Application Processor Unit (APU), the on-chip DDR memory, I/O peripherals (IOP) and interconnections, whereas the PL side consists of the FPGA fabric and PS/PL interfaces.

3.1.1 PS/PL Communication

The Zynq-7000 device builds up the communication bridge between PS and PL by implementing several types of interfaces which are based on a standard AXI protocol. These AXI interfaces are implemented with different features, providing various communication options. Three types of AXI interfaces are provided in our platform :

- **AXI General-Purpose Interfaces (AXI-GP)** is directly connected to the PS-PL interconnection net and is general-purpose. It is aimed to provide simple communication between PS and PL, and is not intended to achieve high performance.
- **AXI High-Performance Interfaces (AXI-HP)** provides PL bus masters with

high bandwidth datapaths to the system memories. Each interface includes two FIFO buffers for read and write traffic. It can be programmed as 32-bit or 64-bit to perform high-speed data exchange between PL and PS parts.

- **AXI Accelerator Coherency Port (AXI-ACP)** is a 64-bit AXI interface that allows the PL to access the CPU L2 cache while maintaining memory coherency with CPU L1 caches. This interface is mostly used in computations where software and FPGA accelerators are closely connected.

In TABLE 3.1, we present the technical details of three AXI interface types. We can notice that AXI interfaces can be implemented as bus masters or slaves on both PL or PS side. Most interfaces, i.e. AXI-HP and AXI-ACP, support only master modes on the PL side, meaning that the FPGA is granted to access to system memory implicitly. The only way that enables PS to access PL directly is through the AXI-GP interface, which is quite essential to control FPGA accelerators. As shown in FIGURE 3.2, the accesses on AXI-GP are led via the OCM interconnection, which are implemented as a unified memory space for MMU. Thus, through AXI-GP, the CPU regards PL as physical memory space. A reserved address space, 0x40000000 to 0xBFFFFFFF, is allocated to AXI-GP accesses. Within this domain, a user is free to map PL registers into physical addresses, which will then be managed by the MMU. Ker-ONE leverages this feature to create isolated accelerator domains for virtual machines, which will be introduced in later sections.

TABLE 3.1 – Technical details of the AXI interface

AXI Type	Num	Mode(PL side)	Throughput	Width	Usage
AXI-GP	4	2 Master, 2 Slave	600MB/s	32-bit	Simple access
AXI-HP	4	4 Master	1200MB/s	32/64-bit	Burst data transfer
AXI-ACP	1	1 Master	1200MB/s	32/64-bit	Burst transfer with cache coherency

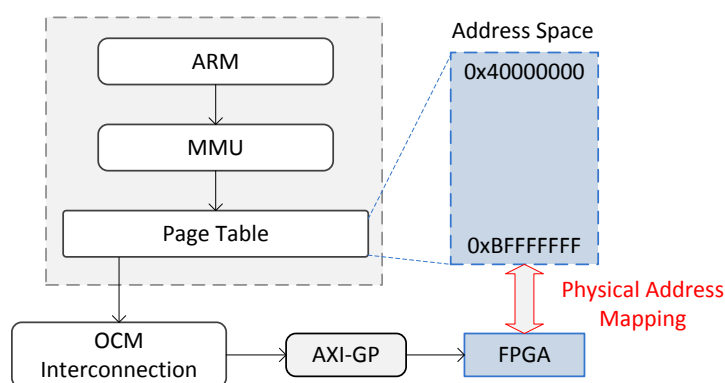


FIGURE 3.2 – The physical address mapping of PL resources via the AXI-GP interface.

3.1.2 Partial Reconfiguration

With the partial reconfiguration (PR) technology, the FPGA fabric on the Zynq-7000 device can be partially reconfigured at run-time while the rest of the logic continues running. PR occurs in a reconfigurable partition (RP), which is an independent reconfigurable region that is divided into areas via partitioning. An RP is often denoted as partial reconfiguration region. Each RP is composed of several frames, which are the smallest reconfigurable regions within an FPGA device. Frames may contain different logic resources, such as Look-Up Tables (LUTs), Digital Signal Process(DSPs) or RAMs. The logic of an RP is determined by a Reconfigurable Module (RM), which is the netlist or an HDL description that is implemented when instantiated in an RP. There may be multiple RMs for one RP. Other FPGA logical elements that are not part of a Reconfigurable Partition are denoted as static logic. These logical elements are never partially reconfigured and are always active when RPs are being reconfigured.

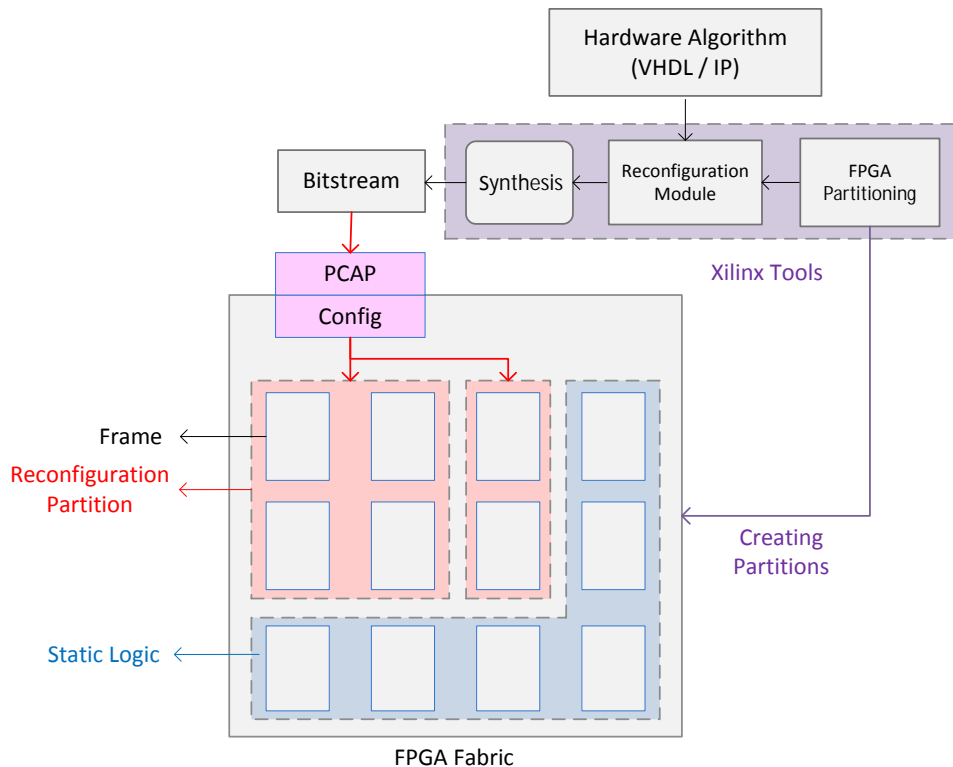


FIGURE 3.3 – The partial reconfiguration technology on Xilinx FPGA.

The RP regions in FPGAs are determined before the system get launched, and cannot be modified during execution unless re-implementing the whole fabric. Targeted hardware algorithms and logic on each RP have to be implemented as Reconfigurable Modules which will then be translated into actual configuration information such as a netlist. Xilinx development software tools such as PlanAhead and Vivado are in charge of creating

Reconfiguration partitions, synthesizing RMs and generating configuration files as bitstream files. Note that the size of a bitstream is solely determined by the RP, and varies depending on the amount and type of resources used by this RP. It does not depend on the actual algorithm which means that different algorithms RMs on the same RP may end up with identical bitstream size. In FIGURE 3.3, PR technology concepts are illustrated.

Reconfiguration is performed by downloading bitstream files into the corresponding Reconfiguration Partitions. Two downloading methods are supported on the Zynq platform. As shown in FIGURE 3.3, by default, a Processor Configuration Access Port (PCAP) is implemented in PS. The PCAP is connected to the Device Configuration Interface (DevCfg), which is a DMA controller. Through the PCAP interface, DevCfg is enabled to launch bitstream transfers from system memory to PL by commanding DMA transfers. The advantage of PCAP is that it provides a light API interface for software, and extensive built-in features to control the reconfiguration, including interrupts and encrypted transfers. Besides, PCAP provides a high transfer bandwidth as 130MiB/s, which is sufficient in most scenarios. Bitstreams can also be downloaded via Internal Configuration Access Port (ICAP), which is a more traditional technology from earlier Xilinx products, e.g the Virtex family. ICAP is implemented on the FPGA fabric and allows PL to perform self-configuration from external memories. ICAP can be customized to achieve high-bandwidth data transfer, which has been studied in several works [HGNT10][LKLJ09]. However, ICAP costs extra FPGA resources to get implemented. Moreover, ICAP lacks the versatility of PCAP which makes it convenient for software development. In our system, PCAP is used to perform reconfiguration.

3.1.3 Interrupt Sources

As shown in FIGURE 3.1, PL is able to assert asynchronous interrupt signals to CPUs. Up to 20 interrupt sources are reserved for PL. These interrupt signals are routed to the Generic Interrupt Controller, where each interrupt signal is set to a priority level and managed as a normal interrupt (see Section 2.3.3). However, only PL decides whether to use these interrupts and the way to use them, which requires custom logic from developers. The designation of PL interrupts must guarantee that the interrupts can be manually enabled, disabled and cleared by software.

3.2 DPR Management Framework

To facilitate the application of DPR accelerator on virtualization, Ker-ONE provides a management framework based on the PR features of the Zynq platform, which is aimed to dynamically allocate DPR resources for virtual machines at run-time. This section introduces the proposed framework in a bottom-up sequence. We start with the lower physical layer of DPR resources, and then move up to describe the allocation mechanism and the behavior of the software manager. We also discuss HW/SW security issues and isolation in this framework.

3.2.1 Framework Overview

In this part, we first present the terminology applied in this thesis. As introduced in Section 3.1.2, reconfigurable modules (RM) are implemented in pre-determined reconfiguration partitions (RP), by downloading corresponding bitstream files via PCAP transfer. In our designation, RP is denoted as partial reconfiguration regions (PRR) and RM is denoted as HW task.

HW tasks provide accelerators for various functions and algorithms. Each function or algorithm is denoted as a virtual device (VD) and is completely isolated from the implementation details. Therefore, one specific device can be implemented by different HW tasks in different PRRs. To better clarify this concept, an example is presented in FIGURE 3.4. Let us consider two reconfigurable areas in the FPFA fabric (PRR #1 and PRR #2). In addition, four HW tasks `pr1_fft256`, `pr1_fft512`, `pr2_fft256` and `pr2_fft128` are designed to work in PRR #1 and #2 respectively. In this system, three virtual devices are available : FFT512, FFT256 and FFT128. In this example, the virtual device FFT256 can be implemented in both PRR #1 and #2, and PRR #1 is able to host two virtual devices, i.e. FFT512 and FFT256.

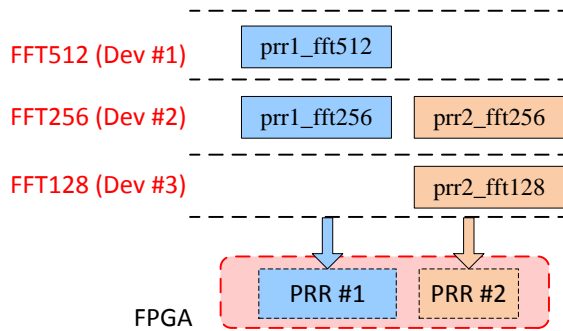


FIGURE 3.4 – HW tasks and the implementation of virtual devices.

Virtual devices are mapped to the fixed addresses in all guest OS address spaces, and are considered as ordinary devices. A unified interface, i.e. a standard structure of registers, is provided to users. Like other peripherals in ARM systems, user software accesses these devices by reading/writing at the address of the corresponding device interface. Meanwhile, the physical positions of these virtual devices are not known since they can be implemented in different PRRs.

Therefore, we have introduced an intermediate PR interface (IF) on the FPGA side, which can be seen as an intermediate layer between the logical virtual devices and the actual reconfigurable accelerator modules. These IFs are in charge of connecting the virtual machines with accelerator modules so that software can control them as peripheral devices. Each IF is exclusively associated to a specific virtual device. Thus, allocation of DPR resources is performed in two steps : first, the IF is mapped to the VM address space as a virtual device interface. Second, on the FPGA side, the IF is connected to the target PRR that implements the corresponding device function.

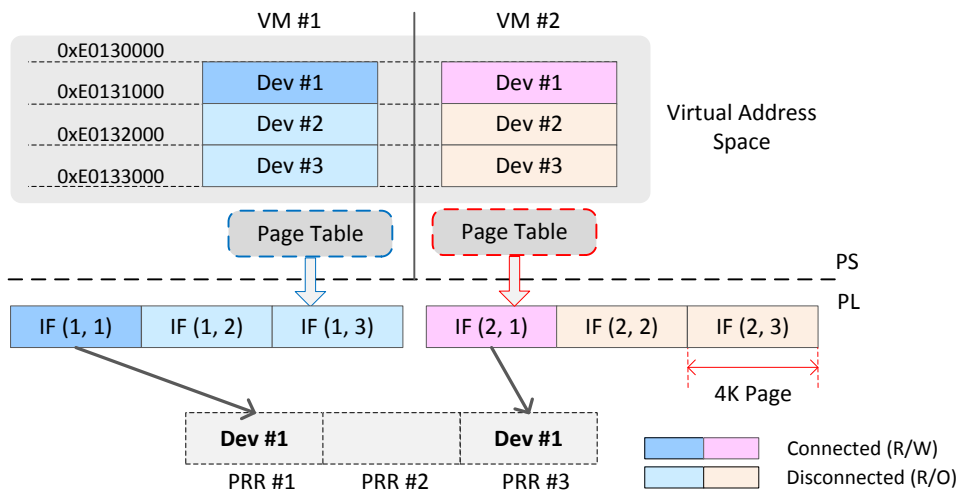


FIGURE 3.5 – Allocation of virtual devices for virtual machines via manipulating the mapping of IFs.

FIGURE 3.5 describes how IFs are used to connect virtual devices in the VM space with PR accelerators. IFs are initiated on the hardware side, whose physical address are configured to be aligned to independent 4KB memory pages. VMs access IFs via independent page tables, which maps IFs as memory pages in virtual address space. Therefore, though a virtual device is mapped to the same virtual address across VMs, in physical layer it is implemented by using separated IFs in the FPGA.

As shown in FIGURE 3.5, the mapping between a particular IF and the VM space of a virtual device is fixed. An IF has two identifiers, vm_id and dev_id (i.e. referred to as $IF(vm_id, dev_id)$) to help identify the virtual machine and the virtual device to which it is associated.

An IF has two states, *connected* to a certain PRR or *unconnected*. When an IF is *connected*, it is considered that the corresponding virtual device is implemented in the PRR and that it is ready to be used. Being in the *unconnected* state means that the target accelerator is unavailable. Once connected, a virtual machine can control PR accelerators by manipulating IF registers. On the other hand, for unavailable devices (with an *unconnected* IF), the IF registers are mapped as read-only pages, and a VM cannot configure or command this virtual device by writing to its interface. This mechanism guarantees the monopoly use of PR resource at any time.

We can take FIGURE 3.5 as an example of this mechanism. In VM #1, an application is free to program and command Dev #1 as the IF associated with it is currently connected to PRR #1, where the device accelerator is implemented. Meanwhile, VM #1 cannot give orders to Dev #2 and #3 since these interfaces are currently read-only. Any writing on these interfaces will cause a page-fault exception to the VMM, and will help VMM detect the VM's request for virtual devices.

One major characteristic of virtualization is that virtual machines are totally isolated

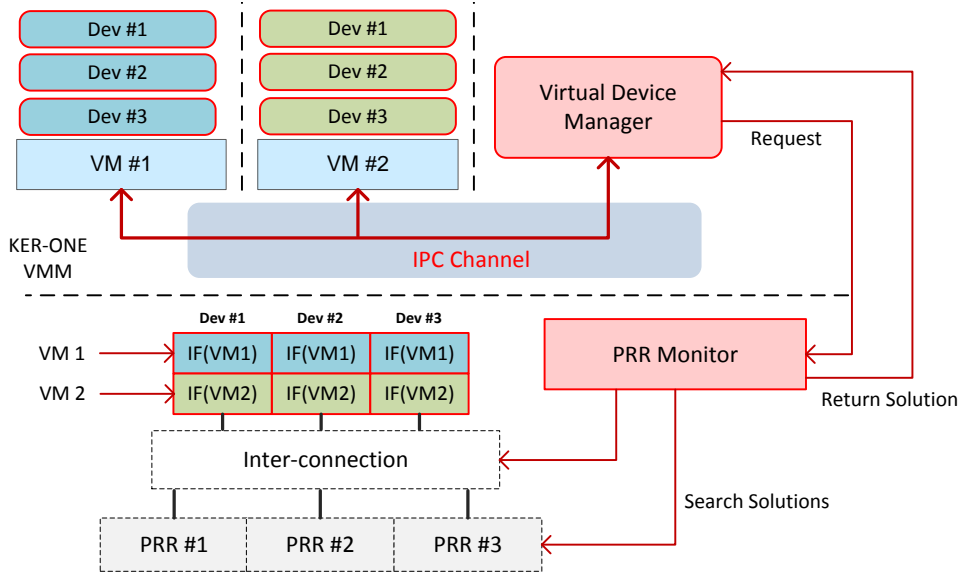


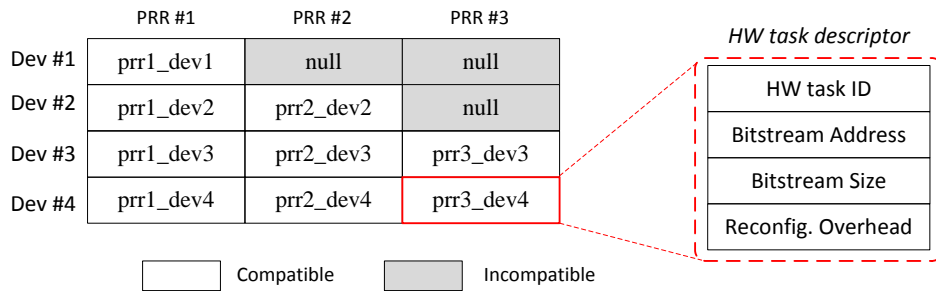
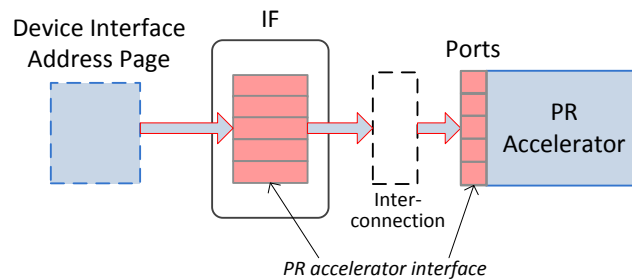
FIGURE 3.6 – Overview of the DPR management framework in Ker-ONE.

from each other. In our system, VMs are also completely isolated from PR accelerators. This can obviously lead to resource sharing issues that are well known in computing systems. In traditional OS, such problem can be solved by applying synchronization mechanisms such as semaphores or spin-locks. For Ker-ONE however, such mechanisms are not suitable since they may undermine the independence of VMs. Therefore, our system introduces additional management mechanisms to dynamically handle virtual machines' request for PR resources. Note that such requests can occur randomly and are unpredictable.

In Figure 3.6 the proposed management framework is described. A *Virtual Device Manager* and a *PRR Monitor* component are made available in both software and hardware parts of the FPGA. The *Virtual Device Manager* is a particular software service implemented in an independent virtual machine domain. It automatically detects the requests coming from virtual machines that want to use the virtual devices, and handle these requests by allocating DPR resources. The inter-communication is performed through the IPC mechanism on Ker-ONE. In the static part of the FPGA, a *PRR Monitor* is created and is in charge of maintaining the connections between IFs and PRRs. This monitor runs concurrently with software to dynamically monitor reconfigurable accelerators and search for available solutions of PR requests. It communicates with the *Virtual Device Manager* synchronously with dedicated interface and interrupts.

3.2.2 Hardware Task Model

HW tasks are associated with PRRs. Hence, there may be several HW tasks implementing the same algorithm (i.e. virtual device), targeting different PRRs. In this

FIGURE 3.7 – HW task index table with *HW task descriptors*.FIGURE 3.8 – Implementation of the *PR accelerator interface* for virtual devices.

case, a given PRR may not be compatible with some virtual device, if its area (i.e. resource amount) is insufficient to implement the corresponding function. Therefore, the compatibility information of HW tasks must be initialized beforehand.

As listed in Figure 3.7, an HW task index table is created to provide a quick look-up search for HW tasks. In this table the compatible virtual devices for each PRR are listed. For each compatible virtual device, a *HW task descriptor* structure is given, which holds the information of the corresponding bitstream file, including its ID, memory address and file size. This information is used to correctly launch PCAP transfers and perform reconfiguration. This table also holds the reconfiguration overheads of each HW task, whose values can be precisely estimated via previous measurements.

