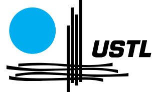

Équilibrage de charge pour systèmes temps-réel asymétriques sur multi-processeurs

Mémoire de DEA d'informatique
filière conception de systèmes embarqués

Eric Piel
Eric.Piel@lifl.fr

Responsables :
Jean-Luc Dekeyser
Philippe Marquet

Équipe WEST
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France

Juin 2004

Résumé

Le sujet entre dans le cadre du projet européen ITEA HYADES qui vise à promouvoir les machines SMP comme plate-forme pour des applications temps-réel de calcul intensif. Un ordonnancement asymétrique pour le temps-réel, nommé ARTiS, a été proposé. Des premières évaluations ont montré la faisabilité de la solution.

Le principe d'ARTiS est de distinguer deux types de processeurs, les processeurs pouvant exécuter toutes les tâches et les processeurs interdisant l'exécution de fonctions mettant en danger la qualité temps-réel du processeur. Lorsqu'une tâche tente d'exécuter une telle fonction, elle est automatiquement migrée. La force du modèle ARTiS est d'autoriser simultanément la réservation de ressources pour les applications temps-réel et l'équilibrage de la charge entre les processeurs temps-réel et les processeurs non temps-réel.

Le mécanisme d'équilibrage de charge original de Linux ne prend pas en compte cette asymétrie entre les processeurs. Nous avons étudié et explicité l'ensemble des migrations possibles entre processeurs. À partir de cette étude, il a été possible de spécifier des modifications au mécanisme original. En particulier, nous proposons :

- l'utilisation de queues à accès non bloquants couplée à une politique de déclenchement active ;
- une politique de désignation locale estimant les migrations futures des tâches ;
- un calcul de la charge des processeurs distinguant la charge des tâches temps-réel des autres tâches.

Enfin, l'implantation dans le noyau ARTiS est en cours et des protocoles de mesures ont été écrits afin de vérifier et d'estimer les améliorations apportées par cette implantation.

Mots clés : Équilibrage de charge, temps-réel, ordonnancement, SMP, multiprocesseur, Linux.

Abstract

The subject is part of the ITEA european project HYADES which attempts to promote SMP computers as platforms for HPC real-time applications. An asymmetric real-time scheduling, called ARTiS, has been proposed. The first evaluations have proven viability of the solution.

The principle of ARTiS is to distinguish two types of processors, the processors which can execute every kind of task and the processors prohibiting execution of real-time endangering functions. When a task attempts to execute such function, it will be automatically migrated. The power of the ARTiS model is to allow simultanously ressources reservation for real-time applications and load-balancing between real-time and non real-time processors.

The Linux original load-balancing mechanism is not aware of this asymmetry between the processors. We have studied and listed all the possible migrations between the processors. From this study, modifications to the orginal mechanism were specified. More specificaly, we propose:

- the use of lock-free queues associated to a "push"trigger policy,
- a local designation policy which can estimate the probability of the future migrations of the tasks,
- an evaluation of the processors load which distinguish between the real-time tasks and the others.

Finally, the implementation into the ARTiS kernel is in progress and the design of specific measurement tests were written in order to verify and estimate the enhancements provided by this implementation.

Keywords: Load-balancing, real-time, scheduling, SMP, multi-processor, Linux.

Remerciements

Tout d'abord je tiens à remercier mon responsable de stage Philippe Marquet qui m'a aidé face aux problèmes théoriques et administratifs et qui m'a introduit au métier de chercheur. Je suis également particulièrement reconnaissant envers Jean-Luc Dekeyser pour m'avoir permis d'effectuer mon stage de DEA dans son équipe.

Merci également aux membres du projet ARTiS, Alexandre, Daniel, Javed et tout particulièrement Julien pour leur soutien moral et technique.

Enfin je souhaite remercier toute le reste de l'équipe WEST, Abdelkader, Ashish, Arnaud, Cédric, Chadi, Lossan, Linda, Mickaël, Ouassila, Pierre, Philippe, Samy et Stéphane pour leur accueil, leurs précieuses aides et leur bonne humeur quotidienne.

Table des matières

Introduction **11**

1 Temps-réel et Linux **13**
1.1 Approches tradionnelles 13
1.1.1 Le système standard GNU/Linux 13
1.1.2 Approche co-noyau 14
1.1.3 Approche multi-processeur asymétrique 15
1.2 Le projet ARTiS 16
1.2.1 Le projet européen HYADES 16
1.2.2 Principe de fonctionnement d'ARTiS 17
1.2.3 Implantation d'ARTiS 19

2 Équilibrage de charge **25**
2.1 Principe de l'équilibrage de charge 25
2.1.1 Politique de mise à jour des informations 25
2.1.2 Politique de déclenchement 26
2.1.3 Politique de sélection 27
2.1.4 Politique de désignation locale 27
2.1.5 Politique d'appariement 27
2.1.6 Intégrations des politiques 28
2.1.7 Propriétés comportementales d'un équilibrage de charge 28
2.2 Équilibrage de charge dans Linux 29
2.2.1 Déclenchement de la détection 30
2.2.2 Détection du déséquilibre 30
2.2.3 Sélection des tâches à migrer 30
2.2.4 Mécanisme de migration 31
2.2.5 Calcul de la priorité d'ordonnancement des tâches 31
2.2.6 Comparaison avec l'implantation de FreeBSD 32
2.2.7 Conclusion 32

3 Équilibrage de charge adapté au temps-réel **33**
3.1 Les migrations dans ARTiS 33
3.2 L'équilibreur de charge 35

3.2.1 Pondération de la run-queue . . . 35
3.2.2 Élimination des verrous inter-processeurs . . . 37
3.2.3 Estimation de la prochaine tentative de migration . . . 38
3.2.4 Association du type de tâche au type de processeur . . . 39
3.3 Mise en œuvre . . . 40
3.3.1 Implantation . . . 40
3.3.2 Validation . . . 43

Conclusion **47**

Bibliographie **49**

A Extrait du code d'implantation des statistiques **51**

B Exemple de scénario lollobrigida **53**

C Load-balancing for a real-time system based on asymmetric multi-processing **57**

Introduction

Ce document présente le travail qui m'a été confié dans le cadre du projet ARTiS. Le projet vise à développer une extension au noyau Linux afin de pouvoir obtenir un système avec des qualités de temps-réel dur et de calcul haute-performance (HPC). La solution proposée est basée sur une asymétrie volontaire entre les processeurs d'une architecture SMP. Certains processeurs peuvent exécuter n'importe quel code tandis que d'autres n'autorisent que l'exécution de code ne remettant pas en cause la réactivité du processeur aux interruptions (c'est-à-dire sa qualité temps-réel). La valeur ajoutée d'ARTiS par rapport aux projets similaires est de permettre malgré cette asymétrie l'exécution de toutes les tâches sur tous les processeurs, autorisant ainsi l'utilisation maximale de la puissance de calcul de l'ordinateur.

La coexistence des deux propriétés d'ARTiS est assurée par un mécanisme qui migre automatiquement les tâches tentant d'exécuter du code non autorisé sur un processeur dédié au temps-réel vers un autre processeur. En plus de ce mécanisme, un second est nécessaire pour équilibrer la charge entre les processeurs de manière transparente, ce qui permet de tirer le maximum de puissance de la machine. Bien que le noyau Linux possède déjà un équilibreur de charge, il ne prend pas en compte l'organisation asymétrique de la machine ni les qualités de temps-réel à respecter. Le projet a donc consisté à proposer et implanter un nouvel équilibreur de charge qui tienne compte du contexte particulier d'ARTiS.

Après avoir détaillé dans une première partie l'état de l'art dans le domaine du temps-réel et le projet ARTiS, nous nous attellerons dans une deuxième partie à étudier l'état de l'art de l'équilibrage de charge. Une dernière partie décrira le modèle développé au cours du stage ainsi que l'implantation commencée et les méthodes de validation proposées.

Chapitre 1

Temps-réel et Linux

1.1 Approches tradionnelles

Alors qu'il n'y a encore pas très longtemps les systèmes temps-réel étaient pratiquement exclusivement développés à l'aide de systèmes dédiés, il est de plus en plus fréquent de voir des solutions à base du système libre GNU/Linux. Cela a été possible surtout grâce à l'augmentation de la puissance des machines (rendant possible les systèmes d'exploitation « à tout faire ») et l'adoption par le public de ce jeune système d'exploitation, attirant les utilisateurs temps-réel par ses nombreux avantages tels que la facilité de développement et la liberté d'extension du système. Commençons par un rapide tour d'horizon des différentes approches proposées pour rendre Linux capable d'assurer des contraintes temps-réel.

1.1.1 Le système standard GNU/Linux

Le système d'exploitation GNU/Linux est basé sur les outils GNU et le noyau Linux. Le noyau a été développé depuis le début avec le soucis de performance et de conformité aux standards (par exemple POSIX) pour en faire un système d'exploitation à usage général. Le support de plates-formes SMP est maintenant mâture et ses performances concernant les applications non temps-réel ne sont plus à démontrer. Malheureusement, pratiquement aucune attention n'a été portée aux contraintes liées au temps-réel. De plus lors de l'implantation, ces contraintes sont souvent incompatibles avec les performances.

Apparue seulement très récemment (dans la version 2.6 du noyau), une option disponible à la compilation permet la « préemptivité du noyau ». Proposée par MontaVista, vendeur d'un Linux pour systèmes embarqués, cette option [1] permet le réordonnancement d'une tâche à l'intérieur même du noyau. Par conséquent, lorsqu'une interruption matérielle est reçue il n'est plus nécessaire d'attendre que l'on ait quitté l'espace noyau pour changer de contexte et donner la main à la tâche associée à l'interruption. Pour pouvoir assurer l'atomicité nécessaire de certaines sections, la fonction

`preempt_disable()` permet d'inhiber localement la propriété de préemptivité (tandis que `preempt_enable()` la réactive).

La préemptivité du noyau améliore effectivement les latences utilisateur, le temps que met une tâche à être réordonnancer après qu'une interruption qui lui est associée soit déclenchée. Cependant les garanties restent dans l'optique de temps-réel mou : la très grande majorité des latences sont courtes mais il y en a parfois de plus longues. Ces garanties sont suffisantes pour les tâches de traitement multimédia qui étaient visées lors de l'intégration dans le noyau, mais le temps-réel dur ne peut être assuré.

Pour obtenir des garanties plus fortes plusieurs solutions impliquant le noyau Linux ont déjà été proposées. Il est possible de les catégoriser en deux approches différentes. Nous allons voir ces approches dans les deux parties suivantes.

1.1.2 Approche co-noyau

La première de ces approches se base sur la présence simultanée d'un second noyau spécialisé dans le traitement temps-réel. Les projets RTAI [2] et RTLinux [3, 4] proposent de telles solutions. De manière générale l'implantation de tels systèmes consiste en un petit noyau (souvent nommé micro-noyau) qui met à disposition les services temps-réel et qui ordonnance le noyau Linux comme une tâche de priorité faible, lorsqu'aucune tâche temps-réel n'est éligible. Le fonctionnement se fait par une virtualisation des interruptions qui ne sont jamais véritablement masquées. Cela permet de préempter le noyau Linux à n'importe quel instant. L'architecture est représentée par la figure 1.1.

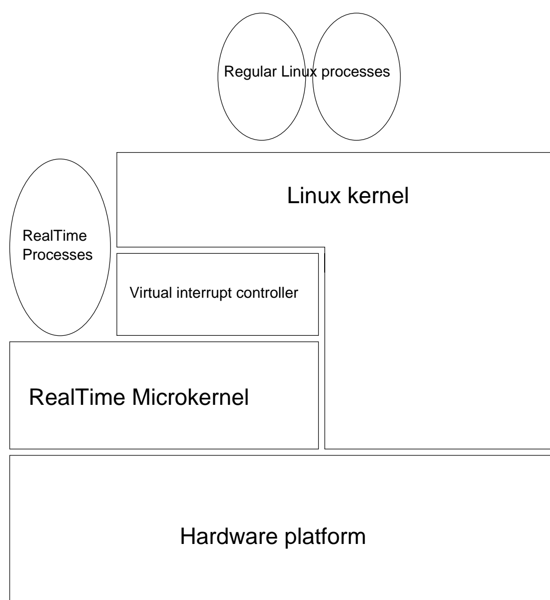

FIG. 1.1 – Architecture d'un système co-noyau.

Le micro-noyau est suffisamment compact et possède un nombre de chemins d'exécution suffisamment petit pour qu'il soit techniquement possible de prouver le temps

de réponse à une interruption comme étant borné. Cette preuve est impossible à obtenir sur le noyau Linux en raison de sa trop grande complexité. Les latences obtenues sur un tel système sont particulièrement bonnes, de l'ordre de la dizaine de microsecondes. L'inconvénient majeur cependant est que le modèle de programmation est *asymétrique*, les tâches temps-réel n'ont pas accès aux fonctions de Linux et doivent se restreindre à l'API limitée proposée par le micro-noyau. Inversement, les tâches Linux n'ont aucun moyen de bénéficier des garanties temps-réel. Des mécanismes plus ou moins aisés à utiliser ont été réalisés pour permettre la communication et l'interaction entre tâches Linux et tâches temps-réel du micro-noyau, mais aucun ne peut effacer les contraintes de programmation. Par exemple dans RTLinux le mécanisme qui permet ce type de communication est appelé LXRT, via une API spécifique il autorise une tâche à avoir un thread temps-réel et un thread dans l'espace Linux, ils peuvent alors communiquer à l'aide de la mémoire partagée.

La seconde approche pour procurer à Linux des propriétés temps-réel ne possède pas cet inconvénient, comme nous allons le voir par la suite.

1.1.3 Approche multi-processeur asymétrique

Cette approche exploite l'architecture SMP et introduit la notion de processeur « protégé », c'est-à-dire le principe de multi-processeur asymétrique. Sur une machine multi-processeur il existe alors deux types de processeurs : ceux spécialisés pour le temps-réel qui n'exécutent que des tâches temps-réel et ceux qui exécutent toutes les tâches non temps-réel. En plus, les processeurs temps-réel sont délestés des traitements d'interruption qui ne sont pas associées directement à une tâche temps-réel. Même si le mélange des termes SMP et asymétrique peut paraître antinomique, cette solution reste basée sur le modèle SMP de Linux et le concept d'asymétrie est introduit par dessus à l'aide de modifications très minimes. Des systèmes tels que CCC RedHawk Linux [5, 6] ou SGI REACT/pro pour IRIX [7] utilisent cette approche et peuvent annoncer des latences inférieures à la milliseconde.

Dans ce modèle le programmeur n'a à faire aucune distinction entre une tâche temps-réel et une tâche normale, tout se fait à la configuration du système. À l'aide de mesures nous avons vérifié les faibles latences d'interruption procurées par cette solution. Les résultats sont présentés dans la partie suivante. Cependant, contrairement à l'approche co-noyau, il est impossible de certifier un temps de réponse théorique maximum étant donné la complexité du noyau Linux sur lequel les tâches sont ordonnancées. En particulier, les tâches temps-réel se doivent de ne pas appeler des fonctions susceptibles de perturber la réactivité du processeur (comme par exemple l'écriture directe sur le disque dur). En plus, et c'est l'une des restrictions des plus importantes, comme seules les tâches temps-réel peuvent être exécutées sur les processeurs temps-réel, de la ressource CPU est gâchée dès qu'elles n'utilisent pas toute la puissance disponible du processeur.

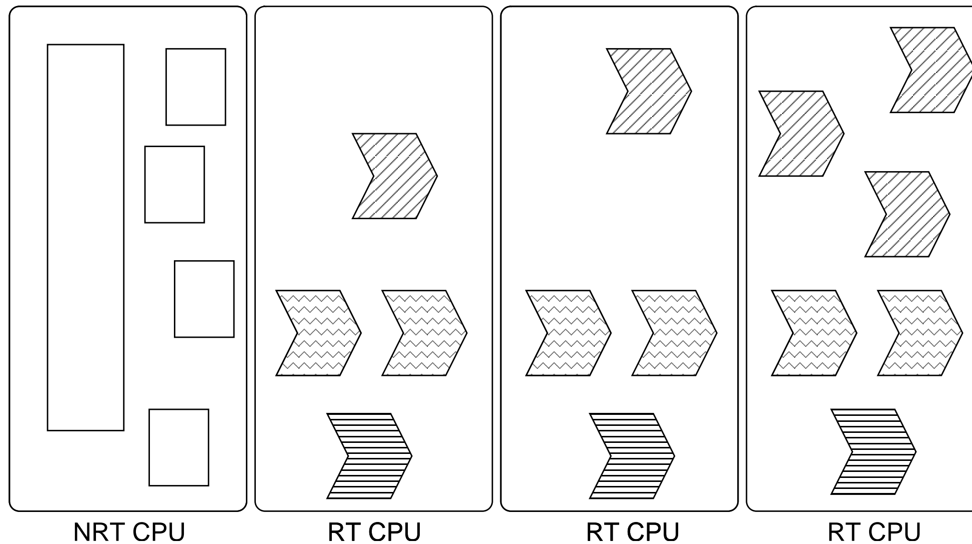

FIG. 1.2 – Schéma d'une architecture basée sur l'asymétrie entre les processeurs pour garantir le temps-réel.

1.2 Le projet ARTiS

ARTiS est une évolution de ce type d'approche mais étend le modèle afin de permettre le partage des ressources CPU entre tâches temps-réel et non temps-réel et ainsi bénéficier de la pleine puissance CPU des machines SMP. Voyons maintenant en détail cette approche.

1.2.1 Le projet européen HYADES

ARTiS est développé dans le cadre de HYADES, un projet européen ITEA (Information Technology for European Advancement) qui a pour objectif la mise au point d'un système multi-processeurs disposant d'un support pour le temps-réel s'appuyant sur une interface de programmation standard. Il a été décidé que les ordinateurs seraient basés sur des processeurs Itaniums (processeurs Intel 64 bits) principalement en raison de leur capacité industrielle (calcul flottant très efficace, support standard multi-processeurs, adressage de grandes zones de mémoire...). Il a aussi été spécifié que le système d'exploitation serait Linux (pour son support de la norme POSIX, pour son extrêmement modularité et pour sa licence Open-Source).

Le partenariat autour de ce projet s'est établi à la fois entre des fournisseurs de matériel et de solutions logicielles, et des candidats pour devenir utilisateurs des technologies. Les partenaires sont français et norvégiens. Entre autres, on peut citer comme partenaire Bull, Thalès Communications, Dolphin, MandrakeSoft et le LIFL.

Le LIFL travaille à fournir une solution pour rendre le noyau Linux capable de répondre à des contraintes de temps-réel dur tout en gardant la puissance disponible des

machines multi-processeurs. Le projet ARTiS est donc développé pour IA-64 mais afin d'intéresser un plus large public, il est fait en sorte qu'il fonctionne aussi sur x86. Pour donner un ordre de grandeur, le projet HYADES a définit comme contrainte des latences d'interruption toujours inférieures à 300 μs tandis que des mesures sur un noyau Linux récent (2.6.4) contenaient des latences allant jusqu'à 47ms.

Nous allons maintenant nous intéresser aux propositions techniques sur lesquelles ARTiS repose.

1.2.2 Principe de fonctionnement d'ARTiS

ARTiS est une extension temps-réel au noyau Linux qui vise les ordinateurs multi-processeurs (SMP) [8, 9]. Une des caractéristiques de cette extension est de pouvoir exécuter les tâches temps-réel dans l'espace utilisateur. Par conséquent le programmeur peut développer son application de manière usuelle, en se basant sur les API POSIX et/ou Linux. Ceci est d'autant plus commode lorsqu'il s'agit de réutiliser une application déjà existante. Dans ARTiS les tâches sont temps-réel dans le sens où elles sont identifiées par une priorité élevée et sont assurées de ne pas être perturbées par des tâches de priorité inférieure.

Gestion de l'architecture multi-processeurs L'architecture SMP de l'ordinateur permet d'accroître la puissance de calcul potentielle. Cependant, pour rendre cette puissance utilisable facilement le système d'exploitation doit fournir un ensemble de fonctionnalités particulières :
- le partage mémoire ;
- la migration de tâche ;
- l'équilibrage de charge entre processeurs ;
- la communication entre les tâches.

Ces services sont déjà implantés dans Linux, ce qui le rend donc attrayant comme base de départ. Par contre, il a l'inconvénient de ne pas être orienté vers le temps-réel, seules des propriétés de temps-réel mou sont proposées (via les ordonnancements FIFO et round-robin).