Virtual machines access HW tasks via IFs. We proposed a standard interface to facilitate the multiplexing of PR modules, denoted as *PR accelerator interface*. The implementation of this interface is shown in FIGURE 3.8. It is included in both IFs and HW tasks, and conveys the register values from the IF to HW tasks. Once the IF is connected to an HW task, a virtual machine can write commands or configurations into the IF registers to control the HW task behavior.

In TABLE 3.2 the structure of the *PR accelerator interface* is listed. Virtual machines start the process by setting the *START* flag. When the required computation is over, the *OVER* flag is set and the result is returned in the *RESULT* register. HW tasks can be programmed to perform DMA data transfers via the AXI-HP interface to exchange massive data with VM memory. PL Interrupts can also be generated to acknowledge

TABLE 3.2 – List and description of ports in *PR accelerator interface*.

Register	Width	Description
STAT	32-bit	HW task status register
START	8-bit	Start flag
OVER	8-bit	Over flag
CMD	32-bit	Command register
DATA_ADDR	32-bit	Data buffer address register
DATA_SIZE	32-bit	Data buffer size register
RESULT	64-bit	Computation result register
INT_CTRL	32-bit	Interrupt controller register
Custom Ports	8*32-bit	Provide 8 IP-defined ports

critical events to VMs, like errors or completion. From FIGURE 3.8, we should note that a *PR accelerator interface* structure of registers is implemented in IF. When an accelerator is disconnected with a PRR, the states of virtual device execution (e.g. results, status) are still stored in this structure in IF, so that the VM can restart from the interrupt point of the virtual device when it gets re-allocated. In this way, the consistency of the virtual device interface is guaranteed.

The VMs that are currently using HW tasks are denoted as *clients*. HW tasks inherit the priorities of VM *clients*. We use preemptive policy for HW tasks, meaning that the HW task corresponding to the low-priority VMs can be forced to stop and get replaced by the desired HW task corresponding to the VM of higher priority.

This raises another issue that is data integrity. A running HW task cannot be stopped or preempted at any time, otherwise it may cause a loss of data consistency. For example, in some algorithm there exists unbreakable execution paths. Therefore, the designers of HW tasks have to provide the points in the code where their execution may stop. Moreover, these points must allow the HW task to be fully resumed from the same point of interruption. These points are denoted as *consistency points*.

To summarize, the designation of HW tasks or accelerator modules must fulfill two requirements : to feature a compatible interface with the unified *PR accelerator interface*, and to consist of a preemptive algorithm with *consistency points* pre-defined in its execution path.

3.2.3 PRR State Machine

A PRR houses HW tasks to implement different devices. Such a region can be allocated to a VM as a virtual device by connecting it to the corresponding IF. The allocations are performed by the *Virtual Device Manager* and the *PPR Monitor*, and can be performed in different ways : direct allocation and preemption. If the required device is not implemented in the allocated PRR, a PCAP transfer will be launched for reconfiguration.

TABLE 3.3 – Contents of the *PRR descriptor* data structure.

Contents	Description
STAT	Current state of PRR
VM_ID	VM that is currently using PRR
DEV_ID	Device that is currently implemented
PRIO	Priority of the client VM
RCFG_DELAY	Reconfiguration overhead

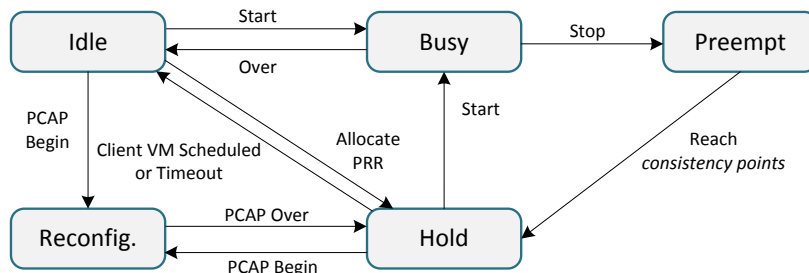


FIGURE 3.9 – The behavior of PRRs as a state machine.

The essential information of a PRR is stored in the *PRR descriptor* data structure, which consists of the contents of TABLE 3.3. Note that, in the DPR technology, the bitstreams size is strictly determined by the size of the reconfigurable area. Considering that the PCAP bandwidth is fixed, the reconfiguration time overhead of HW tasks, i.e. the download time, can be predicted. In our framework, the reconfiguration overheads *RCFG_DELAY* are used to indicate the size of the PRR area in the FPGA.

In our system, PRRs run as a state machine with five states :

- **Idle** : The PRR is idle without any ongoing computation and is ready for allocation.
- **Busy** : The PRR is in the middle of a computation
- **Preempt** : The PRR is running, but the computation will be stopped (preempted) once it reaches a consistency point.
- **Hold** : The PRR is allocated to a VM and is preserved for a certain amount of time
- **Reconfig** : The PRR is in the middle of a PCAP reconfiguration.

The PRRs behaviour can be described according to the flow chart given in Figure 3.9. As depicted, a PRR can only be directly allocated to VMs when it is in *Idle* state and requires no reconfiguration. In other situations, the allocation process requires extra overheads caused by PCAP transfer or preemption. In virtualization, this will cause the VM requests and PRR allocations to be asynchronous. Let us imagine that we allocate PRR #1 to VM #1 via reconfiguration, and the PCAP transfer is performed in parallel

with software. Then, VM #1 gets scheduled out before the PCAP transfer completes and can only start to use the allocated device when it gets scheduled again. In this case, there is a risk for PRR #1 to be re-allocated to another VM before VM #1 gets scheduled, and that VM #1 could never use the requested PRR #1.

To solve this problem, we have introduced *Hold* as a special intermediate state. The PRRs that are allocated to a VM will firstly enter this state. This indicates that the PRR is reserved to a certain VM client. PRRs in the *Hold* state will block any re-assignment and will wait to be used by the VM. PRRs will be released and return to the *Idle* state under two conditions : the target VM is scheduled into the CPU, or the pre-set waiting time *Expire* runs out.

In this case, the time *Expire* parameter determines how long an allocated accelerator should wait before it gets aborted. The value of *Expire* should be configured out of the experience of experiments, and is related to the VMM scheduling policy and the granularity of HW tasks. For instance, in a system where virtual machines are frequently scheduled, the *Expire* parameter should be relatively small, so that the DPR resources are more flexible. In our system, the value of *Expire* is set to be 50 *ms*.

3.2.4 PR Resource Requests and Solutions

Each time that a VM tries to use an unavailable virtual device, an exception will be triggered and then handled by *Virtual Device Manager* as a PR resource request : *Request (vm_id, dev_id, prio)*, which is composed of the VM ID, the virtual device ID and a request priority. The request priority is equal to the priority of the calling VM.

This request is posted to the *PPR Monitor* on the FPGA side to search for an appropriate allocation plan. This plan is referred as a *solution*. A complete *solution* is formatted as :

$$Solution\{vm, dev, Method(prr_id), Reconfig\}, \quad (3.1)$$

which includes the target VM, the required device, the actual allocation method and reconfiguration flag. The different methods include :

- ***Assign*** (*prr_id*) : this solution directly allocates the returned PRR (i.e. *prr_id*) to the request VM. If the requested device *dev_id* is not implemented in this PRR, a *Reconfig* flag will also be added.
- ***Preempt*** (*prr_id*) : this solution means that no PRR can be directly allocated, but the returned PRR (i.e. *prr_id*) can be preempted and re-allocated. If the requested device *dev_id* is not implemented in this PRR, a *Reconfig* flag will also be added.
- ***Unavailable*** : this state means that currently no PRR is available for *Request (vm_id, dev_id, prio)*.

The *PPR Monitor* searches for the best *solution* by checking the *PPR descriptors* (see table 3.3). The searching routine is described in the flow given in FIGURE 3.10. For a given *Request (vm_id, dev_id, prio)*, the *PPR Monitor* first obtains the list of compatible PRRs for the target device (*dev_id*) by checking the HW task index table in

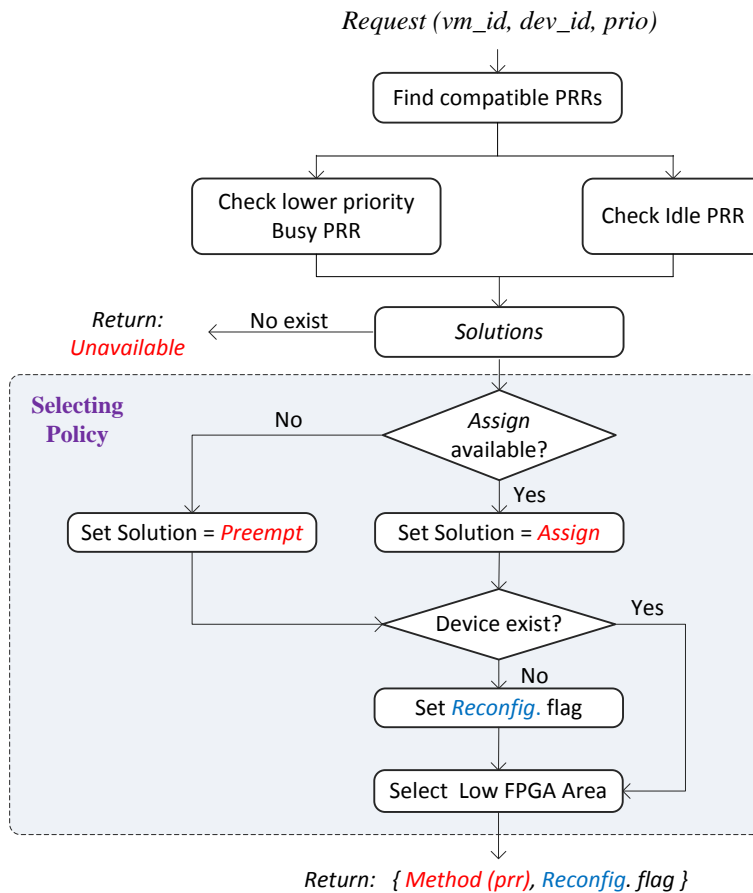


FIGURE 3.10 – The solution searching sequence and the selecting policy for solutions in the *PRR Monitor* logic.

Figure 3.7. The states of these compatible PRRs are then checked for possible solutions. If multiple solutions are found, the best one is chosen according to the selecting policy. This decision-making algorithm is based on the application scenario, and can include various factors. In our algorithm, *Idle* PRRs are considered to be best solutions. Preemptions are chosen only when no *Idle* PRR exists. Besides, the selector always chooses the solution with a minimal reconfiguration overhead since it means a faster device response and a lower power consumption. However, these policies can be easily modified and adapted.

FIGURE 3.11 depicts the interaction between the *PRR Monitor* and the *Virtual Device Manager*. Normally the selected solution is sent to the *Virtual Device Manager* for further handling. However, if there is no valid solution (i.e. *Unavailable*), this unsolved request will be added to the *search list*, which is a waiting list of all unsolved requests. *PRR Monitor* keeps searching solutions for requests in this list, and acknowledges the *Virtual Device Manager* whenever a new solution is found. The searching runs in parallel with VMs, following priority-based FIFO principle, so that when a requests conflict occurs,

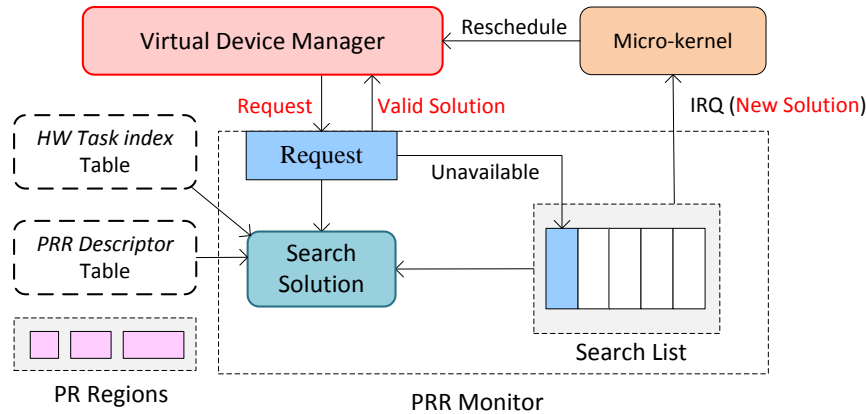


FIGURE 3.11 – The interaction between the PRR monitor and the *Virtual Device Manager* to search for appropriate allocation solutions.

the *PPR Monitor* always chooses the highest priority request.

An important issue about the preemption mechanism of accelerators is that it may influence the scheduability of VM tasks. Since the priority of requests is equal to the VM priority, which makes it possible that a high-priority RTOS task can be blocked by low-priority task. This problem can be fixed by using a more sophisticated scheduling policy of HW tasks, for example, using "sub-priority" to indicate the task priorities in order to avoid the priority inversion/blocking issues of RTOS tasks. However, such a mechanism will significantly increase the complexity of scheduling algorithm of multiple HW tasks on multiple PRR containers. The preemption of HW tasks is unlike software ones, since their stopping and reconfiguration tasks extra overheads (sometimes quite big). All these factors will introduce high complexity in the real-time schedule for VM users, since the task execution in such a system is quite difficult to predict beforehand. Therefore, to keep the behavior of critical tasks predictable, we assume that the FPGA resources are always sufficient for the high-priority VM, whereas they can also be shared and reused by low-priority VMs. This assumption seems reasonable in practice, since critical tasks are pre-determined in most embedded systems.

3.2.5 Virtual Device Manager

The *Virtual Device Manager* is a special service provided by Ker-ONE. As previously described, this service runs in an independent VM and communicates with other VMs through the IPC channel. It has a higher priority than guest OSs and can preempt them once it gets scheduled (see Section 2.5.1). After its execution, it suspends itself and the VMM resumes the interrupted VM immediately.

The *Virtual Device Manager* stores all the HW tasks in its memory and is the only component that can launch PCAP reconfigurations. The main task of this manager is : (1) to communicate with VMs and manage the virtual devices in their space; (2) to

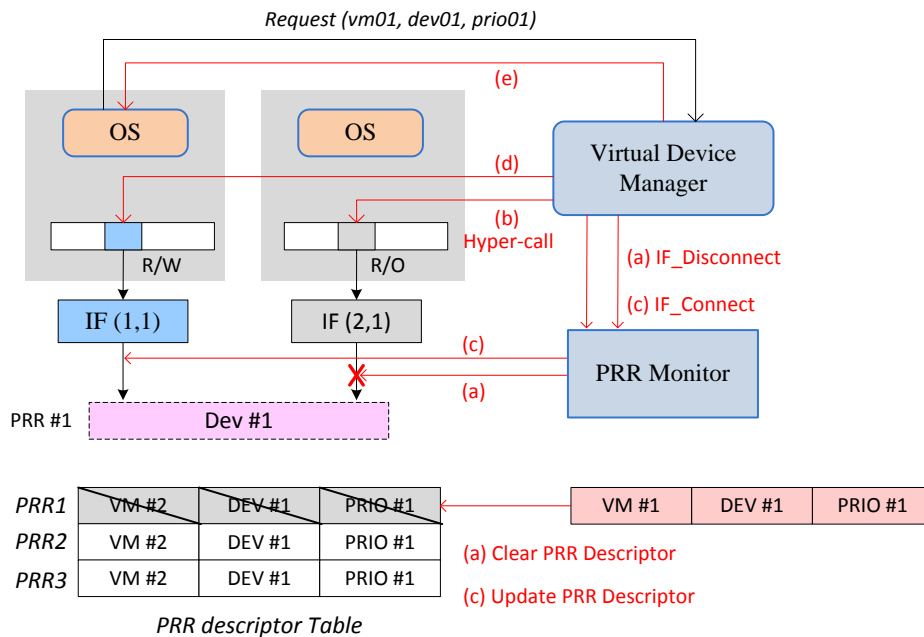


FIGURE 3.12 – Execution flow for solution $\{assign(prr01), non-Reconfig\}$ to directly allocate PRR #1 to VM #1.

correctly allocate PR resources to VMs.

When a virtual device is unavailable, the corresponding IF is not connected to any PRR and its address is set as read-only. Any writing operations on this address will trap to a VMM as a page-fault exception. We assume that, in order to command a virtual device, a VM always needs to configure the device interface in the IF. In this case, any VMs’ attempt to use unavailable virtual devices will be detected by the VMM, and then passed to the *Virtual Device Manager*.

Since the virtual devices are pre-determined and identically mapped in all VM virtual address spaces (see FIGURE 3.5), it is easy to identify the target device by simply checking the page-fault address. Then this exception is translated into the $Request(vm_id, dev_id, prio)$ format and is used to search solutions. The *Virtual Device Manager* allocates PR resource to VMs according to different solutions.

The allocation/ de-allocation of PR resources are realized by manipulating IF connections and VM page tables. In Figure 3.12, we have depicted the complete flow to allocate a PR module to VM as a virtual device. In this example, after a given $Request(vm01, dev01, prio01)\}$, a solution $\{Assign(prr01), non-Reconfig\}$ is performed. We assume that PRR #1 is previously used by VM #2 and currently in *Idle* state and can be directly re-allocated. First, de-allocation of PRR #1 is performed in step (a) and (b). Then, allocation is performed by re-connecting PRR #1 with VM #1 using $IF_Connect$ in (c) and (d). The details of steps are as following :

1. Command $IF_Disconnect$ is given to the *PRR Monitor* to disconnect the IF of

- VM #2. Meanwhile, the corresponding PRR descriptor entry is cleared.
2. A hyper-call is used to set the no-more-available device interface as read-only in VM #2's page table.
 3. Using *IF_Connect* to connect PRR to the IF of VM #2. With this command, *PRR Monitor* also updates the PRR descriptor entry with the new client VM #2.
 4. Another hyper-call is used to change VM #1's *dev01* interface as read-write.
 5. *Virtual Device Manager* suspends itself. The VMM resumes VM #1 to the exception point and VM #1 continues to use this device.

For a guest OS, the ideal solution is *{Assign, non-Reconfig}*, because a PRR can be allocated immediately as shown in the previous example, and the allocation is totally transparent. On other non-immediate solutions which requires reconfiguration or pre-emption, or when there is no valid solution, the target device is not ready or preempted. This influences the execution of VM software and must be acknowledged by releasing IPC messages (see Section 2.3.3), and must be properly handled at the VM side. There are currently three types of messages :

- *IPC_WAIT (dev_id)* : This IPC is released to the VM that generates a request when the PRR cannot be immediately allocated.
- *IPC_READY (dev_id)* : This IPC is released in pair with *IPC_WAIT*, indicating that the earlier unavailable device is now ready for use.
- *IPC_PREEMPT (dev_id)* : This IPC is released when a running PRR is preempted. It informs the former client VM that its virtual device has been stopped.

The first-step handling of solutions is performed by the *run_solution()* function, which is called from the main function of the *Virtual Device Manager*. In Listing 3.1 we demonstrate the pseudo code of this function. This function is called with a *Solution* structure as the argument and returns the allocation results. If the PRR resource is not successfully allocated, a signal *IPC_WAIT (dev_id)* will be sent to the requesting VMs.

Listing 3.1 – Pseudo code of solution handling process

```

1  /* Virtual Device Manager Main function*/
2  int VDManager_Main(){
3      Solution s;
4      ...
5      s = search_solution(VM_ID, Dev_ID, Priority);
6      if(Run_Solution(&s))
7          send_IPC_WAIT(VM_ID, Dev_ID);
8      ...
9  }
10
11 /* Run_Solution(Solution *)
12 *   Argument: Solution{VM_ID, Dev_ID, PR_ID, Method, Reconfig}
13 *   Return: 0 (Success), 1 (Wait)
14 */
15 int Run_Solution(Solution *s){
16     HWTask_Descriptor *HW_Task;
17
18     switch(s->Method){

```



```

19
20 /* Method.unavailable means no appropriate PR for now.
21  * Request is suspended until a solution is found by PR controller. */
22 case unavailable:
23     return 1;
24
25 /* Method.assign means to assign an IDLE PR to IF:
26  * (1) If need reconfiguration: Launch PCAP Transfer and Return 1 (WAIT)
27  * (2) If don't need reconfiguration: Connect PR with current IF */
28 case assign:
29
30     IF_Disconnect(s->PR_id); //Disconnect target PR from previous IF
31
32     if(s->Reconf == true){
33         HW_Task = HWTaskIndexTable[s->PR_id][s->DevID];
34         if(PCAP_Launch(HW_Task->Bitsream_Addr, HW_Task->Bitsream_Size ))
35             print("PCAP Error! \n\r");
36         PRR_STATE_RCFG_SET(s->PR_id);
37         return 1; }
38     else{
39         IF_Connect(s->VM_ID, s->Dev_ID, s->PR_id);
40         ClearSolution(s);
41         PRR_STATE_HOLD_SET(s->PR_id); }
42     break;
43
44 /* Method.preempt means to preempt low-priority non-IDLE PR:
45  * 1) Give STOP command to PR Controller
46  * 2) Return 1 (WAIT) */
47 case preempt:
48     PR_STOP(s->PR_id);
49     return 1;
50
51 case nonvalid:
52 default:
53     panic("ERROR: Undefined Solution Method! \n\r");
54 }
55
56 return 0;
57 }

```

From the process shown in Listing 3.1 we can notice that *Virtual Device Manager* cannot complete non-immediate solutions in one-shot execution, since it needs to wait for the completion of reconfiguration or preemption to make further operations. So it saves the unfinished solutions, and gives the CPU time back to guest OSs by suspending itself. Meanwhile, the *PRR Monitor* keeps tracking these solutions on the FPGA side and delivers interrupts in the following cases :

- ***IRQ_New_Solution*** : A new solution is found in the Search List and is returned to *Virtual Device Manager*.
- ***IRQ_PCAP_Over*** : The PRR reconfiguration is complete and it is ready to use.
- ***IRQ_PRR_Stop*** : A running PRR has been preempted and is ready for re-allocation.