La solution ARTiS combine ces deux intérêts en introduisant la notion d'asymétrie à la plate-forme SMP Linux. ARTiS signifie Asymmetric Real-Time Scheduler (ordonnanceur temps-réel asymétrique). Tout en gardant les services de Linux standard et la gestion multi-processeur, les propriétés temps-réel du système d'exploitation sont améliorées. La solution se base sur la distinction forte entre processeurs temps-réel et processeurs non temps-réel ainsi que sur la migration des tâches qui tentent de différer la réponse aux interruptions sur un processeur temps-réel. Les mécanismes suivants sont utilisés :
- **Le partitionnement des CPU en sous-ensembles** : un ensemble CPU NRT (Non Real-Time) et un ensemble CPU RT (Real-Time). Chacun a sa politique d'ordonnancement propre. Le but est d'assurer la meilleure latence d'interruption à des

tâches désignées s'exécutant sur l'ensemble des CPU RT ;

- **Deux types de tâches temps-réel** Ce sont des tâches temps-réel Linux normales. Elles se différencient entre elles par leur affinité aux CPU RT (cf figure 1.3 :
  - Chaque CPU RT a une tâche temps-réel liée, appelée RT0 (tâche de priorité maximale). Ces tâches ont la garantie de disposer du CPU RT avec la priorité maximale. Seules ces tâches sont autorisées à entrer dans des sections bloquant les interruptions sur les CPU RT. C'est cette propriété qui assure une latence aussi faible que possible aux tâches RT0. Ces tâches sont les tâches temps-réel dur d'ARTiS ;
  - Les CPU RT peuvent exécuter d'autres tâches temps-réel mais uniquement si elles ne bloquent pas les interruptions. Ces tâches sont appelées RT1+ (tâches avec priorité 1 et plus). Elles peuvent utiliser le temps processeur et bénéficier de latences d'interruption faibles tant que les tâches RT0 ne sont pas en cours d'exécution. Pour assurer de faibles latences à ces dernières, les tâches RT1+ sont migrées par l'ordonnanceur ARTiS dès qu'elles tentent de bloquer les interruptions (un appel à `preempt_disable()` ou `local_irq_disable()`). Ce sont les tâches temps-réel firme d'ARTiS ;
  - Les tâches non temps-réel sont nommées NRT, elles n'ont aucune contrainte temps-réel. Elles coexistent avec les tâches temps-réel tant que celles n'ont pas besoin du processeur. Comme les tâche RT1+, elles migrent automatiquement lors d'une tentative de blocage les interruptions ;
  - Les CPU NRT exécutent principalement les tâches NRT. Ils prennent aussi en charge l'exécution des tâches RT1+ lorsqu'elles ont été migrées. Pour garder la charge processeur équilibrée, toutes ces tâches peuvent migrer vers des CPU RT mais sous la condition qu'elles ne bloquent pas les interruptions.
- **Un mécanisme de migration particulier** La migration a pour but d'assurer des latences faibles aux tâches RT0. Les tâches RT1+ et NRT sont automatiquement migrées vers un CPU NRT lorsqu'elles tentent de bloquer les interruptions. Un des changements principal à apporter au mécanisme déjà existant est la suppression de verrous inter-CPU. Ainsi, les tâches sont migrées en transitant par des files non bloquantes qui relient les CPU RT aux CPU NRT [10]. Ces FIFO sont asymétriques et ne possèdent qu'un lecteur et qu'un écrivain. Grâce à ce mécanisme, l'ordonnanceur ARTiS est capable de migrer les tâches sans attente ;
- **Un mécanisme de communication adapté** Sur les machines SMP, les tâches échangent des données via un système de lecture/écriture sur la mémoire partagée. Pour assurer la cohérence des données, il y a besoin de sections critiques. Les sections critiques sont protégées des accès concourants à l'aide de mécanismes de verrouillage. Malheureusement ce type d'implantation, voir de communication, ne convient pas à des tâches temps-réel s'exécutant sur ARTiS : si une tâche RT0 communique avec une tâche RT1+ alors cette dernière migrera forcément à chaque émission/réception de message. Par conséquent il faut développer un modèle particulier de communication qui soit sans verrou, en se limitant éventuellement au contexte un lecteur/un écrivain.

– **Une politique d'équilibrage de charge efficace** C'est l'objet principal de ce stage de DEA, elle doit permettre de tirer toute la puissance d'une machine SMP malgré les contraintes supplémentaires imposées par ARTiS. Habituellement il s'agit juste de déplacer quelques tâches en cours d'exécution sur un processeur chargé vers un moins chargé. La tâche est ici plus délicate puisqu'il faut jouer aussi des spécificités des types de tâches RT. Les tâches RT0 ne migrent pas, par définition. Par contre, les tâches RT1+ doivent avoir une affinité plus grande que les tâches NRT envers les CPU RT puisqu'ils proposent de meilleures latences. Nous reviendrons par la suite en détail sur le modèle et son implantation.

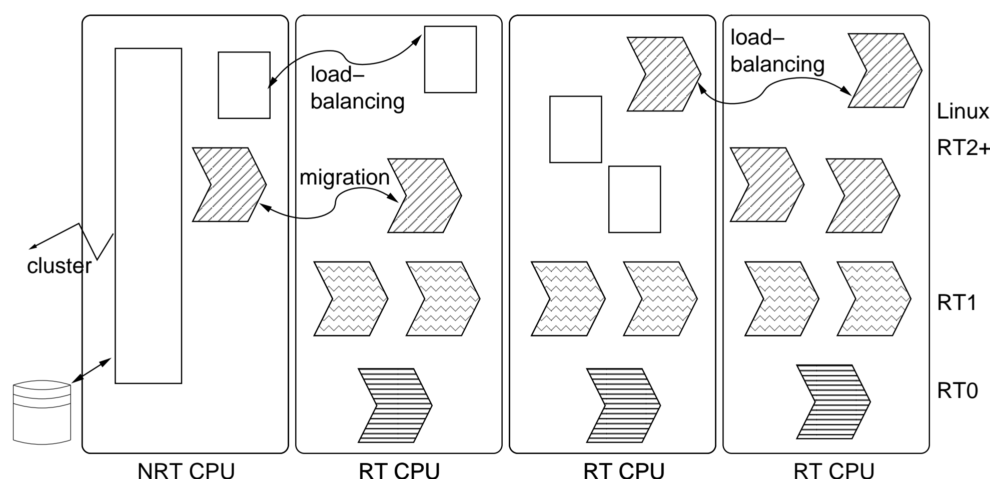

FIG. 1.3 – Schéma de l'architecture d'ARTiS. Association dynamique de tâches à des processeurs RT ou NRT.

ARTiS supporte donc trois niveaux différents de traitement temps-réel : RT0, RT1+ et NRT. Les tâches RT0 sont assurées d'un temps de latence minimal tout en restant des tâches de l'espace utilisateur Linux. Les tâches RT1+ sont temps-réel firmes mais peuvent profiter de la souplesse et des avantages de l'architecture SMP, ce qui est particulièrement intéressant dans le cadre du calcul intensif. Finalement, les tâches NRT peuvent utiliser le reste des ressources de la plate-forme SMP sans mettre en danger la réactivité des processeurs RT.

1.2.3 Implantation d'ARTiS

Actuellement l'implantation de l'ordonnanceur ARTiS est en cours de finalisation. Il reste encore des problèmes techniques mineurs mais la mise en œuvre générale ne devrait plus changer. Même si le code reste relativement concis, la mise au point est particulièrement difficile et sensible aux erreurs dues au fait que les modifications touchent à l'ordonnanceur du noyau.

Techniques d'implantation

Pour assurer que les tâches RT1+ et NRT n'augmentent pas les latences d'interruption sur les CPU RT, il faut assurer non seulement que les interruptions ne soient jamais masquées mais aussi que le noyau soit toujours préemptible. Cette seconde propriété a été introduite récemment dans le noyau (version 2.6) et permet de réordonnancer une tâche même lorsque l'exécution est située dans l'espace noyau (auparavant il fallait attendre le retour à l'espace utilisateur). Seules deux fonctions permettent d'entrer dans un tel état, respectivement `local_irq_disable()` et `preempt_disable()`.

ARTiS modifie donc les deux fonctions citées en introduisant au début un appel à `artis_try_to_migrate()`. Cette fonction vérifie dans quelles conditions l'appel a été effectué et, éventuellement, la tâche est migrée grâce à la fonction `artis_request_for_migration()`. Étant donné la fréquence des appels, les conditions doivent être rapides à détecter. Elles comprennent entre autres la vérification que la tâche n'est pas RT0, que le CPU est RT ou encore que l'on ne se trouve pas dans une partie de code interdisant la migration.

La migration d'une tâche s'effectue en plusieurs étapes. Tout d'abord, comme la tâche est en cours d'exécution il faut la désactiver. En effet, on ne peut pas migrer directement la tâche courante car sinon il se pourrait que plusieurs processeurs se mettent à l'exécuter en même temps. La désactivation consiste à supprimer la tâche de la queue d'exécution (run-queue). En conséquence, une autre tâche sera ordonnancée, c'est elle qui s'occupe de placer la tâche à migrer dans la file spéciale associée au couple de CPU RT/NRT choisi. Le CPU RT finit son travail en envoyant une IPI (Inter-Processor Interrupt) au CPU NRT pour lui indiquer qu'une nouvelle tâche doit être retirée de la file. La dernière étape est accomplie par le CPU NRT qui lit la file et place la tâche migrée dans sa run-queue. La tâche sera plus ou moins rapidement réordonnancée en fonction de sa priorité.

Évaluation de performances

Dans le but de valider le travail effectué sur ARTiS nous avons pratiqué des mesures de latence d'interruption. Différents noyaux Linux (tous basés sur la version 2.6.4) avec et sans charge furent soumis aux tests, tous sur la même machine, un quadri-processeur IA-64 1,3GHz. A l'aide d'un registre particulier du processeur il est possible de spécifier avec une précision de l'ordre du cycle l'instant de déclenchement d'une interruption. En général les latences sont très faibles et même sous une charge très élevée seule une partie infime des interruptions subit de longues latences. Chaque expérience comportait 300 millions de mesures (correspondant à 8 heures de test) afin d'obtenir avec plus de probabilité des temps proches du pire cas.

Méthode de mesure Les expériences que nous avons mis en place consistaient à mesurer la latence entre le déclenchement d'une interruption matériel et l'exécution du code utilisateur concernant cette interruption précise. En générant les interruptions à l'aide

d'un registre de débogage du processeur, la précision obtenue sur le temps de réaction à une interruption est précise au cycle près (inférieur à la nanoseconde). Le fonctionnement est décrit par le chronogramme de la figure 1.4. L'interruption est armée au temps t'_0, elle est déclenchée au temps t_0 (déduit de t'_0), le gestionnaire d'interruption est appelé au temps t_1 et la tâche temps-réel au temps t_3. Le temps t_2 est le temps auquel le gestionnaire d'interruption se termine, il n'apporte pas d'information particulièrement pertinente vis-à-vis des résultats. Par la suite nous l'omettrons donc de la description des résultats.

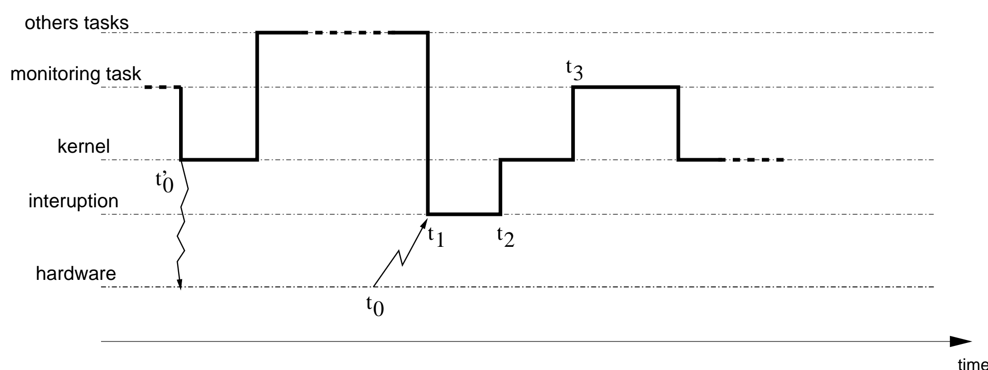

FIG. 1.4 – Chronogramme des tâches impliquées dans le code de mesure.

Charge système La charge système est utilisée afin de simuler un environnement aussi extrême que possible en ce qui concerne la réactivité des CPU. Nous avons composé la charge de quatre éléments différents : une charge de calcul (sur chaque processeur), une charge de lecture/écriture sur le disque (génère des interruptions), une charge réseau (la carte réseau génère des interruptions) et une charge de verrous (via un appel système qui prend un verrou dans le noyau).

Types de latences Deux types de latences d'interruption furent mesurées. Les latences noyau correspondent au temps nécessaire pour que le gestionnaire d'interruption spécifique soit appelé suite au déclenchement de l'interruption ($t_1 - t_0$). Un pilote de périphérique temps-réel subirait les même latences. Les latences utilisateur correspondent au temps requis pour transmettre le signal jusqu'à une tâche temps-réel en attente ($t_3 - t_0$). Ce sont les latences que subirait une tâche temps-réel ayant la priorité maximale, les tâches RT0 dans le modèle de programmation ARTiS.

Description des configurations mesurées Les cinq configurations présentées dans le tableau 1.1 ci-dessous sont les plus représentatives parmi celles testées. D'abord, le noyau Linux standard sans charge, c'est un noyau « vanilla » sur lequel seul le programme de mesure est exécuté. Ensuite, le même noyau est testé avec la charge système décrite précédemment. Puis c'est un noyau compilé avec l'option de préemptivité qui

est testé. Cette option est une des bases sur lesquelles ARTiS repose, elle permet de réordonnancer une tâche même si le code courant appartient au noyau. Cette option à pour optique le temps-réel mou. ARTiS « idéal » est une simulation d'un noyau ARTiS, il se base sur le noyau précédent et présente une asymétrie statique entre les processeurs forcée manuellement. C'est l'approche asymétrique décrite dans la partie précédente. Enfin, l'implantation actuelle d'ARTiS (sans l'équilibrage de charge spécifique) a aussi été mesurée.

Signification des mesures Pour chaque type de latence (noyau et utilisateur), deux colonnes sont associées. La colonne *Maximum* indique la latence maximale mesurée au cours des 8 heures. La colonne *99,999%* indique la latence maximale des 99,999% meilleures mesures, ce qui correspond dans notre cas à ne pas compter les 3000 pires cas. Pour un système temps-réel dur cette mesure n'a pas vraiment d'intérêt car une seule latence d'interruption plus longue que la limite mettrait en péril le système. Par contre la mesure est intéressante pour comparer des systèmes temps-réel mou, car elle donne une estimation de la latence pour la « majorité des cas ».

TAB. 1.1 – Latences noyau et latences utilisateur sur différentes configurations

	Noyau		**Utilisateur**	
Configurations	99,999%	Maximum	99,999%	Maximum
Linux standard sans charge	78μs	94μs	82μs	220μs
Linux standard avec charge	77μs	101μs	2,82ms	42ms
Linux avec préemption et charge	76μs	101μs	457μs	47ms
ARTiS « idéal » avec charge	3μs	9μs	8μs	28μs
ARTiS avec charge	72μs	101μs	91μs	120μs

Interprétation des résultats La configuration sans charge apporte peu d'information en elle même, elle n'a été effectuée que dans le but de pouvoir vérifier que la charge avait vraiment un impact sur les latences. Ceci apparaît nettement aux vues des latences utilisateur qui passent 220μs à 42ms.

Par définition, l'option de préemption du noyau ne peut améliorer que les latences utilisateur. Cependant, lorsque l'on compare les configurations avec et sans, cette amélioration n'est pas perceptible au niveau des maxima (qui restent dans le même ordre de grandeur). Par contre, comme le montre la colonne 99,999% il y a nettement moins de latences élevées, ce qui confirme l'intérêt de cette option pour le temps-réel mou.

Le noyau ARTiS « idéal » est prometteur en réduisant grandement aussi bien les latences utilisateur (les plus importantes dans le modèle ARTiS) qui passent en dessous de 30μs que les latences noyau. Le noyau ARTiS réel a des latences plus élevées, de l'ordre de 120μs, mais dans le cadre du projet HYADES qui requière des latences inférieures à 300μs, celles-ci permettent quand même de certifier le système comme temps-réel dur.

Les écarts avec la version idéale s'expliquent par le coût supplémentaire requit pour gérer dynamiquement l'asymétrie (détection des cas bloquant les interruptions et migration des tâches) et aussi par l'équilibrage de charge original encore présent sur les CPU NRT (qui prend des verrous inter-processeurs).

L'implantation actuelle respecte donc déjà les critères de temps-réel souhaités, cependant aucun mécanisme n'est présent pour assurer l'utilisation maximale de la puissance de la machine : l'équilibrage de charge original sur les CPU RT a été désactivé pour éviter les verrous inter-processeurs. Il faut donc mettre en place un équilibrage de charge qui intègre les concepts introduits par ARTiS dans son mécanisme.

Chapitre 2

Équilibrage de charge

Avant d'étudier plus en détail le cœur du travail effectué au cours du stage nous allons parcourir l'état actuel des connaissances sur l'équilibrage de charge ainsi l'implantation dans le système d'exploitation Linux et brièvement dans FreeBSD.

2.1 Principe de l'équilibrage de charge

Le but de toute technique d'équilibrage de charge (load-balancing) est d'optimiser l'utilisation des processeurs en spécifiant la localité des tâches. Autrement dit, l'équilibrage de charge doit minimiser le temps d'exécution d'un ensemble de tâche donné, ce qui revient souvent à maintenir une charge équivalente sur l'ensemble des processeurs. En général ce mécanisme est aussi utilisé pour gérer l'équité de temps d'exécution entre les tâches.

De manière plus précise, les algorithmes d'équilibrage de charge peuvent se distinguer par cinq caractéristiques majeures, comme décrit dans la thèse de Cyril Fonlupt [11] :

2.1.1 Politique de mise à jour des informations

La politique d'informations tente d'obtenir un état ou une carte du système en faisant transiter ou en collectant des informations sur l'ensemble ou une partie des nœuds. Il faudra alors déterminer quel type d'information est utile, quand sont elles nécessaires et quels sont les processeurs qui en sont demandeur/fournisseur.

Mesure de la charge L'état de charge des différents processeurs est la principale source d'informations pour les techniques d'équilibrage. Ce qui nous amène à expliciter la notion de charge d'un processeur. Le but premier de la mesure de la charge est d'estimer la quantité de traitement attribué à un CPU. Aucune solution universelle existe pour représenter parfaitement cette information. Une technique courante consiste

à se servir du nombre de processus en attente d'exécution. Cela a l'avantage d'être rapide à calculer. Il est tout à fait possible de se servir d'autres critères, voir même une combinaison linéaire de plusieurs. Pour éviter les fluctuations brusques des valeurs il arrive qu'elles soient lissées sur le temps.

Déclenchement de la transmition d'informations Trois classes de politiques peuvent être distinguées. **Les politiques à la demande**, dans ce cas les informations sont assemblées et envoyées à chaque fois qu'un processeur a besoin des informations. **Les politiques périodiques**, les informations sont récoltées de manière régulière et elles sont stockées soit de manière centralisée ou distribuées sur un ensemble de processeurs. Enfin **les politiques à changement d'état**, les fournisseurs décident lorsqu'il faut transmettre une mise à jour des informations parce qu'ils passent d'un état remarquable à un autre (inactif ou surchargé par exemple).

2.1.2 Politique de déclenchement

Afin d'éviter le gaspillage des ressources du système, la politique de déclenchement détermine le moment optimal d'équilibrage à partir des informations fournies par la politique précédente. Un équilibrage trop tardif provoque une augmentation de l'inactivité des processeurs, tandis qu'un déclenchement trop précoce entraîne des surcoûts importants dus à des communications et des redistributions de données inutiles.

Deux types de déclencheurs peuvent être envisagés, des mécanismes décentralisés dont la responsabilité est distribuée sur l'ensemble du système et des mécanismes centralisés où l'opportunité de déclenchement est décidée par une seule entité. Les politiques de déclenchement décentralisées sont réservées aux machines asynchrones qui permettent une certaine autonomie des nœuds. Dans ce cas le processus de décision obéit en général à une technique à base de seuils (charge très haute ou très faible par exemple). Les politiques de déclenchement centralisées impliquent en règle générale des déclenchement à des fréquences fixes, ce qui fonctionne bien sur des ensemble de processeurs synchronisés.

Lorsque le déclenchement est décentralisé on peut encore distinguer les politiques en fonction du type du demandeur. Si ce sont les nœuds chargés qui déclenchent l'équilibrage afin de pouvoir « pousser » certaines tâches vers des nœuds moins chargés alors c'est une stratégie dite **active**. Si, au contraire, ce sont les nœuds ayant une faible charge de travail qui tentent de « tirer » des tâches à partir de pairs plus chargés alors la stratégie est dite **passive**. Comme nous le verrons par la suite, Linux utilise une telle stratégie. Chacune de ces stratégies a ses avantages et ses inconvénients concernant l'opportunité du déclenchement (ou du non déclenchement). Pour tenter de réunir le meilleur des deux il existe parfois des stratégies **mixtes** où à la fois les processeurs chargés et ceux non chargés déclenchent l'équilibrage (une telle stratégie est employée dans FreeBSD).

2.1.3 Politique de sélection

La politique de sélection détermine les différents nœuds déséquilibrés qui joueront le rôle d'émetteur ou de receveur. La politique de sélection détermine, si la participation d'un site à l'équilibrage est ou non propice. Dans l'affirmative, elle détermine quel rôle jouera chaque site ; dans la plupart des méthodes existantes, on distingue deux cas :

- les nœuds sous-chargés qui joueront le rôle de récepteur ;
- les nœuds avec un excédant de charge qui joueront le rôle d'émetteur.