Whenever these IRQs trigger, the VMM reschedules the *Virtual Device Manager* imme-

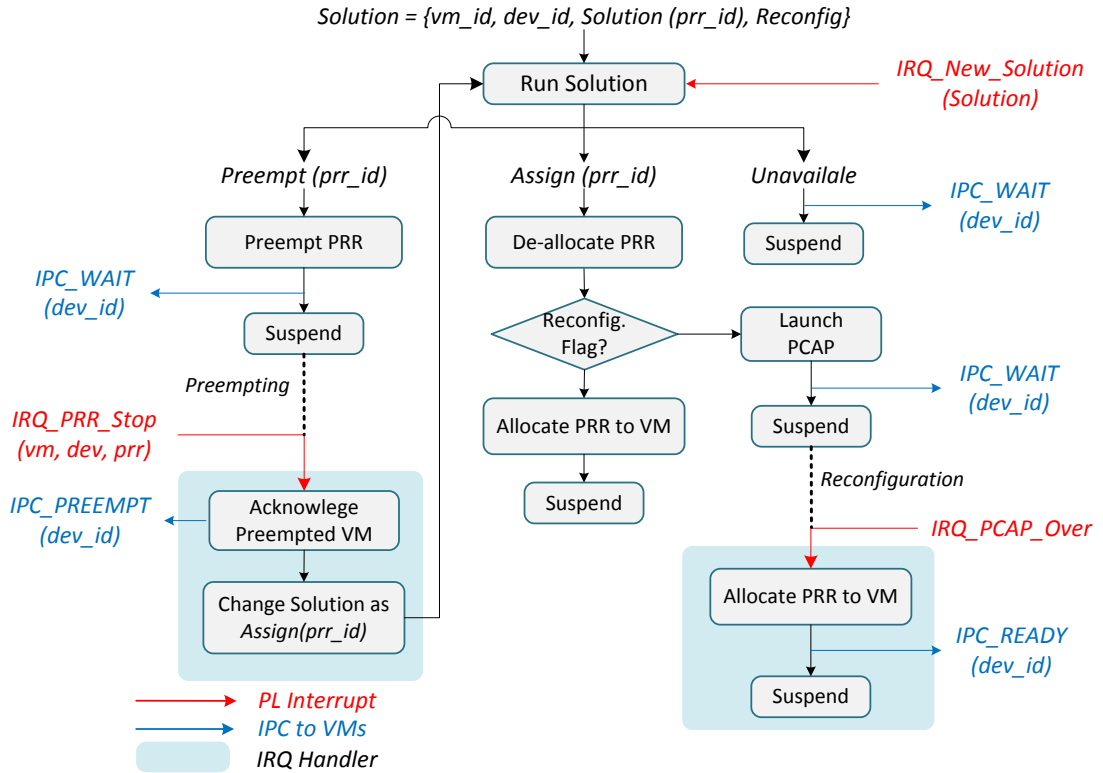


FIGURE 3.13 – The process of Virtual Device Manager handling $Solution \{vm_id, dev_id, Method (pr_id), Reconfig\}$.

diately to handle them.

In FIGURE 3.13 we demonstrate the process of solution handling in the *Virtual Device Manager*. The routine is composed of one main function $Run_Solution()$ and several interrupt handlers. We can notice that preemption and reconfiguration solutions are performed in two stages : (1) the main function $Run_Solution()$ launches the reconfiguration or preemption and then goes to sleep, (2) the *Virtual Device Manager* is awakened by IRQs and completes the solution in the interrupt routine service. Note that, in the processing of the *Preempt* solution, the manager first stopped the target accelerator, and then handles re-allocate this PRR to the target virtual machine by starting a typical *Assign* solution by calling $Run_Solution()$. This is because after preemption, the target PRR can be considered as an idle allocatable component.

3.2.6 Security Mechanisms

The strong isolation among virtual machines is one of the most essential features of virtualization, which guarantees the security of each component. The sharing of DPR accelerators may undermine the system isolation since it increases the attack surface of a virtual machine. A compromised OS may try to use accelerators of another VM, or

try to leverage its accelerators to attack other VM domains. In this part we discuss the potential security threats and propose our solutions.

To protect the isolated environment of a virtual machine, it is mandatory to follow two principles : first, one accelerator can be shared by any VM, but should be exclusively used once it is dispatched to a specific guest OS. Second, accelerators should only access the memory region of the VM which is currently using it. Accessing a memory space outside the specific section is forbidden.

The solution for the first challenge is addressed by the allocation mechanism we described. During the allocation, the *Virtual Device Manager* manipulates the page tables of VMs to guarantee that a VM and a PRR are exclusively connected, so that a VM is only permitted to access the DPR resource that is allocated to it.

Meanwhile, the protection of memory space requires extra mechanisms. In classic virtualization systems, the separate execution environment relies on the MMU, which automatically controls accesses from different privilege levels and blocks illegal access. Isolated memory spaces are ensured by managing the page tables (see Section 2.2.3). However, considering that the Zynq-7000 provides AXI-HP interfaces as bus master on the PL side, the FPGA accelerators can directly access physical CPU memory without going through MMU (see FIGURE 3.1), which means that it is impossible to monitor and control the FPGA access via the page table mechanism. There is a risk that the accelerator accesses other VM domains or even the micro-kernel domain to attack the system.

Therefore, we created a custom unit, denoted as the hardware memory management unit (*hwMMU*), to monitor any access to the CPU memory. *hwMMU* creates a memory region table to store the physical memory regions of each VM. This table is initialized during the start-up stage of system. Whenever a PR accelerator attempts to access the CPU memory via AXI-HP, *hwMMU* checks the target address according to the memory region table, and any access outside the current client VM domain will be rejected. This mechanism guarantees that DPR accelerators are strictly constrained in a determined VM domain, and are isolated from other parts of the system.

In FIGURE 3.14 we demonstrate the secure environment when virtual machines use PR accelerators. Each virtual machine and its PR resources are logically and physically isolated, guaranteeing the system safety.

3.2.7 Initialization Sequence

In our framework, DPR management is an optional feature of the Ker-ONE micro-kernel. Users can choose this option when PR resources are shared among virtual machines. The *Virtual Device Manager* is an independent loadable module which will be launched as a user-level process when the DPR management feature is enabled.

Ker-ONE creates a high-priority virtual machine to host *Virtual Device Manager*. Bitstreams of PR modules are generated beforehand via Xilinx synthesis tools and stored in a dedicated region of system memory. This memory region is exclusively mapped in the memory space of *Virtual Device Manager* during the launching stage. Because of

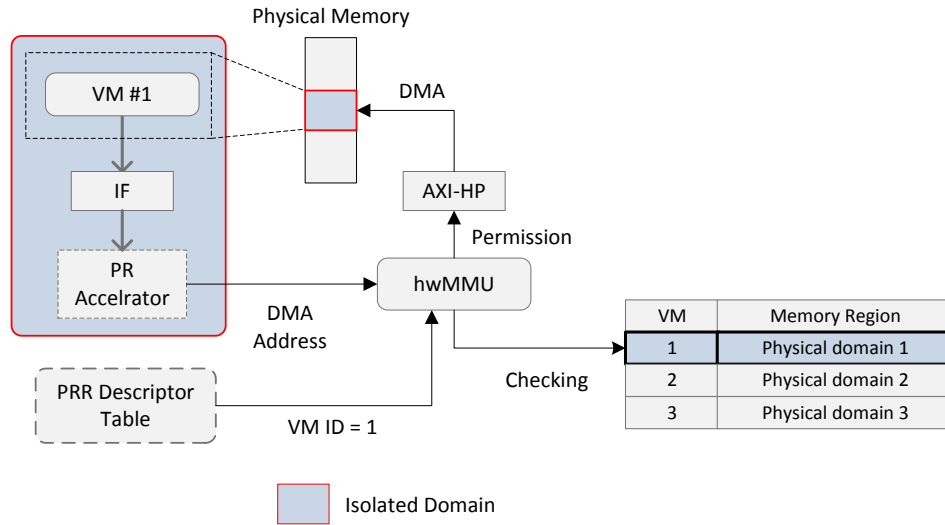


FIGURE 3.14 – The isolated execution environment of virtual machine and its allocated PR accelerator.

higher priority, *Virtual Device Manager* starts initialization before other guest OSs, and suspends itself after the boot-up stage is over. This stage follows the sequence below :

1. Ker-ONE schedules the *Virtual Device Manager* to boot up.
2. Initialization of the PCAP and DevCfg for DMA transfers.
3. Initialization of the HW task index table to load information of bitstream files, i.e. compatibility, IDs, addresses, sizes, configuration overheads. (see FIGURE 3.7)
4. Hyper-calls to create memory mapping to the bitstream memory region so that these files are available.
5. Hyper-calls to create page mapping to the interface of *PRR Monitor* registers.
6. *PRR Monitor* command to initialize PL logic.
7. Initialization of the memory region table in *hwMMU* with physical memory domains of guest OSs.
8. Initialization of interrupts (*IRQ_New_Solution*, *IRQ_PCAP_Over*, *IRQ_PCAP_Stop*) and their handlers.
9. Hyper-call to suspend itself and reschedule. Other virtual machines are rescheduled to execute.

3.3 Application of Virtual Devices

With our framework, coding is significantly simplified for software applications to use virtual devices. From their point of view, the use of virtual devices is performed by a series of write/read operations on the interface registers as ordinary devices. Though

the access of PR resources are transparent for user applications, it still requires extra mechanisms from the OS kernel that deal with special situations such as unavailable or preempted devices. In this section, we discuss the policies followed by guest OSs when using PR resources and give some practical advises for software developers.

3.3.1 Virtual Device Blocking

In user tasks, an exception will be generated on the first writing instruction on unavailable virtual devices. In an ideal situation, the interrupted virtual device would immediately be allocated and the task would continue from the interrupted point seamlessly. However, it is also possible that the virtual device would currently be not ready and that an IPC signal *IPC_WAIT* would be sent back. In this case the usage of virtual device should be suspended until *IPC_READY* is received.

From Section 2.3.3, we know that IPC signals are sent to virtual machines as interrupts. One appropriate policy for guest OS is to handle these interrupts by blocking/unblocking the involved tasks. In simple OSs, this policy can be simply implemented by leveraging the synchronization mechanisms such as semaphores.

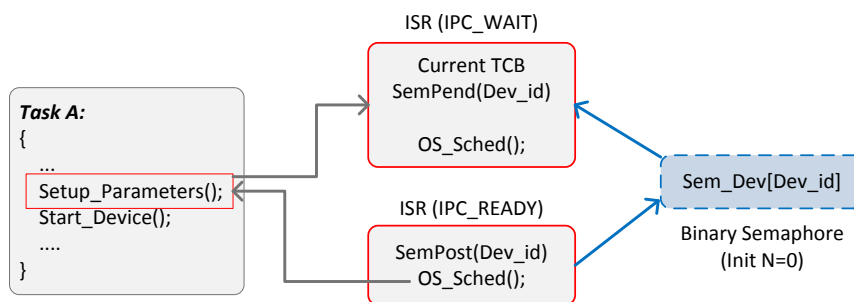


FIGURE 3.15 – An example in which the $\mu C/OS-II$ guest uses a binary semaphore to handle the IPC signals.

In FIGURE 3.15, we demonstrated our approach on a simple $\mu C/OS-II$ as an example solution. Initially, each virtual device is allocated a semaphore with initial value 0. Imagine that an IRQ *IPC_WAIT* is received during the executing of a task, indicating that this task is trying to use an unavailable virtual device. In the ISR process, we use a slightly-modified semaphore-pending function to block the current task by getting the corresponding semaphore from the semaphore list according to the unavailable device ID. This task will be kept blocked until an IRQ *IPC_READY* with the same device ID is received. In this case, the ISR will post the semaphore and resume the earlier blocked task, so that its operation on the virtual device can continue.

3.3.2 Virtual Device Preemption

Another issue caused by the PR resource virtualization is the preemption of virtual devices. Recall that PR accelerators can only be stopped when they reach the *consistency*

points, which depends on the types of computation. For algorithms that perform a single-shot computation, e.g. image filtering and video encoding, these processes cannot be interrupted during their execution. Software tasks using these algorithms will not be influenced. Even though their devices are re-allocated to other VMs, the execution results are guaranteed.

The situation becomes much more complex when the computation of a given accelerator is preemptive, i.e. with valid *consistency points*. The complete computation process may be broken into several stages when it gets preempted in the middle. This algorithm could be continuous, where software begins the computation and sleeps until some event occurs. Normally users expect that this type of computation keeps running in the background and therefore the preemption is unpredictable.

To react to the preemption of these devices, the OS kernel must provide a mechanism to re-launch the stopped device as soon as possible. This policy requires a task that is responsible for detecting the preemption (via *IPC_PREEMPT*) and for restarting it. This dedicated task should resume the computation according to the execution results and the status that is stored in IF registers.

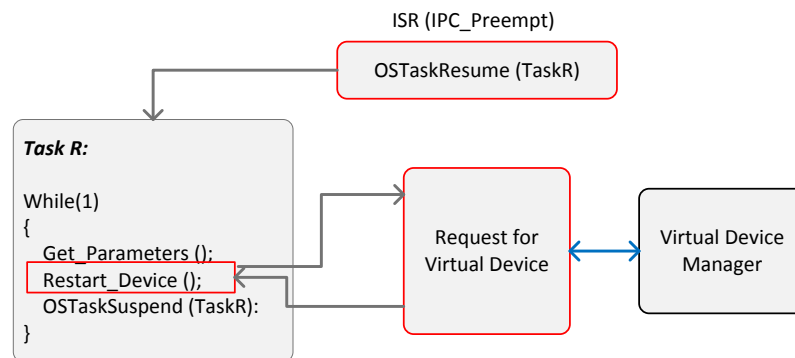


FIGURE 3.16 – An example that Guest $\mu\text{C}/\text{OS-II}$ handles the preemption of virtual devices by calling a dedicated task to re-launch the interrupted computation.

In FIGURE 3.16 we present another example of a $\mu\text{C}/\text{OS-II}$ guest dealing with the preemption of a PR accelerator. A dedicated task is created to restart the interrupted accelerator. When an IRQ *IPC_PREEMPT* arrives, the ISR resumes this task, which, without doubt, will cause an allocation request to the *Virtual Device Manager*. This task will be suspended again after the accelerator is re-allocated and the computation is re-launched.

The policies that we introduced previously imply that a guest OS requires modifications to provide additional features about virtual devices, including IRS and dedicated tasks. Considering that the purpose of our research is to isolate users from the details of low-layer PR implementation as far as possible, we suggest a software structure for guest OS kernels, where the involved modifications are implemented as a patch package. These modification are kept invisible to the user applications, so that software development can be simplified.

3.4 Summary

In this chapter we have introduced a framework which facilitates the DPR resource management for virtual machines on top of the Ker-ONE micro-kernel. Our framework is intended to provide a highly-abstracted, transparent program interface to access re-configurable accelerators.

To this purpose, in each virtual machine, PR accelerators are mapped as universally-addressed peripherals, which can be accessed as ordinary devices. Through dedicated memory management, our framework automatically detects the request for DPR resources and allocates them dynamically according to a limited preemptive allocation policy. Dedicated management components, i.e. *Virtual Device Manager* and *PRR Monitor* are implemented on both software and hardware sides to handle allocations at run-time. We also thoroughly discussed the security issues for virtual machines that are caused by the sharing of PR accelerators. With the combination of page table control and its dedicated component *hwMMU*, the virtual machine domains (including software stack, memory space and allocated PR accelerators), are safely isolated from other parts of the system.

In this chapter we also discussed the programming policies for guest OSs which are willing to use PR resources on Ker-ONE. For special occasions as the blocking and preemption of PR accelerators, we presented detailed explanations and practical examples based on an actual $\mu\text{C}/\text{OS-II}$ RTOS.

In the next chapter, we demonstrate the details of Ker-ONE implementation. Then we will evaluate the performance of our system with extensive experiments and benchmarks. Overheads in both virtualization and PR resource allocation will be analyzed. We will also discuss the real-time schedulability and how it would be influenced by these overheads.

CHAPTER 4

IMPLEMENTATION AND EVALUATION

The purpose of our virtualization framework is to provide a lightweight approach for small-scaled embedded systems. The methodology of implementation determines the size of the kernel, and the complexity of development. On the other hand, we also pay attention to the system performances for virtualization and DPR resources management. To perform an extensive evaluation of our system, it is necessary to measure the overheads caused by the proposed mechanisms and analyze them with thorough discussion. Meanwhile, it is also essential to discuss the real-time schedulability of virtual machines in the context of our mechanisms. In this chapter, we first present the implementation principles and details of the proposed approach. Then we present the results of both custom and standard benchmarks on the Ker-ONE micro-kernel to demonstrate the virtualization efficiency. We also measure the overheads of DPR resource allocation to evaluate the management mechanism. At the end of this chapter, we discuss the real-time schedulability based on the acquired experiment results.

4.1 Implementation

In this section, we present the implementation of the Ker-ONE micro-kernel on the ARMv7 architecture based on a Zynq-7000 platform, which features a dual-core Cortex-A9 processor. In this thesis, to simplify the architecture, we consider a single core, leaving the multi-core studies to future research. Detailed hardware information is shown in TABLE 4.1.

In the software domain, the system is composed of three objects : the Ker-ONE micro-kernel, the user environment, and guest OS/applications. Both the micro-kernel and the user environment are provided from the Ker-ONE project and built independently. They cooperate with each other to establish the virtualization environment. Guest OSs/applications are developed and built by users. The image of guest OSs is then used by Ker-ONE to create virtual machines. To properly build a para-virtualized guest OS, the source code of OS kernels must be modified with the para-virtualization patch codes provided by the Ker-ONE project. This process takes extra efforts for different OSs, as we have to manually analyze and modify the OS kernels code according to their architec-

TABLE 4.1 – Development platform information for the proposed approach.

Feature	Description
Processor	ARM Cortex-A9
CPU Frequency	660MHz
Cache hierarchy	32KB L1 ICache, 32KB L1 DCache, 512KB L2 unified cache
RAM	512 MB
External Memory	4 GB SD Card
Board	Xilinx ZedBoard
Hosted RTOS	μ C/OS-II

tural characteristics. This is also one of the reasons why we chose simple guest OSs, since complex OSs such as Linux require tremendous efforts. As can be noticed in TABLE 4.1, μ C/OS-II is used for researches in this thesis.

In case where DPR technology is used, on the FPGA side, the fabric is divided into static circuits and reconfigurable partitions (RP) beforehand. Reconfigurable accelerators for these partitions are synthesized as bitstream files, whose information is stored in the user environment and will be used by the *Virtual Device Manager*.

4.1.1 RTOS Para-virtualization

To present the validity of real-time virtualization, we para-virtualized a well-known RTOS, μ C/OS-II, which is commonly used in education and research. The virtualized guest OS is named Mini- μ C/OS.

In Section 2.2.1.2, we have analyzed the sensitive instructions that should be processed for para-virtualization. They need to be replaced by hyper-calls or assembly macros to re-direct these operations to VMM or virtual resources. With this target in mind, we have scanned the kernel source code of μ C/OS-II to locate these instructions, which are mostly involved in the following functionality :

- PSR operations are performed to manipulate the processor status. For example, an interrupt mask is often modified when the OS kernel performs critical execution, which is required for most kernel functions.
- Exception handling routine deals with sensitive resources, i.e. PSR register to save/resume task context and the GIC to handle interrupts (see Section 2.3.1).
- Timer operations are frequently performed to handle timer ticks.