Il existe trois classes de politique de sélection. De nombreuses stratégies de sélection sont fondées sur une politique **à base de seuils** où un nœud compare sa charge locale avec un ou plusieurs seuils. Suivant le résultat de cette comparaison, le nœud deviendra émetteur ou récepteur de charge. Des stratégies **de comparaisons** peuvent également être employées, où le processeur adopte un comportement en fonction de l'état de l'ensemble ou d'un sous-ensemble de nœuds. Certaine méthodes dites **systématiques** ne tiennent pas compte de l'état actuel des nœuds. Ces techniques sont basées sur des résultats théoriques. Elles provoquent des échanges de travail dans un ordre donné et désignent alternativement les nœuds comme émetteurs, receveurs ou inactifs.

2.1.4 Politique de désignation locale

La politique de désignation locale identifie les processus qui vont être transférés. Cette étape est réalisée immédiatement après la politique de sélection lorsque le nœud est désigné comme émetteur. Par contre, elle s'exécute après la politique d'appariement si le nœud est receveur. Lorsque le système est capable de déplacer un processus déjà en cours d'exécution, c'est un système préemptif. Un tel système est plus souple mais il est plus difficile à implanter. Dans ce sens, Linux est un système préemptif.

Il y a deux types de méthode pour sélectionner les tâches. Les méthodes systématiques considèrent toutes les tâches identiques et sélectionnent de manière très simple celle à déplacer, par exemple par FIFO ou par tirage aléatoire. L'avantage est que le processus de sélection est pratiquement immédiat et donc le coût de l'équilibrage est faible. Le deuxième type de méthode consiste à filtrer les tâches à migrer. Ceci peut être fait manuellement par l'utilisateur ou en ordonnant les tâches en fonction de leur propriété (utilisation du CPU, affinité à un nœud, âge du processus...).

2.1.5 Politique d'appariement

Il s'agit de trouver un ou plusieurs partenaires au processeur qui a été choisi par la politique de sélection. Cette partie est à juste raison considérée comme étant le cœur d'une méthode d'équilibrage. Le partenaire peut être recherché sur un domaine limité spatialement à un voisinage de processeurs ou sur l'ensemble du système. La politique d'appariement est bien évidemment étroitement liée à la politique d'informations ; il est par exemple beaucoup plus aisé d'introduire une politique d'appariement centralisée si la politique d'information est également centralisée.

Les politiques d'appariement sont extrêmement nombreuses. En plus des techniques aléatoires (choix du nœud au hasard), il y a des politiques centralisées, d'autres distribuées et des politiques mixtes.

Une stratégie centralisée désigne un nœud comme serveur, il détient des informations qui seront utilisées pour la recherche de nœuds destinataires. Cette stratégie est dite « aveugle » lorsque le serveur choisit le processeur de destination en se basant uniquement sur ses actions passées (aucune information du système n'est nécessaire). Une stratégie asynchrone signifie que les processeurs recherchant des tâches à prendre ou à transmettre demandent au serveur de désigner pour eux un processeur adapté à la requête.

Les stratégies d'appariement distribuées sont caractérisées par une répartition du choix du site destinataire. Les stratégies distribuées permettent d'éviter l'inconvénient majeur des stratégies centralisées, à savoir la formation d'un goulot d'étranglement lorsque la taille du système devient importante. La recherche d'un site destinataire peut s'effectuer en aveugle sans connaissance de l'état des autres nœuds. Elle peut aussi être faite sous forme de sondages en interrogeant un certain nombre de nœuds pour choisir le plus adapté. Il est aussi possible d'effectuer une recherche exhaustive ou de lancer des enchères.

2.1.6 Intégrations des politiques

Ainsi l'ensemble des cinq types de politiques choisies permet de définir un équilibrage de charge. La combinaison de ces politiques est cruciale mais non aisée. En particulier comme les politiques ont de nombreuses interactions entre elles, le choix d'une stratégie à une étape donnée amène des contraintes sur le choix des politiques des autres étapes. Comme nous l'avons vu, quelque soient les techniques employées, l'équilibrage dynamique impose en permanence un surcoût de travail qui doit être estimé en regard des améliorations qu'il peut apporter. Un bon mécanisme d'équilibrage de charge doit donc être adapté au système qu'il contrôle.

2.1.7 Propriétés comportementales d'un équilibrage de charge

Le comportement est défini par la manière qu'un équilibrage de charge améliore l'exécution des tâches. Les seuls choix qu'il puisse prendre sont de déplacer une tâche vers tel ou tel autre processeur ou ne pas la déplacer du tout. Il y a trois principales propriétés qui permettent de distinguer la qualité de différents équilibrages de charge. Ce sont donc ce que l'on cherche à améliorer lorsque l'on modifie un équilibreur de charge.

Efficacité La qualité principale d'un équilibreur de charge doit être de pouvoir utiliser la puissance de la machine à son maximum. Cela implique que les processeurs doivent passer le minimum de temps oisif (lorsqu'il y a du travail de disponible). En général il

est possible de vérifier l'efficacité d'un équilibrage de charge en observant directement la proportion de temps passé oisif pour chaque processeurs.

Équité À un instant donné, les tâches ayant les mêmes priorités doivent se voir attribuer le même temps d'exécution. En corollaire, le temps accordé doit être fonction croissante de la priorité. Sur un processeur donné c'est l'ordonnanceur qui se charge d'assurer l'équité entre les tâches. Cependant il n'a pas les moyens d'ajuster le temps octroyé à chaque tâche en fonction de ce qui se passe sur les autres processeurs. C'est à l'équilibreur de charge d'assurer l'équité inter-processeur. L'observation de cette propriété peut consister à comparer les temps d'exécution de deux tâches avec la même priorité et le même travail, si l'équité est bonne ils sont toujours très proches l'un de l'autre.

Localité Les tâches qui ont une affinité particulière pour un ou plusieurs processeurs doivent y être affectées en priorité. Typiquement sur les ordinateur SMP classiques cette localité peut être temporaire parce que la tâche a des données dans le cache mémoire. Sur les systèmes NUMA (Architecture à mémoire non uniforme) une tâche peut s'exécuter plus rapidement sur un groupe de processeurs car les temps d'accès mémoire y sont plus courts. Dans ARTiS il y a aussi une affinité spécifique entre les tâches RT1+ et les CPU RT, parce que les temps de latence y sont meilleurs. Cette propriété de l'équilibrage de charge implique en général une réduction du nombre de migration et aussi, pour une tâche donnée, un ratio de temps d'exécution supérieur sur certains processeurs que sur d'autres.

Après avoir eu une vue d'ensemble de l'équilibrage de charge nous allons nous intéresser de près à l'implantation qui servira de base au travail du stage.

2.2 Équilibrage de charge dans Linux

Dans la dernière version de Linux un nouvel ordonnanceur nommé O(1) (d'après sa complexité) a été intégré. Une des nouveautés réside dans le fait que chaque processeur d'une machine SMP possède sa propre run-queue (file d'exécution) indépendante des autres processeurs. Afin de fournir le meilleur usage de l'ensemble des CPU, il est nécessaire d'avoir un mécanisme explicite qui s'assure que chaque processeur reçoit approximativement la même charge de travail. C'est l'équilibrage de charge.

L'idée générale est de détecter les cas déséquilibrés et alors déplacer des tâches d'un processeur à un autre pour rééquilibrer la charge. Dans le noyau, l'implantation d'un tel mécanisme doit donc répondre aux questions de savoir quand procéder à la détection, comment détecter le déséquilibre, comment choisir les tâches à migrer et aussi comment migrer une tâche entre deux run-queues [12]. Nous allons décrire les solutions mises en place dans Linux 2.6 (précisément, les versions antérieures à 2.6.7).

2.2.1 Déclenchement de la détection

La détection du déséquilibre est fait par la fonction `load_balance()`. Tout d'abord, elle est appelée à chaque fois que la run-queue de l'ordonnanceur est vide, c'est-à-dire qu'il n'y a aucune tâche prête à être exécutée sur le processeur. Comme de toute façon le processeur n'aurait rien à faire, cela ne coûte rien et permet éventuellement de récupérer tout de suite du travail à faire. En plus cette fonction est appelée à chaque tic d'horloge, c'est-à-dire à chaque milliseconde, lorsque le processeur est oisif (qu'il n'exécute aucune tâche) et toute les 200ms autrement. C'est la fonction `rebalance_tick()` qui s'occupe de lancer la détection périodiquement.

2.2.2 Détection du déséquilibre

L'état de déséquilibre de charge du système est détecté au début de `load_balance()` avec l'appel à la fonction `find_busiest_queue()`. Cette fonction commence par rechercher le processeur ayant la plus longue run-queue parmi tous les processeurs (à l'exception du processeur courant). Pour éviter les problèmes de fluctuation de la longueur de la run-queue, c'est le minimum de la taille courante et de la taille précédente (sauvegardé localement) qui est utilisé. Pour que l'état de déséquilibre soit décrété, la plus grande longueur doit être supérieure d'au moins 25% par rapport à la longueur de la run-queue locale.

Déséquilibre détecté Au cas où un déséquilibre aurait été détecté, un verrou est prit sur la run-queue la plus longue (la run-queue locale a déjà été verrouillée) et un test est effectué pour vérifier que le déséquilibre est toujours présent. Dans l'affirmative, une référence vers la run-queue la plus longue ainsi que la différence entre sa longueur et la longueur de la run-queue locale sont retournées.

2.2.3 Sélection des tâches à migrer

Nombre de tâches à déplacer Lorsque le CPU est oisif, une seule tâche est choisie et migrée. Autrement, le nombre de tâches élues est tel qu'une fois déplacées les deux run-queue soient de même longueur : c'est la différence des deux longueurs (déjà calculée) divisée par deux. Lorsque le CPU est oisif l'équilibrage s'effectue 200 fois plus souvent que lorsqu'il est chargé, par conséquent ce n'est pas un problème de ne déplacer qu'une seule tâche à la fois. Cette implantation permet une granularité plus fine de la répartition des tâches sur les machines avec plus de deux processeurs.

Ordre de tri des tâches Les tâches sont déplacées dans l'ordre suivant : d'abord les tâches qui sont dans la partie « expired » de la run-queue (c'est-à-dire qu'elle viennent d'être exécutées) puis ce seront les tâches sur le point d'être exécutées. Dans ces sous-ensembles les tâches choisies en premier sont celles avec la priorité la plus élevée, cela

permet de fournir plus de puissance aux tâches les plus importantes. Finalement, juste avant de réellement migrer la tâche, l'équilibreur de charge vérifie que la migration est possible grâce à la fonction `can_migrate_task()` qui teste trois propriétés : que la tâche n'est pas actuellement en cours d'exécution, qu'elle n'a pas été exécutée trop récemment (car alors il est probable qu'elle ait encore des données en mémoire cache et donc son attachement est encore fort) et que l'utilisateur n'ait pas interdit à la tâche d'aller sur le processeur destination.

Notons que grâce à la structure des données sur lesquelles l'ordonnanceur et l'équilibreur travaillent, les mécanismes de sélection du processeur et des tâches ont une complexité en O(1), le temps de traitement est indépendant du nombre de tâches sur les processeurs. Bien sûr le mécanisme de migration proprement dit a une complexité en O(n), sa durée dépend linéairement de la différence de longueur entre les deux run-queues.

2.2.4 Mécanisme de migration

Il existe deux mécanismes de migration différents dans Linux. L'un est simple et direct tandis que le second est indirect mais permet de déplacer la tâche en cours d'exécution. L'équilibreur de charge n'utilise que le mécanisme simple puisqu'il peut se permettre de ne jamais déplacer la tâche courante. Ce mécanisme consiste à verrouiller les deux run-queues entre lesquelles la migration va être faite puis à supprimer la tâche sélectionnée de la run-queue de départ et à la réinsérer dans la run-queue d'arrivée. Le mécanisme indirect n'est utilisé que lorsqu'une tâche change son affinité aux processeurs. La tâche est supprimée de la run-queue, mise dans une file particulière et un thread noyau spécial est ordonnancé juste ensuite. Ce thread peut alors appeler le mécanisme simple pour finir la migration.

2.2.5 Calcul de la priorité d'ordonnancement des tâches

Le tri des tâches à déplacer est fonction de leur priorités. Comme le noyau Linux gère les priorités de manière assez subtile [12] nous allons en préciser le fonctionnement ici. Il y a 140 niveaux de priorité, compris entre 0 (le plus élevé) et 139 (le plus bas). Les 100 premiers niveaux sont dédiés aux priorités temps-réel, dans l'ordre inverse de la norme POSIX. Les 40 derniers niveaux correspondent aux tâches UNIX normales. Ils représentent leur « niceness » (spécifié par l'utilisateur) affiné d'un bonus/malus variant entre +5 et -5. Ce bonus est calculé dynamiquement en fonction de l'« interactivité » de la tâche. Cette propriété représente la tendance de la tâche à plutôt dépendre de périphériques ou au contraire à être demandeuse en temps CPU. Elle est calculée via la variable `sleep_avg` qui est grossièrement la proportion de temps passé à attendre par rapport au temps d'exécution réel.

2.2.6 Comparaison avec l'implantation de FreeBSD

La récente version de FreeBSD 5.0 a intégré ULE [13] qui est la combinaison d'un ordonnanceur et d'un équilibreur de charge capable de prendre en compte la non-uniformité de la mémoire dans les machines SMP. L'implantation s'est inspirée de celle de Linux et il n'est donc pas étonnent que le fonctionnement partage beaucoup de similarités avec celui-ci.

Sans s'attarder sur les détails, l'une des différences principales est la politique de déclenchement basée sur une stratégie mixte (alors que celle de Linux est passive). Pour éviter l'oisiveté des processeurs une stratégie passive est employée car c'est le moyen le plus simple pour détecter le besoin de migration le plus tôt possible. Par contre une stratégie active est utilisée pour équilibrer les charges lorsque tous les processeurs sont chargés. Deux fois par seconde, une recherche de la run-queue la plus longue et de la run-queue la plus courte est lancée. Cette dernière reçoit alors un nombre de processus suffisant pour égaliser les deux longueurs. Par rapport au mécanisme de Linux cette méthode a l'avantage d'être plus légère en temps d'exécution tout en assurant un équilibrage global de la machine.

2.2.7 Conclusion

Cette implantation est le fruit d'un long processus de tests et de réglages et elle est maintenant réputée pour ses bonnes performances dans la majorité des cas. L'un des points fondamentaux est qu'elle est basée sur des règles simples, rapides à vérifier qui ne sont probablement pas optimales mais qui sont compensées par la haute fréquence avec laquelle l'équilibrage est appelé.

Cependant l'implantation a aussi ses limites, en particulier dans la gestion de l'affinité des tâches. Actuellement des modifications sont en cours pour prendre en charge les architectures NUMA (la version 2.6.7 du noyau inclue ce travail). De plus, comme nous allons le voir par la suite, l'équilibrage ne prend pas non plus correctement en compte la charge processeur des tâches temps-réel.

Maintenant que nous avons vu le fonctionnement d'ARTiS, le principe de l'équilibrage de charge et deux implantations pour des systèmes UNIX nous allons détailler le sujet principal du travail. Il faut définir un nouvel équilibrage de charge qui soit capable de traiter correctement la forte présence de tâches temps-réel ainsi que l'asymétrie des processeurs introduites par ARTiS.

Chapitre 3

Équilibrage de charge adapté au temps-réel

Tel qu'il existe actuellement dans Linux, l'équilibrage de charge fonctionne bien, tout particulièrement les cas réels courants. Cependant, avec les nouvelles notions introduites par ARTiS, son fonctionnement est loin d'être optimal. Avec l'apparition d'asymétrie entre les processeurs il est important que l'équilibreur de charge tienne compte des nouvelles affinités tâche/processeur qui sont induites. Par ailleurs, un problème encore plus fondamental doit être résolu, les CPU RT ne doivent pas se bloquer en attente d'un verrou prit par un CPU NRT, auquel cas la réactivité des CPU RT ne pourrait pas être assurée.

Nous allons maintenant spécifier un équilibreur de charge tenant compte des contraintes imposées par ARTiS et par la suite nous détaillerons sa mise en œuvre en décrivant son implantation et sa validation. Le modèle présenté a été décrit dans un article *work-in-progress* [14] qui est disponible en annexe C.

3.1 Les migrations dans ARTiS

Avant de pouvoir spécifier un nouveau mécanisme d'équilibrage il faut commencer par lister exhaustivement les différentes migrations qui peuvent apparaître entre les combinaisons CPU RT et CPU NRT. A chacune des migrations nous associerons des contraintes et d'autres propriétés particulières à prendre en compte dans le modèle. Notons qu'en raison des contraintes multiples, plusieurs mécanismes peuvent être liés à un seul type de migration.

CPU NRT vers CPU NRT Cette migration intervient lors de l'équilibrage de charge entre deux processeurs non temps-réel. C'est le seul type de migration qui existe dans le noyau Linux normal. Étant donné qu'aucun CPU RT n'est impliqué lors de cette migration, il n'y a pas besoin de prendre de mesure particulière. Le mécanisme original peut être directement réutilisé dans ce cas.

CPU RT vers CPU NRT Ce type de migration est utilisé principalement lorsqu'une tâche de priorité autre que RT0 tente de bloquer les interruptions. Dans ce cas, la tâche est immédiatement migrée. C'est le mécanisme de base d'ARTiS et il a déjà été implanté.

Il y a aussi un équilibrage de charge associé à cette migration. Quand un CPU NRT a une charge d'utilisation inférieure à un CPU RT (par exemple, des processus de calcul viennent de se terminer sur le CPU NRT), alors certaines tâches peuvent être déplacer afin de rééquilibrer les charges. Étant donné que les latences d'interruption sont meilleures sur les CPU RT, il faut essayer de garder les tâches RT1+ sur le processeur d'origine et privilégier le déplacement des tâches NRT. Notons qu'en pratique les tâches bloquent les interruptions généralement suffisamment souvent pour que cet équilibrage ne soit utile que très rarement. Bien sûr, afin de garantir la meilleure utilisation possible dans toutes les configurations, ce cas doit être pris en compte (par exemple si les tâches ne font que du calcul).

Ces deux déplacements entre processeur impliquent la modification de la run-queue des deux CPU concernés. Si le CPU NRT verrouille la run-queue du CPU RT (le mécanisme standard) alors les propriétés temps-réel de ce dernier ne pourront plus être garanties. C'est pour cela qu'il est nécessaire de trouver un mécanisme *ah-hoc* qui n'ait pas besoin de verrou, au moins du coté du processeur RT.

CPU NRT vers CPU RT Cette migration est utilisée dans deux contextes différents. Tout d'abord elle est utilisée pour ramener vers les CPU RT les tâches RT1+ qui ont migré. Dans ARTiS les tâches RT avec la plus grande priorité ne sont jamais déplacées, par contre les autres tâches temps-réel peuvent arriver sur un CPU NRT. Par conséquent, afin de leur fournir un temps de réponse aux interruptions aussi faible que possible, il faut les re-déplacer dès qu'elles sont de nouveau autorisées à s'exécuter sur un CPU RT.

Par contre, il ne faut pas négliger le coût de la migration et donc on ne devrait pas ramener une tâche sur un CPU RT si elle est sur le point de bloquer de nouveau les interruptions. Même s'il est évident qu'il n'est pas possible de prévoir *à l'avance* le comportement futur d'une tâche, le mécanisme d'équilibrage devra être capable de prédire approximativement le temps avant la prochaine migration.

Le second contexte dans lequel cette migration est utilisé est lorsqu'un CPU RT est moins chargé qu'un CPU NRT. Dans ce cas il faut déplacer certaines tâches NRT pour utiliser la puissance disponible sur le CPU RT. Bien que le point le plus important dans ARTiS soit d'assurer le meilleur temps de réponse possible aux tâches temps-réel, l'un des principaux avantages du système face aux autres solutions est de pouvoir utiliser au mieux la puissance de la plate-forme SMP, par conséquent cette migration ne doit pas être du tout négligée.

De même que dans la migration précédente, il est important de s'assurer que lors de cette migration le CPU RT n'utilise pas de verrou partagé avec le CPU NRT.

CPU RT vers CPU RT La migration entre CPU RT est utilisée uniquement pour garder leur charge équilibrée. Comme les deux processeurs impliqués sont du même type

l'algorithme d'équilibrage peut être pratiquement identique à celui d'équilibrage entre CPU NRT. Une contrainte supplémentaire quand même est que les processeurs ne devraient pas avoir à prendre de verrou inter-processeur, évitant ainsi de mettre en péril leurs propriétés temps-réel.

Maintenant que tous les cas de migration ont été étudiés il est possible de modéliser un équilibrage de charge qui respecte les contraintes requises lors des différentes situations de déséquilibre.

3.2 L'équilibreur de charge

L'implantation existante dans Linux a l'avantage d'être simple, claire, modifiable et d'avoir un fonctionnement correct déjà vérifié. C'est pour cela qu'il est apparut évident que la manière la plus simple de développer le nouvel équilibreur de charge était de se fonder sur les bases déjà disponibles dans le noyau Linux.