Besides of these operations, some less-frequent instructions involve Cache/TLB, page tables, and peripherals such as SD cards. These instructions also require modifications. To simplify the modification, we have created code patches that include para-virtualization API interfaces for hyper-calls and alternative functions (i.e. assembly macros) to replace

the original operations. Changes on OS kernel source code are summarized in the following :

- A shared memory region is created to hold virtualized resources such as PSR register and vGIC registers (see Section 2.4).
- Sensitive instructions are replaced by APIs. Some of them are re-directed to the shared memory region. Others are emulated by VMM through hyper-calls.
- Timer operation remains original, as a guest OS is permitted to access it directly.

In the design of Ker-ONE, we attempted to avoid unnecessary modifications of guest OSs. The amount of Lines of Code (LoC) that were required for modifying the original $\mu\text{C}/\text{OS}$ is very low, counted as 139 LoC for modified source code and 223 LoC for additional virtualization patch. Most changes take place in assembly code where sensitive instructions are used.

According to different usages, the Mini- $\mu\text{C}/\text{OS}$ is released with two distributions, aiming for guest RTOS and general-purpose OS respectively. The two distributions are built separately in a conditional compilation way, with only slight differences, including the timer and the idle task hook routines.

To evaluate our workload, we compare our Mini- $\mu\text{C}/\text{OS}$ with xeno- $\mu\text{C}/\text{OS}$, which is a para-virtualized $\mu\text{C}/\text{OS-II}$ on Xen-ARM and is available from the XEN Wiki site [uco12]. In TABLE 4.2 we give a qualitative comparison for the required modifications. XEN-ARM has a higher impact on $\mu\text{C}/\text{OS-II}$ mainly because it imposes the *event channel* mechanism into the OS kernel, which is used to manage virtual IRQs and virtual timers. Therefore, the original exception/interrupt handling routine in $\mu\text{C}/\text{OS-II}$ is totally aborted. In the contrast, Ker-ONE allows $\mu\text{C}/\text{OS-II}$ to use its original handlers with only slight changes (see Section 2.3.1).

TABLE 4.2 – Qualitative comparison of $\mu\text{C}/\text{OS-II}$ source code modification in different characteristics of Ker-ONE and XEN-ARM kernels.

Modification	Ker-ONE	XEN-ARM
Scheduler	Light	Light
Interrupts & exceptions	Light	Heavy
Timer	Light	Heavy
Initialization	Medium	Heavy
Sensitive instruction rewriting	Medium	Medium

4.1.2 Complexity

In order to evaluate an hypervisor complexity, we generally assume that virtualization layers are divided into a host-level micro-kernel (or hypervisor in some cases) and additional user-level functionality. The micro-kernel/hypervisor consists of the most critical components, whose size determines the TCB of system, and should be kept small to

reduce the potential threats or attacks on the code.

The design of Ker-ONE results in a lightweight implementation with a small TCB size. The Ker-ONE micro-kernel is built up with less than 3,000 LoC. This is due to our attempts to eliminate all virtualization features that are unnecessary in a small embedded system. For example, we avoid complex memory virtualization mechanisms and focus on simple OSs with single address space. We also realize real-time scheduling with minimal complexity. Moreover, I/O devices that are not critical to system security (i.e. UART and SD card) are allowed to be directly accessed to avoid heavyweight emulations. Via these methods we have obtained a minimal micro-kernel for small embedded systems.

It is difficult to compare our work with existing approaches, since, to the best of our knowledge, there is currently no available information for systems sharing the same features, i.e. ARM para-virtualization on small systems with single-protection-domain OSs. However, in order to give a global overview of the kernel complexity, in TABLE 4.3, we compare the TCB size for several traditional ARM para-virtualization approaches on the ARMv6/7 architecture, including OKL4 [HL10] and Xen-ARM [XEN14].

TABLE 4.3 – Comparison of TCB size for ARMv6/v7 virtualization environments measured in LoC.

Hypervisor	Kernel	User-level	Total
Ker-ONE	2,142	712	2,854
OKL4	9,800	/	9,800
Xen-ARM	4,487	Dom0	4,487 + Dom0 (Linux)

From this comparison we can notice different design policies. Xen-ARM leverages the functionality of Linux to perform I/O emulation, the virtualization functionality provided by Xen-ARM is quite complete. Following the similar principle, OKL4 also chooses to provide extensive virtualization features to support Linux. It is inevitable that these approaches ends up with higher complexity. The trade-off of Ker-ONE is to give up complex OS, and to target simple OSs in embedded systems. As a result, this manages to greatly reduce the software complexity, while still maintaining virtualization features and real-time support.

4.1.3 System Mapping

Ker-ONE creates a system mapping to organize on-chip resources and software components. Resources mapped to the address space include CPU memory, system registers, peripherals and FPGA fabric. By default they are implemented into different regions in the physical address space, as shown in FIGURE 4.1. Through the address mapping created by Ker-ONE, these resources are re-organized in the virtual machine’s memory space. The scheme of system mapping is demonstrated in FIGURE 4.1. To be specific, it is composed of several regions :

- System registers and peripherals are mainly placed in a high address space, e.g. I/O peripheral registers are located in address section (0xE0000000 - 0xE0300000).

These regions are flat-mapped into the virtual machine space.

- The Ker-ONE micro-kernel is also mapped in the high address space, (0xF0000000 - 0xF0100000), with a footprint of 1 MB. This region is universally mapped into each virtual machine space.
- The FPGA resources section, which is originally in the 0x40000000 - 0xC0000000 address space, are re-mapped to higher addresses (0xE0300000 - 0xE0400000) as independent 4 KB pages (see Section 3.2.1). These are the intermediate interfaces (IF) that are considered as virtual devices in the virtual machines.
- The address space under 0xE0000000 is reserved for virtual machines, and can be freely used by guest OS software.

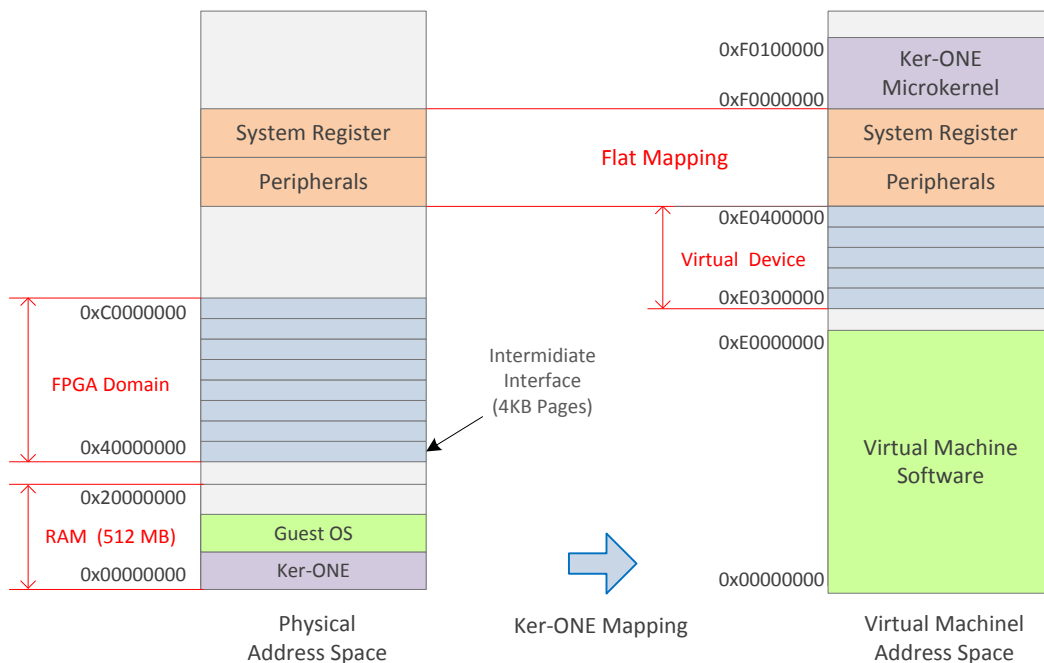


FIGURE 4.1 – Ker-ONE Memory mapping from physical to virtual machine address space.

Note that, in our system mapping, Ker-ONE is included in the address space of all virtual machines, due to its small computing size. Therefore, the switch between VMM and virtual machine is spared from the change of page tables, which facilitates the communication between them. We should also note that the address section reserved for virtual devices is 1 MB, which can hold up to 256 different virtual device interfaces. This section is sufficient in most cases.

4.1.4 Ker-ONE Initialization Sequence

Ker-ONE uses a SD card as boot-up device, which holds essential initialization files, i.e. ELF-format executable images for kernel (*KERNEL.ELF*) and user environment *USER.ELF*, guest OS images and DPR accelerator bitstream files. The boot-up sequence is composed of two stages, respectively being performed in the kernel space and user level.

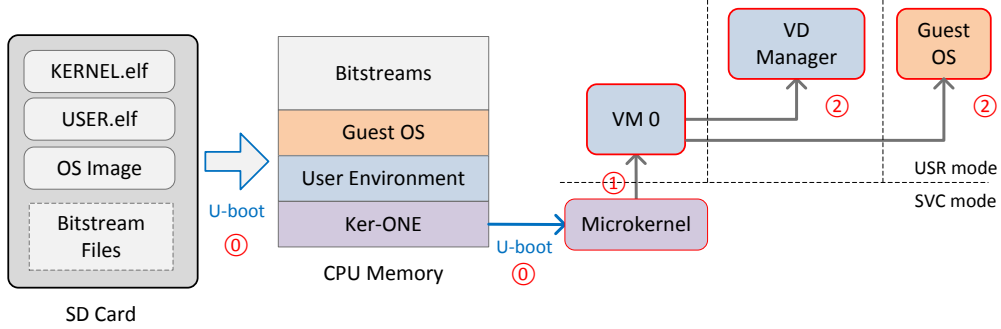


FIGURE 4.2 – Ker-ONE initialization sequence divided into stages. Stage 0 : U-boot load files from SD card and start micro-kernel. Stage 1 : micro-kernel initialize system and creates VM0. Stage 2 : User environment boot up guest OSs and services into independent virtual machines.

FIGURE 4.2 describes the initialization sequence. The first boot-up stage can be launched by any Linux-compatible bootloader such as U-boot [LZQ07], which is responsible for loading initialization files into physical memory and launch the kernel image. This image contains the micro-kernel program executing in supervisor mode. It is responsible for basic system configurations, including mapping and peripheral initialization. At the end of this stage, it creates the first virtual machine domain (denoted as VM0) to start up the user environment (*USER.ELF*) in the user mode. In VM0, system services are instantiated to build up the user level execution environment. For example, *Virtual Device Manager* is launched into another independent virtual machine. To be specific, the guest OS bootloader is called to load the guest OS images, to parse their headers and to create virtual machines with priority levels. Eventually, the initialization sequence ends up by calling the VMM to reschedule virtual machines. Note that, in VM0, software manipulate virtual machines through hyper-call interfaces.

4.2 Ker-ONE Virtualization Performance Evaluations

In this section, we present Ker-ONE experimental results focusing on the virtualization overheads. The measurements have been obtained in three experiments. First, we measured the overheads of fundamental virtualization functions, such as interrupt emulation, VMM scheduling, hyper-calls, etc. Then we evaluated the impact of virtualization on the RTOS performances by measuring extra overheads caused to the VM scheduling.

We also used a standard RTOS benchmark to obtain a global evaluation. Finally, we used our platform to implement real-world applications based on standard benchmarks.

The methodology of evaluation consists in hosting multiple guest OSs (i.e. Mini- μ C/OS) together, on top of Ker-ONE, and executing benchmarks on top of them. We also created custom benchmarks to estimate overheads of various virtualization functions with long-term running and enormous number of samples. We also selected several third-party benchmarks, such as Thread-Metric [Exp07] and MiBench [GRE⁺01]. These benchmarks run as tasks inside guest Mini- μ C/OS. The overheads are measured by setting the Global Timer in the processor.

For all experiments, the VMM scheduling tick interval was fixed to 33 *ms*, and guest OSs were running according to 1 *ms* timer tick. Note that these values are common timing configurations for μ C/OS-II [YY14]. Guest OSs were hosted as general-purpose OSs or real-time OS, according to different experiment purposes.

In this section we also analyzed the schedulability of a guest RTOS, based on the overheads estimated in experiments. Following the equations given in Section 2.5.3, we calculated the impact of virtualization on RTOS task timings and present practical advises.

4.2.1 Basic Virtualization Function Overheads

We have examined various micro-architecture virtualization overheads for the most frequently used and basic VMM functions : hyper-calls, virtual interrupts, virtual machine switch and Vector Floating-Point (VFP) switch. These overheads help us evaluating the VMM performance, and understanding the bottleneck execution path in the micro-kernel.

In this experiment, the system has been configured to concurrently host four Mini- μ C/OS at the same priority level, as general-purpose OSs. All OSs were scheduled according to the round-robin strategy. We created experiment tasks on top of the guest OSs to manually launch hyper-calls and utilize the Vector Floating-point co-processor. The overhead of corresponding VMM functions has then been recorded by a background monitor for a large amount of samples during several hours of execution. FIGURE 4.3 presents the results of the experiments, where minimal, average and maximum overheads are presented in microseconds.

Hyper-call entry/exit measures the overhead latency that is necessary for a virtual machine to generate hyper-calls to the VMM and to return back to a virtual machine immediately. This latency can be used to estimate the extra latency for virtual machines when hyper-calls are launched. Note that hyper-calls are generated frequently by the OS functions, but rarely during user tasks, which mostly execute without sensitive operations. Since Ker-ONE is mapped to the VM address space, no VM switching is required. Hyper-calls entries and exits are relatively low cost processes. They only involve the saving/restoring of the CPU context and the distribution of hyper-calls.

The virtual IRQ emulation experiment represents the cost of emulating a virtual interrupt for a VM. This type of functionality is critical for event-driven OSs, since this latency directly influences the response time of events. Moreover, it is also closely related

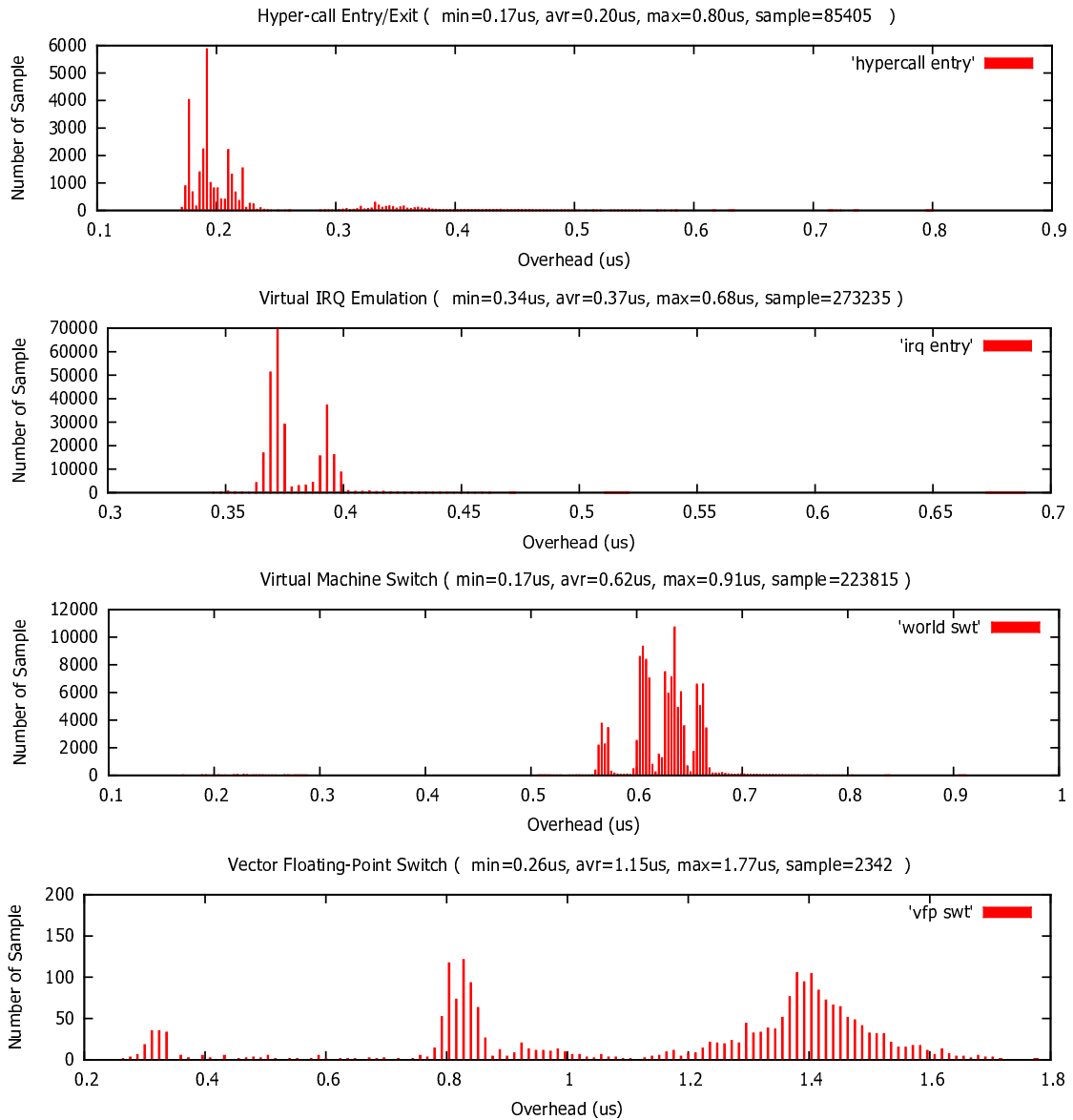


FIGURE 4.3 – Results of basic virtualization function overheads evaluation in microseconds (μs) with minimum, average and maximum values.

to the guest OS' scheduling overhead because a guest OS relies on the virtual timer tick for counting virtual time. This overhead is measured from the physical event's arrival time until the time at which the virtual machine is forced to its local exception vector. This process involves the handling of physical IRQ and the emulation of the virtual GIC interface registers.

Virtual machines switch represents the cost of switching from one virtual machine to

another, which may be relatively heavy. The overhead of virtual machine switch is one of the key metric in most virtualization approaches, as it is usually quite cumbersome, and has a huge impact on VMM efficiency. In Ker-ONE, this switch is performed when a virtual machine consumes its time quantum and moves to its successor, or when it suspends itself and the VMM resumes another virtual machine. This switch includes several major procedures : (1) re-scheduling ; (2) vGIC context switch ; (3) timer state update ; (4) address space (page table) switch ; and (5) CP15 registers update. Note that the address space changing causes a higher TLB/Cache miss rate and inevitably increases the switch latency. Note that, only the round-robin scheduling is presented, whereas the cost for preemptive scheduling is measured in a later section.

We also measured the Vector Floating-point (VFP) switch overhead. As described in Section 2.2.2, the virtualization of VFP co-processor follows the lazy switching policy, which occurs much less frequently than other functions. The switch of VFP includes the saving/resuming of co-processor registers, and the amount of involved registers determines the switch latency. Therefore, we can notice that the experiment results are relatively spread in a wide range, which is due to the fact that the VFP registers are in different states when they switch. We should also note that the VFP switch may cause a relatively high overhead (mostly over $1 \mu s$). In this case, the usage of lazy switch helps reducing the overhead considerably.

Generally, the VMM relies on these basic functions to emulate resources and manage virtual machines. Therefore they represent the efficiency of virtualization. Attributed to the low complexity and simplified mechanisms of a VMM, these functions result in low overheads. As shown in the results, frequently-called functions, i.e. hyper-calls and vIRQ emulation can be handled in less than $1 \mu s$. Besides, as one of the most expensive processes in most virtualization technologies, the virtual machine switch overhead could be limited within $1 \mu s$.

4.2.2 RTOS Virtualization Evaluation

In this part, we will study the influence of virtualization on the performance of guest RTOSs in details, including the OS kernel functions and real-time scheduling overheads. To evaluate the virtualization impact, we led two experiments : first, on a native RTOS on ARM, and second, on a guest RTOS on top of Ker-ONE. According to this study, the virtualization efficiency may be demonstrated. In these experiments, we have implemented Mini- $\mu C/OS$ using the RTOS distribution in one virtual machine and three other virtual machines hosting Mini- $\mu C/OS$ as GPOSs (i.e. using the GPOS distribution). The target benchmarks are executed on top of the RTOS for evaluation. In each test, the comparison of native machine and virtual machine is shown and analyzed.

4.2.2.1 RTOS Benchmarks

The RTOS performance consists of the speed at which a specific RTOS completes its tasks. In this section, we evaluate the performance of the virtual Mini- $\mu C/OS$ with

the Thread-Metric benchmark suite that is dedicated to RTOS performance measurement [Exp07]. Thread-Metric was proposed by Express Logic in 2007 and has been applied in several researches to measure and compare the performance of various RTOSs [MA09][Bes13]. In our experiment one RTOS and three GPOSs (all Mini- $\mu\text{C}/\text{OS}$) were concurrently running on Ker-ONE. The benchmark set was executed on the RTOS. To obtain the performance loss due to the virtualization implementation, the benchmarks results on the native $\mu\text{C}/\text{OS-II}$ have been used as a reference.

Note that in the designation of Ker-ONE, the resource virtualization mechanism is optimized via a shared memory region (see Section 2.4), which is supposed to bring significant performance improvement. This factor is estimated in our experiment by measuring the non-optimized micro-kernel for comparison.

In order to give an extensive evaluation, XEN-ARM hypervisor was also used to perform a comparison with our system. In the experiment, a Version-3.0 XEN-ARM hypervisor [xen12] was ported to our platform (Zynq-7000). A para-virtualized $\mu\text{C}/\text{OS-II}$, released on XEN site as xeno- $\mu\text{C}/\text{OS}$ [uco12], was also ported. On this $\mu\text{C}/\text{OS-II}$ kernel, we executed the Thread-Metric benchmark and compared the results. Note that $\mu\text{C}/\text{OS-II}$ runs on a single protection domain and requires no multiple page tables. Therefore, although Ker-ONE and XEN-ARM have different memory virtualization strategies, the virtualization contexts of $\mu\text{C}/\text{OS-II}$ were identical, because XEN's support of user-level multiple protection-domains was not used and thus made no influence on the overall performance.

The principle of Thread-Metric is to evaluate the OS kernel performance. The test objects consist of a set of common OS kernel services that are representative and that can be usually found in most RTOSs. These services include context switching, interrupts handling, message passing, memory management, etc. For each OS service under test, the methodology is as follows : first, service function and its inverse function execute in pairs, e.g. allocating/ de-allocating memory, or sending/receiving messages. Second, the testbench keeps executing repeatedly and the iterations number is counted. Third, the iterations number is recorded every 30 seconds and denoted as test score. A higher test score represents a shorter overhead from the corresponding OS kernel function, and thus a better performance. The testbenches included in Thread-Metric are :

- *Calibration Test* : A basic single-task rolling counter function to set up a performance baseline for comparisons.
- *Preemptive Context Switching* : Five tasks of different priorities are created. Starting from the lowest priority task, each one resumes the next higher priority task and suspends itself. The sequence of OS scheduling (i.e. OSTaskSuspend - OSTaskResume - OSSched in $\mu\text{C}/\text{OS-II}$) is evaluated.
- *Message Processing* : One task is created to repeatedly send and receive message through the OS message queue (i.e. OSMessagePost-OSMessagePend).
- *Memory Allocation* : One task that allocates and releases memory through the OS memory block (i.e. OSMemGet-OSMemPut).

- *Synchronization Processing* : One task that pends and posts semaphore (i.e. OSSemPost-OSSemPend).
- *Interrupt Handling* : One task is created to generate software IRQ. Semaphore mechanism is used in the IRQ handler routine to guarantee the handling completion.
- *Interrupt Preemption* : Two task of different priorities are created. Lower priority task generates software IRQ and during the IRQ handler routine the other task is resumed and preempts the low priority one.

To rationally evaluate the virtual machine performance compared to the native machine, we introduced the *Performance Ratio* (R_P) parameter, which can be obtained as :

$$R_P = \frac{Score_{VM}}{Score_{Native}} \times 100\%, \quad (4.1)$$

where $Score_{VM}$ and $Score_{Native}$ stands for the result score of benchmarks on guest OS and native OS respectively. R_P is used to measure the percentage of VM performance compared to the native execution. A higher R_P means a better virtualization efficiency. TABLE 4.4 presents the experimental results of the Thread-Metric benchmarks running on both Ker-ONE and native environments.

TABLE 4.4 – Thread-Metric benchmarks results on both native and virtual $\mu C/OS-II$.

Test Object	Native	VM	Performance
	$\mu C/OS-II$	$\mu C/OS-II$	Ratio (%)
<i>Calibration Test</i>	764458	733879	96.1
<i>Preemptive Context Switching</i>	32113328	28927171	90.1
<i>Message Processing</i>	18431136	16748720	90.9
<i>Memory Allocation</i>	104601611	85091278	81.3
<i>Synchronization Processing</i>	108589466	90893213	83.7
<i>Interrupt Handling</i>	32541832	25768399	79.2
<i>Interrupt Preemption</i>	19089282	16425610	86.0

As the results indicate, $\mu C/OS-II$ in a virtual machine suffers from performances degradation. This is inevitable due to the fact that the OS kernel functions in these benchmarks always execute sensitive instructions and operations to access privileged system resources. For example, at every task context switch, software needs to load and restore the value of CPSR/SPSR register. Moreover, at each time entering OS kernel, it has to be switched to critical mode by masking all interrupts. This involves also the operation on CPSR registers. While these operations are performed with single-line instructions originally, in para-virtualized OS kernel, these instructions are replaced with assembly macros which results in longer execution paths. While these functions are continuously repeated during the tests, it causes noticeable extra overheads compared to the original code.

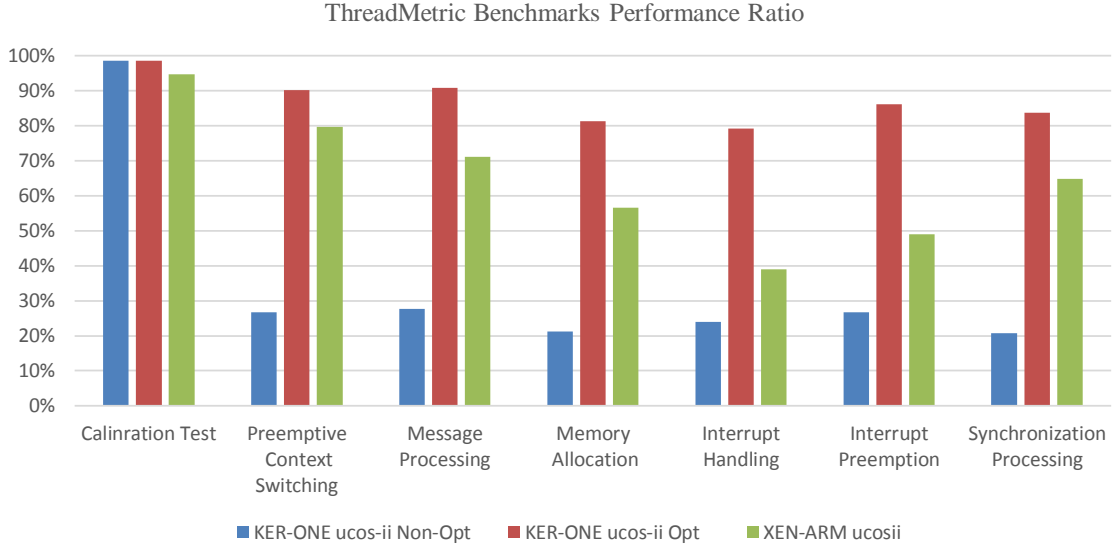


FIGURE 4.4 – Comparison of Thread-Metric *Performance Ratio* (R_P) for para-virtualized $\mu\text{C}/\text{OS-II}$ on Ker-ONE and XEN-ARM.

Another factor that degrades the performance is the virtualization of timers and interrupts. Emulation of these events may be expensive if guest OS executes with high clocking rate or frequent interrupts. This effect can be noticed in the *Interrupt Handling* and *Interrupt Preemption* benchmarks in TABLE 4.4. In these benchmarks massive interrupts are generated and handled, which result in a considerable performance degradation. This explains the relatively low performance in these tests.

TABLE 4.4 also indicates that Ker-ONE hosts RTOS with relatively high efficiency. For kernel services tested in benchmarks, most performance losses are limited to less than 20%. And for some functions such as task scheduling and message processing, their performance are close to the native results, with only 10% performance loss.

In FIGURE 4.4 we compared the benchmark results of three implementations on the same platform : the optimized Ker-ONE kernel, the non-optimized Ker-ONE kernel and the XEN-ARM hypervisor. As we can notice, with optimization, the virtualization performance is considerably improved. As we described in Section 2.4, through optimization, hyper-calls are avoided in common OS kernel functions. For example, in non-optimized solutions, a common task context switch (i.e. OSCtxSw) launches 2 or 3 hyper-calls to manipulate virtual PSR values. For the *Preemptive Context Switching*, this function is repeated continuously, making significant extra overheads. FIGURE 4.4 demonstrates that the optimized mechanism is able to reduce overheads efficiently.

On the other hand, compared to XEN-ARM, using Ker-ONE, the guest $\mu\text{C}/\text{OS-II}$ achieves better performance and is closer to the native machine. Both Ker-ONE and XEN employ the mechanism of shared memory region. The different performances can be explained by the fact that Ker-ONE provides a simpler virtualization interface. For

example, virtualized resources, such as vCPU, vIRQ, are implemented with smaller structures and smaller size. $\mu\text{C}/\text{OS-II}$ can access them with several lines of assembly codes. We can notice that in benchmarks dealing with *Interrupt Handling* and *Interrupt Preemption*, XEN-ARM has more significant performance loss. This is attributed to the virtual interrupt mechanism. As we have introduced, virtual interrupts in XEN-ARM are handled as *event_channel*, which separates physical IRQs from VM event ports. IRQs are remapped to VMs in *event_channels*. This mechanism is useful to provide better VM isolation but also is relatively complex. In contrast, Ker-ONE focuses on the GIC emulation and applies simpler virtual IRQ management. Physical interrupts are forwarded to VMs via a simple monitor-and-redirection mechanism. Additionally, VMs continue to use their original IRQ handlers, which makes this process even simpler. Therefore for interrupt benchmarks, Ker-ONE has a better performance.

4.2.2.2 RTOS Virtualization Overheads

With the Thread-Metric benchmarks, we have performed some qualitative analysis about the performance loss on OS functions. Furthermore, we have created custom benchmarks to accurately measure the RTOS overheads for context switching and scheduling, so that we can precisely discuss the schedulability of the RTOS as explained in [Ste01].

In this evaluation, we have paid attention to the worst-case RTOS task response time, which happens when an RTOS is waken up from the *suspend queue* when events occur, according to the scheduling mechanism of Ker-ONE. In Eq.(2.8) we have analyzed this response time $Response^{VM}$ as composed by : delays caused by VMM critical execution ($\Delta VMMcritical$), by VMM scheduling ($\Delta VMMsched$) and by RTOS task release ($relEv^{VM}$). These three overheads influence the release delay of RTOS tasks, and thereby are important factors for schedulability [APN13]. In our experiment, these overheads have been measured respectively and recorded during hours of execution. A total number of 1,048,576 samples have been obtained. The results of evaluation are shown in FIGURE 4.5.

The *Critical Execution* (i.e. $\Delta VMMcritical$) measures the overhead of VMM critical executions when IRQs are masked. Any events occurring in this period will be delayed until the critical execution is over. When VMs run, the VMM performs critical executions for various reasons, i.e. hyper-calls, IRQs, exceptions or VM switches. In order to cover all possible critical execution overheads, we have executed dedicated test software which helped triggering hyper-calls, IRQs and exceptions. This test ran for several hours and a large amount of samples was obtained. As shown in FIGURE 4.5, the worst-case VMM critical execution is estimated to be $1.47 \mu s$.

The *RTOS Preemption* (i.e. $\Delta VMMsched$) refers to the cost of an RTOS preempting the current GPOS. This process is performed by the VMM and includes several steps : (1) real-time event handling (timer tick interrupts in this case), (2) rescheduling, (3) VM switch, (4) forwarding the timer interrupts to RTOS. As shown in FIGURE 4.5, in most cases, RTOS preemption is completed in the average delay of $0.98 \mu s$, whereas the WCET result is $1.19 \mu s$.

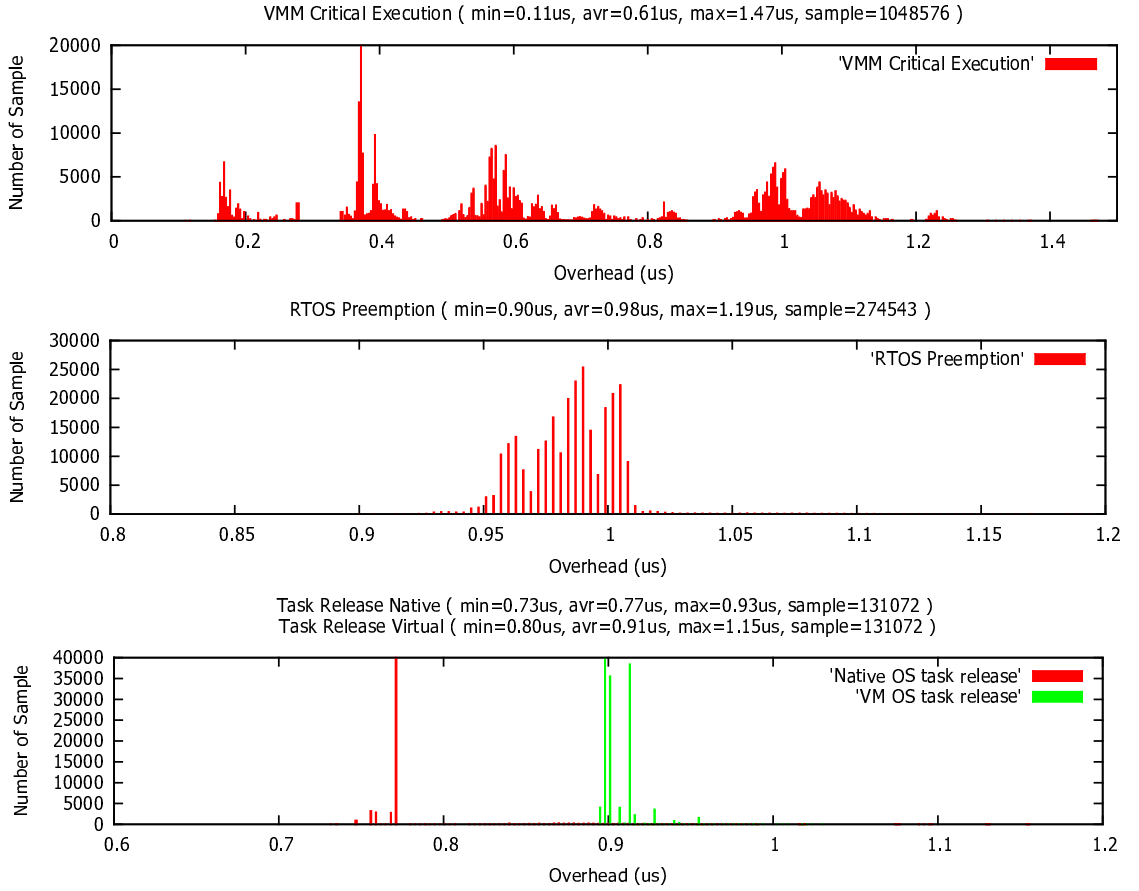


FIGURE 4.5 – Results of VM RTOS task response overheads evaluation in microseconds (μs) with minimum, average and maximum values.

In addition, the *Task Release* latency measures the cost of an RTOS handling the virtual timer tick and scheduling the new ready task. The latency of both native and VM Task Release has been measured, respectively as $relEv^{Native}$ and $relEv^{VM}$. The performance loss of a task Release is mainly caused by the emulation of sensitive instructions that are used in interrupts handling and context switching. According to the former equation Eq.(2.6) and Eq.(2.7) in Section 2.5.3, the worst-case extra *Release Event* overhead can be estimated as :

$$\Delta_{VM}^{relEv} = relEv_{(WCET)}^{VM} - relEv_{(BCET)}^{Native}. \quad (4.2)$$

In this equation, Δ_{VM}^{relEv} is estimated to be $0.42 \mu s$. Therefore, according to Eq.(2.9), the total influence that virtualization causes on the RTOS response time $\Delta_{VM}^{Response}$ can be estimated as $3.08 \mu s$.

4.2.3 Application Specific Evaluation

To measure the virtualization overhead of our system in more realistic use cases, we have run a series of application-related benchmarks. Since our system targets small-scaled embedded systems, which are mostly used for portable devices, vehicles and mobiles, it is reasonable that we verify our system with applications from tele-communication and industrial domains. Therefore, the widely-used embedded benchmark MiBench suite has been selected [GRE⁺01]. MiBench has been used by many embedded virtualization technologies as standard benchmarks [XBG⁺10]. This test suite provides representative applications of embedded systems in several categories. Two groups of applications that can characterize real-time mobile embedded devices have been used in our experiments :

- Telecommunication applications are the basic elements of mobile devices and are most commonly used in RTOS systems. Applications in this group include *CRC32*, *FFT/IFFT*, *GSM encode/decode* and *ADPCM encode/decode* functions.
- Automotive and Industrial Control applications are intended for embedded control systems, typically for air bag controllers, sensor systems, etc. Tests in this category consist of *Basic Math*, *Bit Count*, *qsort* (i.e. information sorting algorithm) and the *Susan* image recognition package with three different filter algorithms (*smoothing*, *edges* and *corners*).

In these experiments, benchmarks have been ported to both native and virtual $\mu\text{C}/\text{OS-II}$ as real-time tasks. Each benchmark has run for hundreds iterations, and the average cost for one iteration has been calculated as a benchmark result. The experiment results on native and VM are listed in TABLE 4.5. The results that have been obtained with the benchmarks that have run in a virtual machine are very close to those obtained in the native execution. All have more than 90% performance. For the computation-intense applications such as *FFT* and *Bit Count*, the performance loss is below 1%, because these benchmarks are all CPU-bounded applications in which sensitive instructions are merely used. Therefore, they barely have any virtualization overhead and execute similarly as on a native machine. The only extra overhead is due to the handling of virtual timer ticks that occur during their execution.

However, there is a relatively higher performance loss for benchmarks like *GSM*, *qsort*, *CRC32* and *Susan* image processing. These benchmarks are all data-bounded or file-based applications, which mainly and frequently interact with input and output files that are stored in external file system, e.g. FAT32 on SD card. For example, one at a time, the *GSM* benchmark reads a frame of 160 samples from input files, encodes, and writes results to an output file when the process is complete. These accesses have to be performed by OS services. For example, DMA transfers are used to load data from the SD card, which should be performed through hyper-calls. A lot of OS functions are used during the execution. From the results of Thread-Metric benchmarks, we know that the OS service function is influenced by the extra virtualization overheads, and thus causes the degradation of overall performances.

TABLE 4.5 – MiBench experiment results on virtual and native Mini- μ C/OS in milliseconds (ms).

Benchmarks	Native OS (ms)	VM OS (ms)	Performance Ratio (%)	Δ_{VM}^{ei} (ms)
<i>FFT</i> (8192 pts)	54.277	54.614	99.4	0.337
<i>GSM</i> (encode/decode)	204.387	214.668	95.2	10.281
<i>ADPCM</i> (encode/decode)	3391.075	3510.580	96.6	119.505
<i>CRC32</i>	206.685	220.932	93.6	14.247
<i>Basic Math</i>	47.931	49.152	97.5	1.220
<i>Bit Count</i>	133.990	134.722	99.5	0.732
<i>qsort</i>	45.693	48.927	93.4	3.234
<i>Susan</i> (smoothing)	141.362	141.695	99.8	0.334
<i>Susan</i> (edges)	51.192	55.557	92.1	4.365
<i>Susan</i> (corners)	45.777	48.928	93.6	3.151

From the MiBench results, we can also determine the extra execution cost Δ_{VM}^{ei} in Eq.(2.7) as the performance loss :

$$\Delta_{VM}^{ei} = e_i^{VM} - e_i^{Native}. \quad (4.3)$$

As shown in TABLE 4.5, though the performance loss is retained within 10%, the Δ_{VM}^{ei} for different benchmarks are quite heterogeneous, ranging from hundreds of microseconds to tens of milliseconds. In fact, the value of Δ_{VM}^{ei} is determined by the implementation of tasks and by more realistic factors, for example the size of the file that is processed by the task. Thus, no quantitative evaluation can be given in this experiment. Taking into account the qualitative analysis, we can conclude that, one has to avoid frequent usage of OS functions or external device accesses because they will cost a lot in virtual processes.

4.2.4 Real-time Schedulability Analysis

Based on the experiment results that evaluate the virtualization impact on the original RTOS, the RTOS task model $T_k = T_i(E_i^{VM}, p_i, d_i)$ should be carefully reviewed to deal with practical implementations.