Le modèle que nous proposons se base sur le mécanisme d'équilibrage de charge actuellement implanté dans Linux (version 2.6.4) et l'améliore pour le rendre capable de répondre à toutes les contraintes définies précédemment.

3.2.1 Pondération de la run-queue

La politique d'appariement dans Linux sélectionne le processeur qui fournira éventuellement des tâches en choisissant le plus chargé. L'estimation de la charge est actuellement calculée en fonction de la run-queue, plus il y a de tâches dans la file d'exécution d'un CPU et plus ce CPU est considéré chargé. Cette estimation fonctionne bien lorsqu'il y a uniquement des tâches normales qui s'exécutent car leur temps d'accès au processeur est partagé : plus la run-queue sera longue et moins une tâche donnée aura de temps CPU autorisé.

Cette dernière hypothèse n'est plus vraie lorsque le système comporte un nombre non négligeable de tâches temps-réel. Étant donné que les tâches temps-réel ne rendent pas la main à des tâches de priorité moins élevée tant qu'elles ne l'ont pas autorisé explicitement (via certaines fonctions d'entrée/sortie, en s'endormant ou en faisant un `yield()`), le nombre de tâche dans la run-queue n'est plus forcément représentatif de la charge du CPU. Pour améliorer l'équité entre tâche normales il faut tenir compte du temps processeur consommé par les tâche temps-réel.

Par exemple, sur un système bi-processeur, si une tâche temps-réel consomme $3/4$ d'un processeur et qu'il y a 5 tâches normales également en cours d'exécution, il vaut mieux qu'une seule de ces tâches partage le processeur avec la tâche RT. En ne tenant compte que du nombre de tâches, l'équilibrage de charge préconiserait 3 tâches sur chaque CPU, c'est-à-dire que certaines tâches normales auraient le droit à $1/3$ d'un processeur tandis que d'autres uniquement à $1/8$. En tenant compte de la consommation réelle de la tâche temps-réel, l'équité est retrouvée et chacune des tâches a le droit à $1/4$

de temps processeur. La figure 3.2.1 illustre cet exemple. Ce type de scénario est particulièrement probable sur un système avec ARTiS car non seulement les tâches temps-réel jouent un rôle important mais en plus elles sont asymétriquement réparties entre les processeurs.

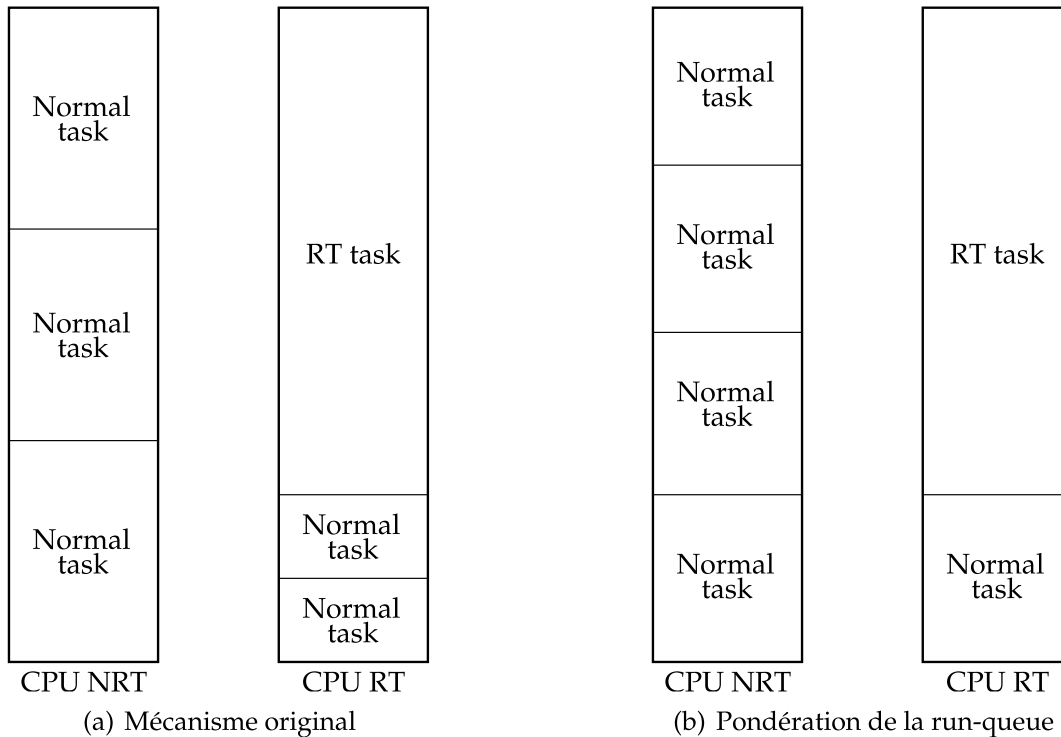

FIG. 3.1 – Amélioration de l'équité entre tâches normales grâce à la pondération de la run-queue.

Nous proposons donc une estimation de la charge légèrement plus sophistiquée qui pondère la longueur de la run-queue par la proportion de temps CPU consommé par les processus temps-réel. La charge de chaque processeur est alors estimée par la formule $L \times \frac{1}{1-RT}$ où RT représente la proportion de temps processeur consommé par les tâches temps-réel et L est le nombre de tâches **sans** compter les RT. Pour connaître le nombre de tâches n à déplacer pour rééquilibrer deux run-queues il faut résoudre l'équation suivante :

$$(L_r + n) \times M_r = (L_e - n) \times M_e$$

où L_r est la longueur de la run-queue réceptrice, M_r le coefficient multiplicateur tel que définit précédemment pour le processeur récepteur, L_e et M_e sont les valeurs équivalentes pour le processeur émetteur. On obtient donc :

$$n = \frac{L_e \times M_e - L_r \times M_r}{M_e + M_r}$$

Notons que lorsqu'aucune tâche RT n'est exécutée alors M_r et M_e valent tous les deux 1, ce qui implique que le nombre de tâche à déplacer est la moitié de la différence des longueurs de run-queue ; on retrouve donc la méthode du noyau original.

3.2.2 Élimination des verrous inter-processeurs

Comme nous l'avons vu précédemment, il n'est pas possible d'assurer les propriétés temps-réel d'un CPU RT s'il partage un verrou avec un CPU NRT. Il n'y a aucune garantie sur le temps que le verrou sera garder par le CPU NRT, donc la latence d'interruption sur le CPU RT ne peut plus être garantie. Ce scénario n'est pas très courant car normalement une tâche qui est sur le point de déclencher l'accès à un verrou sur un CPU RT est automatiquement migrée sur un CPU NRT. Cela peut cependant arriver dans des cas spéciaux, en particulier dans le code de l'ordonnanceur qui, pour des raisons évidentes, ne provoque jamais de migration.

L'ordonnanceur a toutes ses variables *locales*, donc peu de verrous inter-CPU sont pris. De tels verrous ne sont utilisé que dans le cadre de migration de tâche car il faut pouvoir modifier la run-queue du processeur distant (quelque soit la direction de la migration). Pour éviter ce cas nous proposons une nouvelle méthode de migration de tâche, moins direct mais qui évite les verrous.

Queues à accès non-bloquant Le principe est de transmettre les tâches via une queue (FIFO) qui relie les deux processeurs concernés. En ayant une queue attribuée à chaque couple orienté de processeur, celles-ci n'ont qu'un seul écrivain et qu'un seul lecteur, cette spécificité permet une implantation sans verrou comme décrit par John D. Valois [10]. Lorsqu'un processeur A souhaite migrer une tâche vers un processeur B, il sélectionne la queue (A, B) et y place la tâche. Ensuite il signale au processeur B (via une interruption inter-processeur) qu'il y a une queue qui contient une nouvelle tâche, finalement dès que le processeur B est disponible il va vider toutes les queues dont il est destinataire, y comprit (A, B). Nous ne prévoyons pas d'exécuter ARTiS sur des machines à plus de 32 processeurs (le nombre typique est de 4), donc même s'il est nécessaire d'avoir deux queues par couple de CPU la taille totale de ces structures ne devrait jamais atteindre une grandeur excessive.

Politique de déclenchement à stratégie active La modification de la migration décrite ci-dessus ne peut fonctionner que si c'est le processeur qui est à l'initiative de la migration qui envoie la tâche. S'il souhaite la recevoir, utiliser une queue à accès non-bloquant est plus laborieux car il faudrait déjà transmettre la demande de migration par une queue et ensuite recevoir la tâche par une seconde queue, et surtout ce procédé peut être relativement long puisque les processeurs sont asynchrones. Dans l'implantation de Linux actuelle, ce besoin de migration sur initiative du récepteur n'est présent que dans l'équilibrage de charge. Le noyau utilise une politique de déclenchement passive, c'est-à-dire que les processeurs peu chargés vont chercher les tâches sur les processeurs les

plus chargés.

Nous proposons d'inverser cette stratégie en faveur d'une active, les processeurs chargés délèguent une part de leur travail à des processeurs moins chargés. Lorsqu'il y a des processeurs oisifs, cette stratégie est moins efficace que la stratégie passive actuelle qui minimise le temps pour retrouver du travail. Par conséquent cette modification n'est appliquée que sur les équilibrages de charge qui ont absolument besoin de ne pas prendre de verrous inter-processeurs, c'est-à-dire tous ceux concernant des CPU RT.

3.2.3 Estimation de la prochaine tentative de migration

Lorsque l'équilibreur de charge est exécuté sur un CPU NRT et qu'il cherche un destinataire parmi les CPU RT, il doit tenir compte des propriétés spéciales d'ARTiS présentes sur l'autre processeur. Comme nous l'avons déjà évoqué précédemment, outre le fait que les tâches temps-réel doivent retourner sur ce type de CPU avec une priorité bien plus élevée que les autres tâches, il faut aussi privilégier les tâches qui resterons longtemps sur le nouveau processeur. De plus, le mécanisme de migration a un coût en temps non négligeable si jamais le déplacement vers un CPU RT est tout de suite suivi d'une migration vers un CPU NRT alors il vaut mieux éviter les deux déplacements.

Estimation du comportement futur Nous proposons donc un mécanisme d'équilibrage de charge qui favorise les tâches qui ont le moins de probabilité de bloquer les interruptions dans un futur proche. En observant de manière simple les tâches il est possible de mesurer la fréquence à laquelle elle bloque les interruptions et de noter l'instant auquel le dernier blocage (ou tentative) a été fait. A partir de ces statistiques on peut alors estimer la prochaine tentative de bloquer les interruptions (c'est-à-dire de migration si la tâche est sur un CPU RT) dans un intervalle de temps modeste. Ainsi on peut décider de ne pas migrer les tâches dont le temps estimé avant le prochain masquage est inférieur à un seuil. Ce mécanisme est représenté à la figure 3.2.