Usually, the RTOS time-tick interval is set to 10 ms in common cases, and to 1 ms for high-resolution scheduling [YY14]. For developers that design a task set for virtual RTOSs, Eq.(2.11) is used to calculate the actual execution timer ticks demanded by a task $T_i \in T_k$. Given that $\Delta_{VM}^{Response}$ is evaluated as 3.08 μs (see Section 4.2.2.2), Eq.(2.11) can be simplified as :

$$\Theta_i \approx \lceil \frac{E_i^{Native} + \Delta_{VM}^{ei}}{\Delta_{Tick}} \rceil, \text{ if } \Delta_{VM}^{Response} \ll \Delta_{Tick}. \quad (4.4)$$

Eq.(4.4) indicates that the virtualization cost on task *Response* is unnoticeable in real-world implementations. Therefore, the major influence on RTOS scheduling configuration

is caused by the extra execution overhead Δ_{VM}^{ei} . To guarantee the system schedulability, the practical task execution time has to be accurately measured. For T_k that originally runs on a native RTOS, its schedulability is still guaranteed on a Ker-ONE VM if :

$$\forall T_i \in T_k, \lceil \frac{E_i^{Native} + \Delta_{VM}^{ei}}{\Delta_{Tick}} \rceil = \lceil \frac{E_i^{Native}}{\Delta_{Tick}} \rceil. \quad (4.5)$$

Otherwise, the configuration of scheduling parameters for T_k have to be re-calculated to meet hard real-time constraints.

The configuration of the RTOS timer tick should also be discussed. When the timer tick interval is reduced, RTOSs can fulfill the required scheduling with higher accuracy and higher CPU usage rate. Meanwhile, the task schedulability becomes more sensitive to the virtualization environment. On the other hand, a task set based on a longer tick interval is more tolerant regarding virtualization overheads, but with lower schedule precision. From our experience, we suggest that, for systems with tight timing constraints, the timer tick interval should be reduced to increase the scheduling accuracy, whereas loosely tasks set can be chosen to increase the tick interval, so that it can be easily adapted.

4.3 DPR Allocation Performance Evaluation

In this section, we evaluate the performance of DPR accelerator allocation for virtual machines on Ker-ONE. More specifically, we focus on the overheads associated with the allocation mechanism, and the inevitable extra latency caused by virtualization implementations. Based on these overheads, we have calculated the overall latency for DPR accelerator allocations, which is essential to determine the response time of accelerators and the flexibility. For guest OSs with real-time constrains, this is more critical as it may influence the timing of certain tasks. In the following, we first analyze the overheads contributed to the allocation latency in details. Then we present the experiment results. Based on these results, we also discuss the advantages and disadvantages of our allocation policies.

4.3.1 Overhead Analysis

In this thesis, we define the concept denoted as DPR resource allocation latency. This extra execution path begins when a virtual machine traps at accessing unavailable virtual device, and ends when the DPR accelerator is properly allocated and ready to start. This latency is considered as the response time of DPR accelerators.

Allocation latency comes from two main sources : the allocation management (i.e. the execution of *Virtual Device Manager*), and the extra virtualization mechanism overheads, for example, the handling of page-table faults, Inter-VM communication (IPCs) and virtual machine scheduling. Besides, the costs of DPR accelerator reconfiguration and preemption take up extra time, which will noticeably contribute to the total latency. Considering the above elements, we create the allocation execution model to facilitate

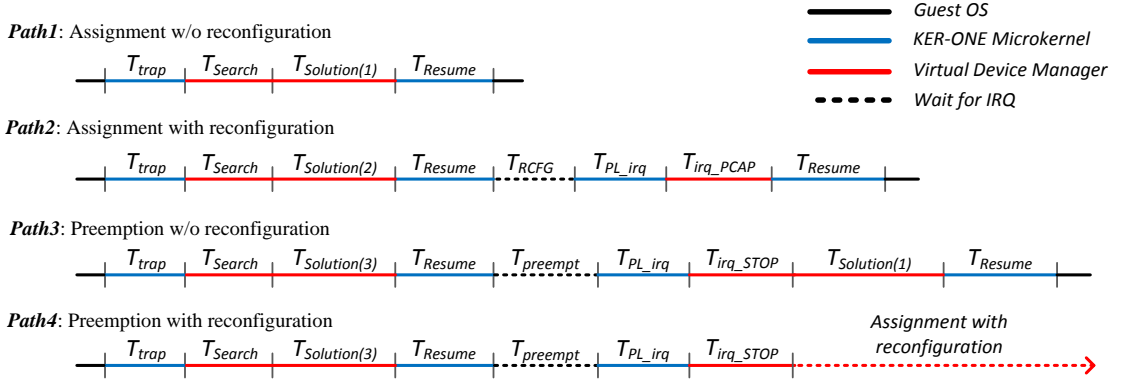


FIGURE 4.6 – The execution paths of DPR resource allocation.

the analysis. Based on the allocation mechanism we proposed, the execution path is only determined by the state of *solutions* (see Section 3.2.4), which can be characterized as 4 execution paths :

- Path 1 (i.e. Solution $\{assign\}$) : Allocate an Idle accelerator to virtual machine without reconfiguration, which is also called immediate allocation, since the virtual machine can start using the virtual device immediately.
- Path 2 (i.e. Solution $\{assign\}$ with *reconfig.*) : Reconfigure an Idle partial re-configuration region (PRR) with the desired accelerator and allocate it to virtual machine.
- Path 3 (i.e. Solution $\{preempt\}$) : Preempt a running accelerator and allocate it to virtual machine without reconfiguration.
- Path 4 (i.e. Solution $\{preempt\}$ with *reconfig.*) : Preempt a running accelerator, which will then be reconfigured with the desired accelerator and get allocated to virtual machine.

The model of these execution paths can be calculated according to the diagrams displayed in FIGURE 4.6. Except for Path 1, other paths are divided into several steps by the events of preemption or reconfiguration. Note that the execution path 4 is equivalent to a combination of preemption (as in path 3) and assignment with reconfiguration (as in path 2), because after the accelerator is successfully preempted, the *Virtual Device Manager* re-runs the solution as normal assignment with reconfiguration, i.e. the process of path 2. This execution is also described in the FIGURE 3.13 in Section 3.2.5.

In this model, each allocation path is decomposed into smaller atomic executions. Note that any specific atomic execution has almost identical path in different paths, and thus is with determined overhead. For example, the execution path of detecting a virtual machine trap and redirecting it to the *Virtual Device Manager* is determined for each path, and is represented as an atomic execution T_{trap} . The detailed description of this execution path model is listed in the following :

- T_{trap} : Time required by Ker-ONE to detect a page-table exception in VM domain and to invoke the *Virtual Device Manager*.
- T_{resume} : Time required by Ker-ONE to schedule back to a VM.
- T_{PL_irq} : Time required by Ker-ONE to receive IRQs from the *PRR Monitor* and redirect them to the *Virtual Device Manager*.
- T_{Search} : Time required by the *Virtual Device Manager* to receive the VM requests and searches for solutions.
- $T_{Solution(1)(2)(3)}$: Execution time of the main function to handle (i.e. *Run_Solution()*) different solutions : (1) direct assignment, (2) assignment with reconfiguration, (3) preemption.
- $T_{irq_pcap}, T_{irq_stop}$: Time required by the *Virtual Device Manager* to handle the following IRQs (i.e. *IRQ_PCAP_Over, IRQ_PRR_Stop*).
- T_{RCFG} : Overhead of waiting for the completion of PCAP reconfiguration.
- $T_{preempt}$: Overhead of waiting for the stop of preempted PRRs.

Therefore, the total allocation latency for the different allocation execution models can be calculated as :

$$\begin{aligned}
T_{Path1} &= T_{trap} + T_{Search} + T_{Solution(1)} + T_{resume}, \\
T_{Path2} &= T_{trap} + T_{Search} + T_{Solution(2)} + 2 * T_{resume} \\
&\quad + T_{PL_irq} + T_{irq_pcap} + T_{RCFG}, \\
T_{Path3} &= T_{trap} + T_{Search} + T_{Solution(3)} + 2 * T_{resume} \\
&\quad + T_{PL_irq} + T_{irq_stop} + T_{Solution(1)} + T_{preempt}, \\
T_{Path4} &= T_{Path3} + T_{Path2} - T_{trap} - T_{Search} - T_{resume} - T_{Solution(1)}
\end{aligned} \tag{4.6}$$

As we have explained, since the execution path 4 is implemented as a combination of preemption and assignment in Eq.(4.6), the latency of T_{Path4} can be approximately estimated from T_{Path2} and T_{Path3} , with some minor adjustments.

Most elements in Eq.(4.6) are system-dependent overheads, which are caused by the virtualization and management software, and their values are determined and are predictable via experiments and measurements. On the other hand, FPGA-dependent overheads, i.e. T_{RCFG} and $T_{preempt}$, are determined by the implementation of accelerators and PRRs on the FPGA side, whose values may vary in with different hardware designs. Therefore they should be considered under specific scenarios. In the next part, we will evaluate these overheads through dedicated experiments.

4.3.2 Experiments and Results

Our evaluation were set on the same platform and environment as in Section 4.2, i.e. Ker-ONE micro-kernel virtualization on Xilinx ZedBoard ARM Cortex-A9 processor.

$\mu\text{C}/\text{OS-II}$ instances are hosted in virtual machines as guest OSs. The operating frequency of the CPU and FPGA logic are 667 MHz and 100 MHz respectively.

To fairly measure the allocation performance of DPR accelerators, it requires dedicated experiments which can be close to real-case scenarios. However, currently to our best knowledge, there is no standard benchmark suite that evaluates the performance of reconfigurable accelerator management in a virtual machine system. It is also difficult for us to establish and implement a real-world reconfigurable computing framework. Therefore, we intend to create an evaluation experiment which is able to emulate the common situation of DPR accelerator usage by following the principles below :

1. Partial reconfiguration regions are implemented with different sizes. For each PRR, different accelerators can be implemented. All DPR accelerators are mapped into virtual machine space as virtual devices.
2. The data frame processed by accelerators should have determined structure, with determined size. Accelerator should respect the completion of the data it processes.
3. Virtual machines are totally independent from each other. Tasks on top of guest OSs access to different virtual devices randomly. In other words, the request for DPR accelerators are predictable, and may come from any virtual machine.
4. Guest OSs have different priority levels, so that preemptions may occur randomly.

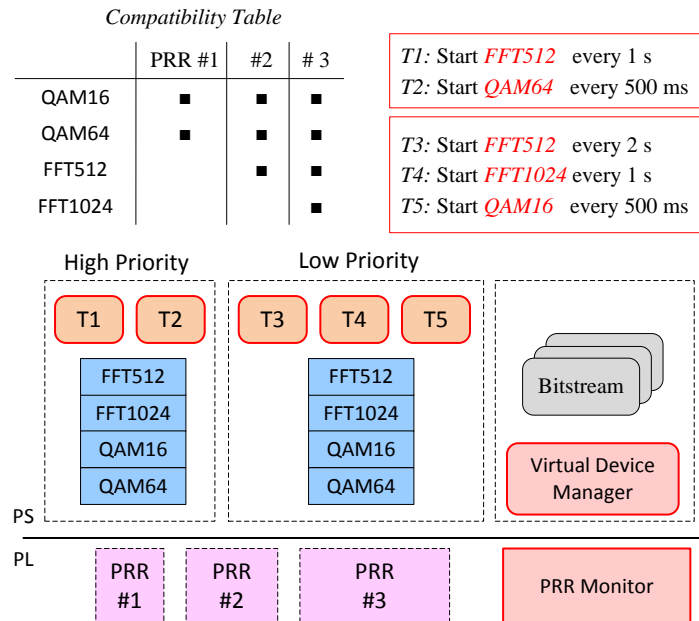


FIGURE 4.7 – Experimental architecture for performance evaluation.

According to the principles above, the FPGA fabric is initially implemented with three PRRs, with different sizes. Four accelerators, i.e. *QAM16*, *QAM64*, *FFT512*, *FFT1024*, have been synthesized into bitstream files. During the initialization stage of Ker-ONE,

these accelerators are mapped in the memory space of virtual machines with unified addresses.

On top of Ker-ONE, two guest $\mu\text{C}/\text{OS-II}$ execute concurrently, with different priority levels. For each guest OS, four available virtual devices are instantiated. Several tasks run in both guest OSs to periodically command virtual devices to process data frames containing 18,800 bits, which causes requests for allocations. The *Virtual Device Manager* detects and handles these requests at run-time. The design is shown in FIGURE 4.7. During the experiment, the various overheads caused by allocation mechanisms were recorded and classified during hours of execution. Based on massive samples, we have calculated and evaluated the overall cost for our allocation mechanism.

The results of our measurements are listed in TABLE 4.6. We should note that Ker-ONE provides an efficient virtualization mechanism, in which virtual machine scheduling and virtual interrupt emulation are performed with an overhead less than $1\mu\text{s}$. The heaviest overhead is caused by $T_{\text{Solution}(2)}$, referring to the PRR assignment with reconfiguration. This is because this process includes the launch of a PCAP transfer which is composed of complex operations to set up the DMA transfer.

TABLE 4.6 – Measurements of Overheads during DPR allocation

Micro-kernel		Virtual Device Manager	
Operation	Overheads (μs)	Operation	Overheads (μs)
T_{trap}	0,76	T_{Search}	0.50
T_{resume}	0,64	$T_{\text{Solution}(1)}$	1.13
$T_{\text{PL_irq}}$	0.81	$T_{\text{Solution}(2)}$	2.77
		$T_{\text{Solution}(3)}$	0.34
		$T_{\text{irq_pcap}}$	0.64
		$T_{\text{irq_stop}}$	0.28

We should also note that the overheads of T_{RCFG} and T_{preempt} are determined by the implementation of accelerators. T_{RCFG} is determined by the size of PRRs, and T_{preempt} depends on the algorithm of accelerators. The computation granularity influences the positions of the *consistency points*, which determines the worst-case waiting time before an accelerator is successfully preempted. Thus, the value of T_{RCFG} and T_{preempt} can be predicted according to the implementation of the PRRs and accelerators. TABLE 4.7 presents the overhead results.

According to the experiment measurements, allocation overheads can be estimated from Eq.(4.6) as :

$$\begin{aligned}
 T_{\text{Path1}} &= 3.03\mu\text{s}, \\
 T_{\text{Path2}} &= 6.76\mu\text{s} + T_{\text{RCFG}}, \\
 T_{\text{Path3}} &= 5.10\mu\text{s} + T_{\text{preempt}}, \\
 T_{\text{Path4}} &= 8.83\mu\text{s} + T_{\text{preempt}} + T_{\text{RCFG}}.
 \end{aligned} \tag{4.7}$$

Note that the estimated overheads in Eq.(4.7) demonstrates the allocation overheads

TABLE 4.7 – Preemption and reconfiguration overheads for DPR accelerators

Virtual Device	$T_{preempt}(\mu s)$	$T_{RCFG}(\mu s)$		
		PRR#1	#2	#3
<i>QAM16</i>	47.0	231	810	1,206
<i>QAM64</i>	31.0	231	810	1,206
<i>FFT512</i>	12.1	-	810	1,206
<i>FFT1024</i>	21.6	-	-	1,206

for the high priority OS, since high priority request can always get a valid solution in our system. It can be clearly noticed that direct allocations can be efficiently performed with $3\mu s$ latency, whereas other solutions suffer from extra overheads of preemption or reconfiguration. The worst-case solution corresponds to the overhead T_{Path4} , which may result in hundreds of microseconds considering the relatively heavy overhead of PCAP reconfiguration.

Based on T_{Path4} , we can derive the worst-case allocation latency of a given virtual device. Assume that N accelerators $\{A_1 \dots A_N\}$ are implemented in K reconfigurable regions $\{P_1 \dots P_K\}$. Each implemented accelerator is denoted as A_n^k , where k represents the host PRR. We denote the worst case allocation latency as R , and then for a certain accelerator A_n , the worst latency R_n can be obtained as :

$$R_n = 8.83\mu s + \max_{i:A_n^i \exists} \{T_{RCFG}(i) + \max_{j:A_j^i \exists \cap j \neq n} T_{preempt}(A_j^i)\}. \quad (4.8)$$

In this equation, we estimate the worst-case by considering all PRRs that are compatible for A_n . $T_{RCFG}(i)$ stands for the reconfiguration overhead of region P_i , which is also the reconfiguration overhead of A_n^i . $T_{preempt}(A_j^i)$ represents the overhead when A_n^i preempts other accelerators in P_i .

From the equations Eq.(4.7) and Eq.(4.8), we can notice that virtualization and allocation mechanisms in Ker-ONE framework have only a slight impact (several microseconds) on the overall performance, while $T_{preempt}$ and T_{RCFG} are inevitable in most DPR systems. On the other hand, for low priority OS, the allocation latency is unpredictable since its requests may be suspended until a valid solution is found.

TABLE 4.7 presents the overhead measurement results of the accelerators used in our experiment. In our example, T_{RCFG} causes the heaviest overhead, which results in a large latency value of R (more than $1ms$). Meanwhile, $T_{preempt}$ is significantly lower than T_{RCFG} . Actually, this is the case in most DPR accelerator implementations. Therefore, a preemptive allocation of DPR resources can effectively reduce the heavy reconfiguration overheads, which, however, will also considerably undermine the schedulability of low priority OS tasks. In a system where preemption occurs frequently, low priority virtual machine may never get DPR resources.

The trade-off between preemption and consistency has been considered when we chose the solution selecting policy in the *PRR Monitor* (see Section 3.2.4). In our current

searching policy, allocating *Idle* PRRs are preferred than preempting PRRs, as we wish to respect the execution of tasks in low priority virtual machines. However, in a system where the higher priority OS is in charge of critical tasks or is used as a real-time OS, an alternative policy which encourages preemption should be applied, so that high priority tasks are guaranteed to acquire DPR resources with minimum latency.

4.3.3 Real-time Schedulability Analysis

In section 4.3.2, we discussed the allocation overheads of reconfigurable resources. In the context of real-time virtualization, this factor brings another problem, that is such extra overheads would inevitably influence the timing characteristics of the system. In this section we will re-analyze the real-time schedulability, based on the consideration of the additional allocation latency.

Considering the execution overhead model for a real-time task set T_i in guest OS, the actual execution time is composed of the response time $Release^{VM}$ and execution path e_i^{VM} (see Eq.(2.7) and Eq.(2.8)). Considering that DPR accelerators are dynamically allocated and shared among virtual machines, both $Release^{VM}$ and e_i^{VM} are affected. In this context, we only focus on real-time guest OS tasks, since the timing constraints of these tasks are most critical and should be guaranteed in embedded systems.

4.3.3.1 Response Time

The response time of real-time tasks (i.e. $Release^{VM}$) is influenced by the allocation latency due to the fact that the execution of *Virtual Device Manager* cannot be preempted by real-time events, since it executes at a higher priority level. In other words, during the allocation process, the real-time OS scheduling is blocked and this prevents the immediate response to schedule real-time tasks. In the $Release^{VM}$ equation Eq.(2.8), we brought in the factor of VMM critical execution ($\Delta VMMCritical$), since this process cannot be preempted by real-time events. In the context of DPR allocation, the non-preemptive execution of *Virtual Device Manager* should be considered as another critical execution, based on which the model of $Release^{VM}$ can be adjusted as :

$$Release'^{VM} = relEv^{VM} + \Delta VMMSched + \max\{\Delta VMMCritical, \Delta DPRCritical\} \quad (4.9)$$

In this new model, the additional parameter $\Delta DPRCritical$ stands for the critical execution during the DPR allocation path, and the worst-case critical execution should consider the critical execution overheads caused by both VMM ($\Delta VMMCritical$) and the allocation ($\Delta DPRCritical$).

From the model of DPR allocation path in FIGURE 4.6, we can notice that the critical overhead is only related to the execution of *Virtual Device Manager*, while the process of reconfiguration (T_{RCFG}) or preemption ($T_{preempt}$) have no influence on the response time. Therefore, the critical executions in allocation paths can be calculated as in the TABLE 4.8.

TABLE 4.8 – Critical execution calculation in DPR allocation paths

Path	Model	Description	Overhead
Path 1	$T_{trap} + T_{Search} + T_{Solution(1)} + T_{Resume}$	Direct allocation	$3.03\mu s$
Path 2	$T_{trap} + T_{Search} + T_{Solution(2)} + T_{Resume}$	Main function	$4.67\mu s$
	$T_{PL_irq} + T_{irq_pacp} + T_{Resume}$	IRQ Routine	$2.09\mu s$
Path 3	$T_{trap} + T_{Search} + T_{Solution(3)} + T_{Resume}$	Main function	$2.24\mu s$
	$T_{PL_irq} + T_{irq_stop} + T_{Solution(1)} + T_{Resume}$	IRQ Routine	$2.86\mu s$
Path 4	$T_{trap} + T_{Search} + T_{Solution(3)} + T_{Resume}$	Main function	$2.24\mu s$
	$T_{PL_irq} + T_{irq_stop} + T_{Solution(2)} + T_{Resume}$	IRQ Routine	$4.50\mu s$
	$T_{PL_irq} + T_{irq_pcap} + T_{Resume}$	IRQ Routine	$2.09\mu s$

In this table, the allocation paths are decomposed into several periods Based on the experiment results obtained from section 4.3.2, their execution overheads can be precisely calculated following the models presented in the table.

Based on this calculation, the worst-case RTOS response time $Response^{VM}$ in Eq.(4.8) is $7.01\mu s$. To estimate the influence of DPR allocation mechanism on the real-time scheduling parameters, the equation in Eq.(2.9) can be derived into :

$$\Delta^{Response}_{VM} = \Delta^{VM}_{relEv} + \Delta VMM Sched + \max\{\Delta VMM Critical, \Delta DPR Critical\}. \quad (4.10)$$

In this equation, $\Delta^{Response}_{VM}$ represents the extra overhead impact when we consider the factor of DPR allocation. Its value can then be calculated to be $6.28\mu s$.

Note that we have measured the overhead impact $\Delta^{Response}_{VM}$ (as $3.08\mu s$) in section 4.2.2.2, in the context where DPR resources were not involved. This result shows that With DPR allocation mechanism, the response time for real-time tasks would be prolonged for several micro-seconds. We should note that, such an augment may only slightly influence the schedulability of real-time tasks, considering the time resolution is set to 1ms or 10ms (see Eq.(4.4)).

4.3.3.2 Execution Time

Determining the worst-case execution time of tasks is critical when designing real-time scheduling parameters. In our system, where reconfigurable accelerators are used, the execution of tasks depends on not only software computing but also hardware resources. The execution time of guest OS tasks may be significantly prolonged if the hardware computation takes a long time to complete.

In fact, the proposed DPR management mechanism grants a guest RTOS several advantages to reduce its impact on the task execution, including :

1. the RTOS is held at higher priority level than GPOSs. DPR requests from RTOS can always find valid solutions immediately, since it can preempt any running accelerators of other VMs.
2. The accelerators being used by RTOS can never be re-allocated (or preempted) to other VMs, unless the computation is complete.

Despite these principles, the execution of RTOS tasks may still be prolonged by the allocation of accelerators, which is inevitable when the DPR resources are shared by multiple virtual machines.

Imagine a RTOS task t_i which utilizes n accelerators $\{A_1 \dots A_n\}$ to speed up its computation. The worst-case execution overhead model (e_i^{VM}) in Eq.(2.7) can be extended as :

$$e_i^{VM} = e_i^{Native} + \Delta_{VM}^{ei} + \sum_{k=1}^n R_k. \quad (4.11)$$

Whereas Δ_{VM}^{ei} stands for the extra overheads caused by virtualization, the parameter R_k is the allocation latency for the request of accelerator A_k , as in Eq.(4.8). All used accelerators should be taken into consideration when calculating the WCET of tasks.

The model for actual scheduling parameters in Eq.(4.4) can then be adjusted as :

$$\Theta'_i \approx \left\lceil \frac{E_i^{Native} + \Delta_{VM}^{ei} + \sum_{k=1}^n R_k}{\Delta^{Tick}} \right\rceil, \text{ if } \Delta_{VM}^{Response} \ll \Delta^{Tick}. \quad (4.12)$$

As in Eq.(4.4), the value of $\Delta_{VM}^{Response}$ is still ignored here. And Δ^{Tick} is still estimated to be 1ms or 10ms.

In some implementations (e.g. the example results in TABLE 4.7), the value of allocation latency R could be several milliseconds. In this case, the extra overheads of R may possibly influence the scheduling parameters. Therefore, it is mandatory that RTOS users take into account this factor to calculate the full execution overheads of tasks and to make scheduling configurations.

From section 4.3.2, we have learned that the allocation latency of accelerators depends on the implementations such as their reconfigurable region sizes and their algorithms. It is therefore impossible to present constant quantitative timing characteristics of our system. However, with the equation Eq.(4.7) and Eq.(4.8), the WCET of real-time tasks can be calculated based on measurements and experiment results.

4.4 Summary

In this chapter we have introduced the implementation details of Ker-ONE micro-kernel, which is based on the Xilinx Zynq-7000 platform, and have evaluated the performance of both virtualization and DPR accelerator allocation mechanisms with extensive experiments. The results have also been thoroughly discussed in the context of real-time embedded systems.

To present the implementation of Ker-ONE, we have described the target platform and the efforts to para-virtualize a RTOS $\mu C/OS-II$. We have also demonstrated the result

code complexity of our micro-kernel. By comparing with other existing technologies, i.e. XEN-ARM and OKL4, we have shown that our approach manages to achieve smaller TCB kernel size because of the lightweight mechanism we applied. We also introduced the system mapping and initialization sequence of our system.

Then we have followed several experiments in order to evaluate the virtualization performance of Ker-ONE. The experiments are based on both custom and standard benchmarks. In the first step, we evaluated the basic VMM functions to make a general evaluation. Then we focus on the performance of guest RTOSs. With this target in mind, we have performed qualitative tests with RTOS functions, and measured the actual overheads impact on the scheduling of guest RTOS. In the end, we ran some real-world embedded applications to understand the overall speed. Based on the obtained results, we have presented an extensive analysis on the real-time schedulability of guest RTOS. The results show that our approach implements real-time virtualization with low overheads and close-to-native performance.

Furthermore, we attempted to estimate the allocation overheads during the allocation of DPR accelerators. we modeled the execution path according to the mechanism presented in CHAPTER 3. In the proposed models of allocation, processes are broken down into smaller execution paths, which are then respectively measured in a dedicated experiment. The results proved that our management framework is able to efficiently allocate DPR accelerators to virtual machines, with low extra overheads. Based on these experiment results, we have made further analysis about the overall influence of system on the schedulability of real-time tasks.

CHAPTER 5

Conclusion and Perspectives

5.1 Summary

Nowadays, embedded systems are playing an important role in the daily life of most people, ranging from customer products such as smartphones and vehicles, to industry domains. With such an expanded range of purposes, embedded systems can be put into different categories. There are systems with high computing power, which can support complex software stacks and enormous resources. There are also small-scaled embedded systems with limited resources, that are intended for low-cost simple devices, such as the Internet-of-Things (IoT). Basically, most of these devices share common characteristics such as a small size, low weight and low power consumption.

Meanwhile, while the complexity of embedded systems is increasing, it is becoming more and more expensive to improve CPU performance by conventional approaches, i.e. IC scaling and ASICs. In this context, the concept of heterogeneous CPU-FPGA architecture has become a promising solution for SoC device vendors, because of the fast time-to-market circle, the high adaptability and the relatively-low cost to improve the computation ability. This emerging convergence point of conventional CPU and FPGA computing makes it possible to extend traditional CPU virtualization technologies into the FPGA domain to fully exploit the mainstream FPGA computing. To achieve this goal, it is necessary to propose an architecture which enhances the ability of existing technology while respecting the features of both software and hardware components.

Our research makes a contribution in this domain, by studying the real-time virtualization in embedded systems with the dynamic partial reconfiguration (DPR) technology. The target object of our research is small-scaled simple CPU-FPGA embedded systems, where the main limitations include computing resources, small memory size, tight timing constraints and power budget. Our works focus on two aspects : a custom lightweight highly-adaptable micro-kernel Ker-ONE, that supports real-time virtualization, and an innovative coordination mechanism of DPR accelerators among multiple virtual machines.

In CHAPTER 1, we began the thesis by introducing the basic technical concepts and existing technologies of embedded virtualization, real-time scheduling and partial re-

configuration. We presented the fundamental theories for virtualization, and introduced the approach of hypervisors (i.e. XEN) and micro-kernels (i.e. L4, OKL4) in domain of embedded system. Then we discussed the real-time scheduling algorithms for virtualization, mainly the compositional scheduling framework. We have introduced the features of these approaches, and have demonstrated that they introduce relatively heavy-weight software, which is against our purpose. In the following part, we have discussed about the application of DPR technology in software and operating systems. In these approaches, hardware virtualization technology was used for multiplexing FPGA resources among multiple users. However, research efforts dedicated on DPR management in embedded virtualization system were not sufficient. At the end of CHAPTER 1, the motivation of our research was explained.

In CHAPTER 2, we described the architecture of the Ker-ONE micro-kernel, which we have been built from bottom-up to provide ARM para-virtualization, following the principle of a lowest complexity and high adaptivity. In the first part of this chapter, we introduced the virtualization mechanism. Our design eliminated unnecessary VMM functionality and focused only on essential execution resources, e.g. CPU, coprocessor, memory, interrupt, timer, IPC, etc. Dedicated methods were taken to reduce the development cost and to optimize the performance. The second part of this chapter proposed a preemptive scheduling algorithm which can realize real-time virtual machine (RTOS) with low software complexity. To analyze the scheduability, we proposed an extensive overhead model to present the impact of virtualization on RTOS timing constrains.

Then in CHAPTER 3, based on the Ker-ONE micro-kernel, we proposed a custom management framework which allocates DPR accelerators. The target of this approach is to create a transparent, efficient and secure framework for virtual machines. We started this chapter by demonstrating the management mechanism, which includes the reconfigurable infrastructure on the FPGA fabric, the virtual device management on the virtual machine memory space, and the cooperation from HW/SW sides. The major content was about the allocation requests/solution mechanisms, and how they were carried out by the independent user-level service *virtual machine manager*. Then we made specific discussions about the security of virtual machines in this context, showing that the sharing of DPR resources does not undermine the isolation. At the end, we took the example of a guest OS, and gave some practical suggestions for the user software programming policy to use DPR resources.

In the last part of thesis, we presented the overall evaluations of the proposed system. The implementation of Ker-ONE results in a small TCB kernel size. The virtualization performance was estimated from two aspects. We measured precisely the extra overheads via custom benchmarks base in which we have also analyzed the guest OS schedulability. We have also studied the performance loss compared to native machine by running standard benchmarks, i.e. Thread-Metric and MiBench. The experiments demonstrated that Ker-ONE was able to host guest OS with low performance loss. Then we examined the DPR allocation mechanism by determining the corresponding latencies and studying its influence on the overall real-time schedulability. We modeled the possible execution paths of allocation and measured their overheads respectively. It can be noticed in the

end that our framework was capable of dynamically coordinating DPR accelerators for virtual machines with high efficiency.

5.2 Perspectives and Future Works

The proposed micro-kernel provides a lightweight real-time virtualization approach with DPR support. Ker-ONE results in low complexity and small footprint, and merely relies on platform-specific architectures. Though developed on the Zynq-7000 platform, it can easily adapt to other ARMv7-based embedded systems. Moreover, Ker-ONE remains adaptable and might be extended to additional features and mechanisms. Meanwhile, there are some future works that may improve our system.

First, our research is currently focusing on simple OSs with single protection domain, which permits us to use a simplified shadow-mapping mechanism. In the future works, we would like to introduce more sophisticated memory management so that multiple guest OS processes (i.e. page tables) can be supported. With this improvement, more complex OS such as Linux can be hosted in our virtual machine. This will certainly expand the usage of Ker-ONE in both academic and industrial domains. We would like to implement a conditional kernel compilation, where users may choose the simple implementation for smaller kernel size, or the complex one to use Linux in the virtual machine.

Second, with the development of hardware virtualization extensions in ARM architectures, more and more embedded systems use CPUs with such features, including ARMv7-A (e.g. Cortex-A15) and ARMv8 (e.g. Cortex-A53). FPGA vendors such as Xilinx are also planning to release FPGA board with Cortex-A15. In this case, we would like to port our approach to more advanced ARM architecture to fully exploit the hardware virtualization extension. With dedicated hardware support, it can be foreseen that Ker-ONE will go through a significant performance improvement, better security, as well as simpler kernel software.

Third, in our current research, the allocation policy of DPR resource allocation hasn't been fully studied. It would be interesting to develop more sophisticated solution searching algorithm for allocations, and discuss the influence of different parameters on the usage of DPR accelerators in different priority virtual machines. For example, the factors of hardware task granularity, solution selecting policy or the size of reconfigurable regions may all influence the allocation and preemption of different accelerators, and their usages by guest OSs.

Last but not the least, we are interested in implementing some real-world scenarios in our framework, e.g. applications in telecommunication or vehicle domains, so that we can rationally evaluate the performance of our framework compared to existing solutions in these domains, and evaluate how much the computation can be improved by using CPU-FPGA architectures. Currently, our research team is trying to build a full communication processing system, including OFDM and WIFI chains, on top of Ker-ONE, which would help estimating Ker-ONE in practical situations. This approach may also be proposed for general telecommunication systems which require flexibility and low complexity.

LIST OF FIGURES

1	Systèmes de virtualisation complète et de para-virtualisation sur une architecture ARM.	v
2	Ker-ONE se compose d'un micro-noyau exécuté en mode privilégié et d'un environnement utilisateur s'exécutant au niveau non-privilégié.	viii
3	Surcoût d'exécution des tâches d'un RTOS lorsqu'exécutée dans une machine virtuelle. Ce dernier est composé de surcoûts critiques liés à l'ordonnancement et les surcoûts intrinsèques liés au RTOS.	xi
4	Vue d'ensemble de la gestion de la zone reconfigurable dans KER-ONE.	xii
1.1	Typical virtual machine system architecture	6
1.2	Two-stage address mapping in virtual machine systems.	9
1.3	Using shadow mapping to perform two-stage address mapping.	9
1.4	Virtualization extension provided by Intel VT-x/VT-i, where the guest software execute in original privilege levels, but in the non-root mode.	12
1.5	In Cortex-A15 with hardware virtualization extension, guest OSs run at the same privilege structure as before so that they can run the same instructions. New HYP mode has higher privilege, where VMM is established to control wide range of OS accesses to hardware.	13
1.6	With DBT technology, guest OS kernel instruction are rewritten. During executing, revised codes execute directly on processor as native codes.	14
1.7	Para-virtualization replaces the non-privilege sensitive instructions in OS code with hyper-calls to emulate these behaviors.	14
1.8	Mobile handset unit sales by type, and smartphones share of unit sales, worldwide, 2011-2018 [Source : Analysys Mason, 2014][dR14]	17
1.9	Types of 32-bit processors that are preferred by embedded device vendors. The results are listed for year 2013 and 2014, based on a wide-range survey on worldwide embedded developers. [Source : UBM Tech 2014 Embedded Market Study, 2014][Tec14]	19
1.10	Architecture of Fiasco L4 virtualization. Guest user threads are supported by Linux server following the sequence :(1)thread syscall to request Linux service ; (2)L4 micro-kernel takes syscalls and redirects it to Linux server ; (3)syscall handiing ; (4)L4 returns to user thread via resuming execution context or inter-process communication.	20

1.11	The virtualization framework of OKL4 microvisor on ARM architecture.	22
1.12	The virtualization path of KVM virtual machines. KVM plays as an intermediate layer that receives the request of emulation from virtual machines, and redirects them to either QEMU or to the Linux kernel [DLC ⁺ 12].	23
1.13	The architecture of Xen-on-ARM hypervisor.	24
1.14	The TrustZone technology in ARM processors.	26
1.15	Using hypervisor mode and security extensions to build up virtual machine systems.	26
1.16	Hierarchical scheduling framework in virtual machine systems.	27
1.17	Compositional scheduling framework in hierarchical systems. Periodic resource model is used to abstract component real-time constraints and establish scheduling interface between layers. Different scheduling algorithms can be used for different components and layers.	28
1.18	General implementation of compositional scheduling framework in virtual machine systems.	29
1.19	The scheduler architecture of RT-Xen.	30
1.20	Execution of servers with the WCPS and CRSP policies.	31
1.21	The general architecture of CPU/FPGA hybrid processors, with CPU and FPGA being implemented in independent packages.	33
1.22	Dynamic Partial Reconfiguration permits users to modify a given area of FPGA circuit on-the-fly, while the rest fabric functions normally.	34
1.23	The concepts of two CPU/FPGA models : <i>offload model</i> and <i>unified model</i> . In <i>offload model</i> , application is executed on CPU cores and FPGA is used as an accelerator. In <i>unified model</i> the application is divided in parts and mapped to CPU and FPGA. OS helps in seamless execution of application parts in the hybrid system.	36
1.24	With DPR resources used by OS or multiple users, the offload model should be modified. Because applications/threads still access FPGA accelerators as peripheral devices, the OS kernel has to supervise their accesses as an intermediate layer.	37
1.25	Dynamic Partial Reconfiguration architecture modeling as a group of computing agency with multiple hardware threads.	38
1.26	<i>ARTICo</i> ³ handles the requests of two parallel applications APP#1 and APP#2. While APP#1 arrives first, the Resource Manager allocates all available DPR slots to accelerate the computation. Then a higher-priority application APP#2 arrives, which drives the <i>Resource Manager</i> to reconfigure slots of APP#1 to work on the new computation.	39
1.27	Hardware resource virtualization in the OS4RS framework. Logic resources, i.e. device node modules, can be linked to hardware functions as many-to-one manner, so that one hardware function can be shared by multiple clients. On the other hand, one device node can be linked to alternate hardware modules so that it can always access available resources.	40

1.28	An intermediate fabric region is composed of a data plane that hosts the programmable hardware computing components, and of the control plane that holds and defines the behavior and context frame of hardware tasks.	41
2.1	Para-virtualization systems on ARMv7 architecture.	47
2.2	Ker-ONE consists of the micro-kernel and virtual machine monitors in privileged level and User environment in non-privileged level.	48
2.3	The allocation of general-purpose registers (R0-R14) and PC (R15) register for different ARMv7 execution modes.	49
2.4	The management of Program Status Registers in ARMv7 architecture.	50
2.5	ARMv7 processor modeling by decoupling integrated functionalities.	51
2.6	The mechanism of CPU resource virtualization. Resource $\langle G, P \rangle$ is managed in a simple save/restore method, and resource $\langle M, C \rangle$ have to be emulated as virtual resources for virtual machines.	52
2.7	a VCPU has three major properties : (1) holds the virtual machine execution context (EC); (2) emulates sensitive instructions; (3) provides access to other system resources/functionality. Virtual machines run on independent vCPUs that play the role of an intermediate layer between virtual machines and the rest of the VMM.	54
2.8	VFP virtualization mechanism via lazy virtualization. (a) <i>VFP Contexts</i> (FPC) are created for VM1 and VM3 respectively since they have processes that use VFP resources. Guest OS are responsible of the preservation of Local FPCs. (b) VFP is disabled at each VM switch, and its register contents are switched only when necessary.	56
2.9	The access control among different address space privileges in virtual machine systems : host, guest kernel and guest user.	58
2.10	By associating memory pages to different DACR domains, the memory space of virtual machines is divided into several domains with different access policies.	58
2.11	The mechanism of address space virtualization by using shadow mapping mechanism and hyper-calls.	59
2.12	Event management in native machines and virtual machine systems.	62
2.13	The behavior and states of hardware interrupts managed by the Generic Interrupt Controller (GIC).	63
2.14	The process of physical interrupts being handled and virtualized through the virtual GIC.	64
2.15	Three independent physical timers are provided to the micro-kernel, RTOS and GPOSs respectively, which can be accessed directly. For one guest OS, only one timer interface is mapped in the page table.	66
2.16	The IVC mechanism leveraging VM/VMM shared memory region.	67
2.17	Implementation of virtual PSR and vGIC interface in a VM/VMM shared memory region.	68

2.18	RTOS Priority-based scheduling mechanism with independent physical timers. An RTOS is added to the run queue and preempts the other OS when RTOS events occurs.	71
2.19	Overhead of an RTOS task execution in VM, which is composed of VMM critical overheads, VMM scheduling overheads and RTOSs intrinsic overheads.	73
2.20	Composition of overall execution time in the virtual machine context, which is composed by the overheads of response time ($Response^{VM}$) and execution time (e_i^{VM}).	74
3.1	The overview architecture of Zynq-7000 ARM-FPGA hybrid platform. . .	78
3.2	The physical address mapping of PL resources via the AXI-GP interface. .	79
3.3	The partial reconfiguration technology on Xilinx FPGA.	80
3.4	HW tasks and the implementation of virtual devices.	82
3.5	Allocation of virtual devices for virtual machines via manipulating the mapping of of IFs.	83
3.6	Overview of the DPR management framework in Ker-ONE.	84
3.7	HW task index table with <i>HW task descriptors</i>	85
3.8	Implementation of the <i>PR accelerator interface</i> for virtual devices. . . .	85
3.9	The behavior of PRRs as a state machine.	87
3.10	The solution searching sequence and the selecting policy for solutions in the <i>PRR Monitor</i> logic.	89
3.11	The interaction between the PPR monitor and the <i>Virtual Device Manager</i> to search for appropriate allocation solutions.	90
3.12	Execution flow for solution $\{assign(prr01), non-Reconfig\}$ to directly allocate PRR #1 to VM #1.	91
3.13	The process of Virtual Device Manager handling <i>Solution</i> $\{vm_id, dev_id, Method(prr_id), Reconfig\}$	94
3.14	The isolated execution environment of virtual machine and its allocated PR accelerator.	96
3.15	An example in which the $\mu C/OS-II$ guest uses a binary semaphore to handle the IPC signals.	97
3.16	An example that Guest $\mu C/OS-II$ handles the preemption of virtual devices by calling a dedicated task to re-launch the interrupted computation.	98
4.1	Ker-ONE Memory mapping from physical to virtual machine address space.	105
4.2	Ker-ONE initialization sequence divided into stages. Stage 0 : U-boot load files from SD card and start micro-kernel. Stage 1 : micro-kernel initialize system and creates VM0. Stage 2 : User environment boot up guest OSs and services into independent virtual machines.	106
4.3	Results of basic virtualization function overheads evaluation in microseconds (μs) with minimum, average and maximum values.	108
4.4	Comparison of Thread-Metric <i>Performance Ratio</i> (R_P) for para-virtualized $\mu C/OS-II$ on Ker-ONE and XEN-ARM.	112

4.5	Results of VM RTOS task response overheads evaluation in microseconds (μs) with minimum, average and maximum values.	114
4.6	The execution paths of DPR resource allocation.	118
4.7	Experimental architecture for performance evaluation.	120

LIST OF TABLES

1	Les overheads de l'allocation des DPR	xv
1.1	Comparisons of existing virtualization solutions.	15
1.2	Required characteristics of embedded devices for current and future applications.[Source : European Commission DG CNECT Report SMART 2009/0063] [AP12]	18
2.1	Description of ARMv7 execution modes.	47
2.2	Description of ARMv7 execution modes.	50
2.3	Sensitive non-privilege 32-bit ARM instructions.	53
2.4	The configuration of <i>Access Permission</i> Flag and DACR Domain State for three different privilege levels : <i>guest user</i> , <i>guest kernel</i> and <i>host kernel</i>	59
2.5	List and description of interrupt priority layers.	65
2.6	Advantages and drawbacks of two policies for virtual resource management.	69
2.7	Pseudo codes of the task context save/resume process with two policies. <i>Policy 1</i> : Virtual PSR is held in VMM domain and is accessed via hyper-calls. <i>Policy 2</i> : Virtual PSR is held in VM domain and accessed directly.	70
3.1	Technical details of the AXI interface	79
3.2	List and description of ports in <i>PR accelerator interface</i>	86
3.3	Contents of the <i>PRR descriptor</i> data structure.	87
4.1	Development platform information for the proposed approach.	102
4.2	Qualitative comparison of $\mu\text{C}/\text{OS-II}$ source code modification in different characteristics of Ker-ONE and XEN-ARM kernels.	103
4.3	Comparison of TCB size for ARMv6/v7 virtualization environments measured in LoC.	104
4.4	Thread-Metric benchmarks results on both native and virtual $\mu\text{C}/\text{OS-II}$	111
4.5	MiBench experiment results on virtual and native Mini- $\mu\text{C}/\text{OS}$ in milliseconds (<i>ms</i>).	116
4.6	Measurements of Overheads during DPR allocation	121
4.7	Preemption and reconfiguration overheads for DPR accelerators	122
4.8	Critical execution calculation in DPR allocation paths	124

BIBLIOGRAPHY

- [AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11) :2–13, 2006.
- [Aic14] Mohamed El Mehdi Aichouch. *Evaluation of a multiple criticality real-time virtual machine system and configuration of an RTOS's resources allocation techniques*. PhD thesis, INSA de Rennes, 2014.
- [Alt15] Altera. *Cyclone V Device Overview*. Altera Corporation, 2015.
- [AP12] Gabriella Cattaneo Nathalie Feeney Lorenzo Veronesi Cyril Meunier Alain Petrissans, Stephane Krawczyk. Final study report : Design of future embedded systems (smart 2009/0063). Technical report, IDC France, 2012.
- [APA⁺06] Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. Run-time services for hybrid cpu/fpga systems on chip. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 3–12. IEEE, 2006.
- [APL11] Andreas Agne, Marco Platzner, and Enno Lübbers. Memory virtualization for multithreaded reconfigurable hardware. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 185–188. IEEE, 2011.
- [APN13] Mehdi Aichouch, Jean-Christophe Prevotet, and Fabienne Nouvel. Evaluation of an rtos on top of a hosted virtual machine system. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 290–297. IEEE, 2013.
- [ARM12] ARM ARM. Architecture reference manual. armv7-a and armv7-r edition. *ARM DDI C*, 406, 2012.
- [ARM13] ARM. *ARM Generic Interrupt Controller Architecture Specification (ARM IHI0048B)*, 2013.
- [BA07] B Brandenburg and J Anderson. Feather-trace : A lightweight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–28, 2007.
- [Bau15] Felix Baum. *Securing Devices with Embedded Virtualization and ARM TrustZone Technology*. Mentor Graphics Corporation, 2015.

- [Bes13] Joel Best. *Real-Time Operating System Hardware Extension Core for System-on-Chip Designs*. PhD thesis, 2013.
- [BHH⁺07] Jürgen Becker, Michael Huebner, Gerhard Hettich, Rainer Constapel, Joachim Eisenmann, and Jürgen Luka. Dynamic and partial fpga exploitation. *Proceedings of the IEEE*, 95(2) :438–452, 2007.
- [BPS15] Meena Belwal, Madhura Purnaprajna, and TSB Sudarshan. Enabling seamless execution on hybrid cpu/fpga systems : Challenges & directions. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015.
- [BSB⁺14] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Fpgas in the cloud : Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116. IEEE, 2014.
- [BSH11] Bernard Blackham, Yao Shi, and Gernot Heiser. Protected hard real-time : The next frontier. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 1. ACM, 2011.
- [CAA09] Tommaso Cucinotta, Gaetano Anastasi, and Luca Abeni. Respecting temporal constraints in virtualised services. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, pages 73–78. IEEE, 2009.
- [DH10] Hanfei Dong and Qinfen Hao. Extension to the model of a virtualizable computer and analysis on the efficiency of a virtual machine. In *2010 Second International Conference on Computer Modeling and Simulation*, pages 503–507. IEEE, 2010.
- [DJ11] Christoffer Dall and Nieh Jason. Kvm for arm. In *Proceeding of Linux Symposium*, pages 45–56, 2011.
- [DLC⁺12] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. Armvisor : System virtualization for arm. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 93–107, 2012.
- [DN14] Christoffer Dall and Jason Nieh. Kvm/arm : the design and implementation of the linux arm hypervisor. *ACM SIGARCH Computer Architecture News*, 42(1) :333–348, 2014.
- [dR14] Ronan de Renesse. Smartphone markets : worldwide trends, forecasts and strategies 2014-2018. Technical report, Ronan de Renesse, 2014.
- [DRB⁺10] Julio Dondo, Fernando Rincón, Jesus Barba, Francisco Moya, Francisco Sanchez, and Juan Carlos López. Persistence management model for dynamically reconfigurable hardware. In *Digital System Design : Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 482–489. IEEE, 2010.

- [Exp07] ExpressLogic. *Measuring Real-Time Performance Of An RTOS*, 2007.
- [Fle05] Bryan H Fletcher. Fpga embedded processors : revealing true system performance. In *Embedded Systems Conference*, pages 1–18, 2005.
- [FM02] Xiang Feng and Aloysius K Mok. A model of hierarchical real-time virtual resources. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 26–35. IEEE, 2002.
- [GBL⁺09] Maria E Gonzalez, Attila Bilgic, Adam Lackorzynski, Dacian Tudor, Emil Matus, and Irv Badr. Ict-emuco. an innovative solution for future smart phones. In *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, pages 1821–1824. IEEE, 2009.
- [Gol74] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6) :34–45, 1974.
- [GRE⁺01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench : A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.
- [Gup15] Prabhat K Gupta. Xeon+ fpga platform for the data center. In *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119, 2015.
- [Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4) :238–241, 1970.
- [HD10] Scott Hauck and Andre DeHon. *Reconfigurable computing : the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2010.
- [Hei08] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16. ACM, 2008.
- [HGNC10] Michael Hübner, Diana Göhringer, Juanjo Noguera, and Jürgen Becker. Fast dynamic and partial reconfiguration data path with low hardware overhead on xilinx fpgas. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.
- [HH09] Chun-Hsian Huang and Pao-Ann Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *Embedded Systems Letters, IEEE*, 1(1) :19–23, 2009.
- [HL10] Gernot Heiser and Ben Leslie. The okl4 microvisor : convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.
- [Hoh96] Michael Hohmuth. Linux-emulation auf einem mikrokern. *Master's thesis, Dresden University of Technology, Dept. of Computer Science*, 1996.

- [Hor07] Chris Horne. Understanding full virtualization, paravirtualization and hardware assist. *White paper, VMware Inc*, 2007.
- [HSH⁺08] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm : System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.
- [ISM09] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110, 2009.
- [JHE⁺13] Krzysztof Jozwik, Shinya Honda, Masato Edahiro, Hiroyuki Tomiyama, and Hiroaki Takada. Rainbow : An operating system for software-hardware multitasking on dynamically partially reconfigurable fpgas. *International Journal of Reconfigurable Computing*, 2013 :5, 2013.
- [JPC⁺14] Abhishek Kumar Jain, Khoa Dang Pham, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. Virtualized execution and management of hardware tasks on a hybrid arm-fpga platform. *Journal of Signal Processing Systems*, 77(1-2) :61–76, 2014.
- [Kai08] Robert Kaiser. Alternatives for scheduling virtual machines in real-time embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 5–10. ACM, 2008.
- [KBT08] Dirk Koch, Christian Beckhoff, and Jürgen Teich. Recobus-builder-a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 119–124. IEEE, 2008.
- [KEH⁺09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4 : Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.
- [Kon12] Kontron. *No end to the possibilities : x86 meets FPGA whitepaper*. Kontron, 2012.
- [Kri99] C Mani Krishna. *Real-Time Systems*. Wiley Online Library, 1999.
- [KS15] Oliver Knodel and Rainer G Spallek. Rc3e : Provision and management of reconfigurable hardware accelerators in a cloud environment. *arXiv preprint arXiv :1508.06843*, 2015.
- [KZ09] Robert Kaiser and Dieter Zöbel. Quantitative analysis and systematic parametrization of a two-level real-time scheduler. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–8. IEEE, 2009.

- [Lan11] Travis Lanier. Exploring the design of the cortex-a15 processor. *URL : http://www.arm.com/files/pdf/atexploring_the_design_of_the_cortex-a15.pdf* (visited on 12/11/2013), 2011.
- [LB05] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2) :257–269, 2005.
- [LCC13] Chien-Te Liu, Kuan-Chung Chen, and Chung-Ho Chen. Casl hypervisor and its virtualization platform. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 1224–1227. IEEE, 2013.
- [Lie94] Jochen Liedtke. Improving ipc by kernel design. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 175–188. ACM, 1994.
- [Lie95] Jochen Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.
- [LKLJ09] Ming Liu, Wolfgang Kuehn, Zhonghai Lu, and Axel Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498–502. IEEE, 2009.
- [LP09] Enno Lübbers and Marco Platzner. Reconos : Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1) :8, 2009.
- [LXC⁺] Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh TX Phan, Chris Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. Realizing compositional scheduling through.
- [LZQ07] Lei Liu, Feng-li ZHANG, and Zhi-guang QIN. Embedded linux’s bootloader based on u-boot. *Application Research of Computers*, 12 :078, 2007.
- [MA01] Aloysius K Mok and Xiang Alex. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 129–138. IEEE, 2001.
- [MA09] P McLean and H Ayoub. ucos ii vs uclinux. Technical report, Technical Report, Computer Architecture Research Group, University of Ottawa. Ottawa, 2009.
- [McD08] Eric J McDonald. Runtime fpga partial reconfiguration. In *Aerospace Conference, 2008 IEEE*, pages 1–7. IEEE, 2008.
- [OBDA11] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.
- [PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7) :412–421, 1974.

- [PG11] François Philipp and Manfred Glesner. Mechanisms and architecture for the dynamic reconfiguration of an advanced wireless sensor node. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 396–398. IEEE, 2011.
- [PKR⁺13] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *Journal of Systems Architecture*, 59(3) :144–154, 2013.
- [POP⁺14] S Pinto, Daniel Oliveira, J Pereira, Nuno Cardoso, Mongkol Ekpanyapong, Jorge Cabral, and A Tavares. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–4. IEEE, 2014.
- [Res15] Transparency Market Research. Embedded system market - global industry analysis, size, share, growth, trends and forecast, 2015 - 2021. Technical report, Transparency Market Research, 2015.
- [Ros12] Daniel Rossier. Embeddedxen : A revisited architecture of the xen hypervisor to support arm-based embedded virtualization. *White paper, Switzerland*, 2012.
- [RVdlTR14] Alex Rodriguez, Juan Valverde, Eduardo de la Torre, and Teresa Riesgo. Dynamic management of multikernel multithread accelerators using dynamic partial reconfiguration. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–7. IEEE, 2014.
- [Seo10] Sangwon Seo. Research on system virtualization using xen hypervisor for arm based secure mobile phones. In *Seminar Security in Telecommunications, Berlin University of Technology, Korea Advanced Institute of Science and Technology*, 2010.
- [SK10] Udo Steinberg and Bernhard Kauer. Nova : a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.
- [SL03] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 2–13. IEEE, 2003.
- [SL04] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 57–67. IEEE, 2004.
- [Ste01] David B Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference (ESC)*, 2001.
- [TCL09] David B Thomas, J Coutinho, and Wayne Luk. Reconfigurable computing : Productivity and performance. In *Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on*, pages 685–689. IEEE, 2009.

- [Tec14] UBM Tech. 2014 embedded market study then now whats next. Technical report, UBM Tech, 2014.
- [uco12] *Para-virtualized $\mu C/OS-II$ RTOS on Xen-ARM*, 2012.
- [UNR⁺05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando Martins, Andrew V Anderson, Steven M Bennett, Alain Kägi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5) :48–56, 2005.
- [VF14] Kizheppatt Vipin and Suhaib A Fahmy. Zycap : Efficient partial reconfiguration management on the xilinx zynq. *Embedded Systems Letters, IEEE*, 6(3) :41–44, 2014.
- [VGS08] Jérôme Gallard-Adrien Lèbre-Geoffroy Vallée, Christine Morin-Pascal Gallard, and Stephen L Scott. Refinement proposal of the goldberg’s theory. 2008.
- [VH11] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011.
- [WBP13] Wei Wang, Miodrag Bolic, and Jonathan Parri. pvfpga : accessing an fpga-based hardware accelerator in a paravirtualized environment. In *Hardware/-Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–9. IEEE, 2013.
- [XBG⁺10] Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, and Attila Bilgic. Performance evaluation of para-virtualization on modern mobile phone platform. In *Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering*, 2010.
- [xen12] *Xen on ARM (PV)*, 2012.
- [XEN14] XEN. *Xen ARM with Virtualization Extensions whitepaper*, 2014.
- [Xil14a] Xilinx. *Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC : User Guide*. Xilinx, 2014.
- [Xil14b] Xilinx. *Vivado High Level Synthesis (UG902)*, 2014.
- [Xil14c] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, 2014.
- [XWLG11] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen : Towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48. IEEE, 2011.
- [YKP⁺11] Jungwoo Yang, Hyungseok Kim, Sangwon Park, Changki Hong, and Insik Shin. Implementation of compositional scheduling framework on virtualization. *ACM SIGBED Review*, 8(1) :30–37, 2011.
- [YY14] Seehwan Yoo and Chuck Yoo. Real-time scheduling for xen-arm virtual machines. *Mobile Computing, IEEE Transactions on*, 13(8) :1857–1867, 2014.

- [YYY13] Seehwan Yoo, Sung-bae Yoo, and Chuck Yoo. Virtualizing arm vfp (vector floating-point) with xen-arm. *Journal of Systems Architecture*, 59(10) :1266–1276, 2013.

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:

Étude des techniques de virtualisation pour des systèmes temps-réel et reconfigurables dynamiquement

Nom Prénom de l'auteur : XIA TIAN

Membres du jury :

- Monsieur GOGNIAT Guy
- Monsieur PREVOTET Jean-Christophe
- Monsieur VERDIER François
- Madame NOUVEL Fabienne
- Monsieur GROLLEAU Emmanuel
- Monsieur BECHENNEC Jean-Luc

Président du jury : *François VERDIER*

Date de la soutenance : 05 Juillet 2016

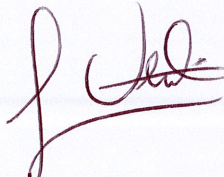
Reproduction de la these soutenue

Thèse pouvant être reproduite en l'état

~~Thèse pouvant être reproduite après corrections suggérées~~

Fait à Rennes, le 05 Juillet 2016

Signature du président de jury

François VERDIER


Le Directeur,

M'hamed DRISSI

Aujourd'hui, les systèmes embarqués jouent un rôle prépondérant dans la vie quotidienne des utilisateurs. Ces systèmes sont très hétérogènes et regroupent une énorme diversité de produits tels que les smartphones, les dispositifs de contrôle, les systèmes communicants, etc. Avec cette large gamme d'applications, ces systèmes ont évolué en différentes catégories. Il existe des systèmes avec une grande puissance de calcul. D'autres systèmes embarqués à faible coût, avec des ressources limitées, sont destinés à l'implantation de dispositifs simples, qui constituent le cœur de l'Internet of Objects (IdO). Fondamentalement, la plupart de ces appareils partagent des caractéristiques communes telles que la taille, le poids et la faible consommation d'énergie.

Tandis que la complexité des systèmes embarqués augmente, il devient de plus en plus coûteux d'améliorer les performances du processeur par des approches technologiques classiques i.e. diminution de la taille des transistors. Dans ce contexte, l'idée d'une architecture hétérogène CPU-FPGA est devenue une solution prometteuse pour les concepteurs de systèmes sur puce, en termes de rapidité de mise sur le marché. D'autre part, la forte capacité d'adaptation et le faible coût en font une solution très recherchée. Cette solution permet d'allier les avantages et la flexibilité d'un processeur aux architectures matérielles classiques. Elle permet également d'étendre les concepts classiques, telles que la virtualisation, aux circuits matériels.

Cette thèse décrit un micro-noyau original (Ker-ONE) permettant de gérer la virtualisation des systèmes embarqués et fournissant un environnement pour les machines virtuelles en temps réel. Nous avons simplifié l'architecture du micro-noyau en ne gardant que les caractéristiques essentielles requises pour la virtualisation, et massivement réduit la complexité de la conception du noyau. Sur la base de ce micro-noyau, nous avons mis en place un cadre capable de gérer des ressources reconfigurables dans un système composé de machines virtuelles. Les accélérateurs matériels reconfigurables sont mappés en tant que dispositifs classiques dans chaque machine. Grâce à une gestion efficace de la mémoire dédiée, nous avons permis de détecter automatiquement le besoin de ressources et permettons une allocation dynamique.

Selon diverses expériences et évaluations, nous avons montré que Ker-ONE ne dégrade que très peu les performances en termes de temps d'exécution. Les surcoûts engendrés peuvent généralement être ignorés dans les applications réelles. Nous avons également étudié l'ordonnancement temps réel dans les machines virtuelles. Les résultats montrent que le respect de l'échéance des tâches du RTOS est garanti. Nous avons également démontré que le noyau proposé est capable d'allouer des accélérateurs matériels très rapidement.

Nowadays, embedded systems are playing important roles in the daily life of most people, ranging from customer products such as smartphones and vehicles, to industry domains. With such an expanded range of purposes, embedded systems have evolved into different categories. There are systems with high computing power which can support complex software stack and enormous resources. There are also small-scaled embedded systems with limited resources and are intended for low-cost, simple devices, such as for Internet-of-Things (IoT). Basically, most of these devices share common characteristics such as requirements in size, weight and low power consumption.

While the complexity of embedded systems is increasing, it is becoming more and more expensive to improve CPU performance by conventional approaches, i.e. IC scaling and ASICs. In this context, the concept of heterogeneous CPU-FPGA architecture has become a promising solution for SoC device vendors, because of the fast time-to-market circle, the high adaptability and the relatively-low cost to improve the computation ability. This emerging convergence point of conventional CPU and FPGA computing makes it possible to extend traditional CPU visualization technologies into the FPGA domain to fully exploit the mainstream FPGA computing. To achieve this goal, it is necessary to propose an architecture which enhances the ability of existing technology while respecting the features of both software and hardware components.

This thesis describes an original micro-kernel that manages virtualization and that provides an execution environment for real-time virtual machines. We have simplified the micro-kernel architecture by only keeping critical features required for virtualization, and massively reduced the kernel design complexity. Based on this micro-kernel, we have introduced a framework capable of DPR resource management in a virtual machine system. DPR accelerators are mapped as ordinary devices in each VM. Through dedicated memory management, our framework automatically detects the request for DPR resources and allocates them dynamically.

According to various experiments and evaluations, we have shown that Ker-ONE causes very low virtualization overheads, which can generally be ignored in real applications. We have also studied the real-time schedulability in virtual machines. The results show that RTOS tasks are guaranteed to be scheduled while meeting their intra-VM timing constraints. We have also demonstrated that the proposed framework is capable of virtual machine DPR allocation with low overhead.