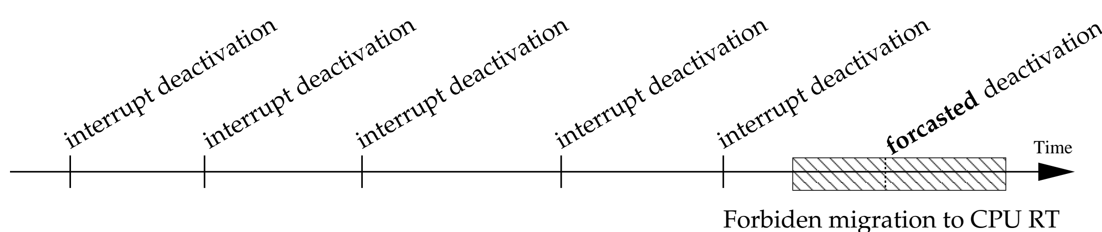

FIG. 3.2 – Période de non-migration forcée (rectangle hachuré). Le mécanisme de migration est coûteux en temps, s'il y a de fortes probabilités que la tâche soit de nouveau migrée vers un CPU NRT dans peu de temps, alors il vaut mieux ne pas la migrer vers un CPU RT.

Typiquement, à un instant t on peut se retrouver dans deux situations distinctes :

– Avant le moment de la prochaine tentative estimée. Si la durée entre l'instant courant et la prochaine tentative de blocage des interruptions est suffisamment grande on considère que le coût d'une double migration est petit et donc on permet la migration vers un CPU RT. Si la durée est trop courte, on interdit la migration.
– Après le moment de la prochaine tentative estimée. Si le moment prévu de la prochaine tentative de migration est déjà passé depuis une durée relativement grande, on considère que la tâche est entrée dans une phase de calcul et donc qu'elle ne bloquera plus les interruptions avant une longue durée. Dans ce cas on autorise aussi la tâche à être migrée vers un CPU RT.

Représentation mathématique Ces deux conditions peuvent être représentées mathématiquement par les deux formules suivantes :

$$\begin{cases} (P_m + t_d) - t > C \\ K \times t_d < t - P_m \end{cases}$$

C est un temps de l'ordre de plusieurs tics d'ordonnancement et K est un coefficient entre 2 et 100 qui sera fixé lors de l'implantation à l'aide de tests. P_m représente la période moyenne entre deux blocages, t_d l'instant du dernier blocage et t l'instant courant. Les constantes n'ont pas encore été fixées précisément, cela devra être fait lors de l'implantation. Ces deux tests permettent alors à la fonction d'équilibrage « CPU NRT vers CPU RT » de mettre de coté des tâches qui bloquent fréquemment les interruptions et qui sans cette vérification ne cesseraient d'effectuer des aller-retour entre processeurs RT et processeurs NRT.

3.2.4 Association du type de tâche au type de processeur

Politique de désignation locale Le mécanisme original du noyau Linux de désignation locale, qui doit déterminer quels sont les processus qui vont être déplacés une fois le processeur cible choisi, ne peut pas être simplement modifié. Pour être capable de tenir compte des différentes combinaisons de processeurs émetteur/récepteur et de tâches il est nécessaire de dériver plusieurs mécanismes. Chacun aura ainsi l'occasion de répondre aux contraintes précises que nous avons spécifiées dans la section 3.1.

Pour les deux équilibrages de charges symétriques (NRT vers NRT et RT vers RT) la politique de désignation locale peut rester la même que l'originale. En effet, la migration ne changera pas le domaine (NRT ou RT) d'exécution de la tâche donc les avantages et inconvénients seront les mêmes avant et après. Il n'y a donc pas de raison d'utiliser des critères de sélection autres que le noyau original.

En ce qui concerne l'équilibrage s'exécutant sur les CPU RT vers des CPU NRT, une contrainte vient s'ajouter : il faut privilégier la sélection de tâches normales. Les tâches RT ont un avantage à rester sur le CPU RT car un meilleur temps de latence leur est garanti, contrairement au mécanisme original, les tâches RT ne seront donc sélectionnées qu'après les tâches normales.

Pour l'équilibrage de charge à partir d'un CPU NRT vers un CPU RT, c'est bien évidement l'inverse et il faut migrer (ramener) les tâches RT en priorité. Pour pouvoir tenir compte des différences de contraintes entre les deux types de tâches, deux mécanismes différents sont dérivés du mécanisme d'équilibrage original :

- Celui qui gèrent les tâches temps-réel ne tient pas compte de la différence de charge entre le CPU courant et les CPU RT, le plus important étant de minimiser le temps d'exécution des tâches RT sur un CPU NRT (tout en gardant en tête l'estimation de la prochaine migration comme vu précédemment). En plus, ce mécanisme ne sélectionnera le CPU destination qu'une fois la tâche désignée afin de choisir le plus approprié (en fonction de la charge du processeur et de l'affinité de la tâche) ;
- Le mécanisme de désignation locale des tâches normales ne s'exécutera que si le précédent n'a pas put déplacer de tâche (en cas de migration les statistiques ne sont plus à jour). Il est très similaire au mécanisme original mais il tient aussi compte de la prochaine migration estimée pour sélectionner les tâches.

Politique de déclenchement La dernière modification qu'il reste à décrire vis-à-vis du mécanisme original est le déclenchement. Lorsque la stratégie est active (le processeur « pousse » les tâches) il n'y a plus d'intérêt à déclencher l'équilibrage de charge à chaque fois que le processeur devient oisif : il n'y a pas de tâche à transmettre. Tous les équilibrages impliquant un CPU RT ont ce type de stratégie donc le déclenchement en cas d'oisiveté sera purement désactivé sur ces CPU. Par contre l'équilibrage de charge pour ramener les tâches RT sur des CPU RT devra être exécuté assez souvent pour les laisser aussi peu que possible sur les CPU NRT. La fréquence exacte devra être décidée à l'implantation mais nous pouvons estimer qu'elle sera de l'ordre de 4 fois plus haute que les autres équilibrages, environ 20 fois par seconde au lieu de 5.

3.3 Mise en œuvre

Une fois que le modèle a été défini l'étape la plus importante reste encore à faire : la mise en œuvre. L'étape d'implantation est évidement suivit par la validation qui vérifiera autant le bon fonctionnement de l'implantation que la justesse du modèle. Comme dans toute implantation, au fur et à mesure de la mise en œuvre le modèle est corrigé en fonction des problèmes rencontrés.

3.3.1 Implantation

L'implantation de l'équilibrage de charge pour ARTiS dans le noyau n'est pas encore terminée. Son avancement est actuellement estimer à 30% du projet complet. Le développement est relativement lent en raison des difficultés supplémentaires impliquées par la modification du noyau (nécessité de redémarrage à chaque modification,

limitation du debugger...). Nous allons parcourir les différents points principaux déjà implantés et les techniques sur lesquelles ils se fondent.

Enregistrement des statistiques

Les algorithmes définis par le modèle se basent sur des informations concernant l'état du système ou des tâches. La plupart de ces informations peuvent être directement obtenues depuis des variables qui sont déjà utilisés par le noyau ou indirectement en dérivant plusieurs variables (par exemple la priorité d'une tâche, la longueur d'une run-queue, heure du dernier déclenchement de l'équilibrage de charge...). Néanmoins il y a des informations requises par le modèle qui sont pas disponibles par défaut et qui ont dû être mises en place.

Statistiques sur le temps consommé par les tâches temps-réel Le noyau enregistre un certain nombre de statistiques sur la proportion de temps consommé sur chaque processeur comme par exemple le nombre de cycles système, utilisateur et oisifs. Ces informations sont destinées principalement à l'utilisateur. Elles sont représentées par le nombre de tic d'horloge utilisés depuis le démarrage du système en fonction du processeur et du type d'action. Dans le cadre de la pondération de la run-queue par la quantité de temps consommé par les tâches RT, il faut être capable de connaître à un instant donné la proportion de CPU consommé par les processus temps-réel (nommé auparavant RT). Bien sûr, si on mesurait une telle valeur en instantané on obtiendrait ou bien 1 (il y a un processus temps-réel ordonnancé) ou bien 0 (ce n'est pas un processus temps-réel qui est actuellement ordonnancé). Pour obtenir une valeur plus représentative nous lissons cette valeur sur une durée de 500ms en échantillonnant toutes les millisecondes (un tic d'horloge).

Nous avons donc mis en place un calcul de la charge instantanée avec atténuation temporelle à la manière du calcul de charge globale (`CALC_LOAD()`) déjà implanté dans Linux. La formule mathématique sous-jacente est ainsi :

$$C_{new} = C_{old} \times (1 - \frac{1}{e^{\frac{1}{500}}}) + C_{cur} \times \frac{1}{e^{\frac{1}{500}}}$$

Où C_{new} est la nouvelle valeur de la charge, C_{old} est l'ancienne valeur et C_{cur} est la valeur courante. Pour pouvoir gérer efficacement les chiffres décimaux les calculs sont effectués en virgule fixe (les calculs en virgule flottante sont difficiles à mettre en œuvre dans le noyau). Le code correspondant est disponible en annexe A. Enfin, la dernière chose à laquelle il faut faire attention est qu'en raison de la troncature lors du calcul, la valeur maximale possible est légèrement inférieure à 1, elle vaut $0,9846$. C'est grâce à cette propriété que le diviseur de la formule $\frac{1}{1-RT}$ n'est jamais nul. Un extrait du code de l'implantation est disponible en annexe A.

Statistiques sur la migration d'une tâche L'autre type d'information qu'il a fallut implanter concerne les tentatives de blocage des interruptions qui, si la tâche est sur un

CPU RT, entraînent la migration. Dans la fonction `artis_try_to_migrate()` qui est appelée à chaque blocage et qui décide si une migration doit être déclenchée, un appel à une fonction `artis_migration_stat()` mettant à jour les statistiques a été ajouté. Si la tâche doit migrer ou si elle ne migre pas pour l'unique raison qu'elle est sur un CPU NRT alors cet appel est effectué. Notons que si la tâche ne migre pas parce que le code est dans une section non migrable ou d'autres raisons indépendantes de l'état actuel de la tâche alors les statistiques ne doivent pas être mises à jour.

A chaque appel deux valeurs sont sauvegardées : l'heure et la nouvelle fréquence de migration. La fréquence est représentée par une moyenne de la période au cours des 100 dernières tentatives de migration, en pondérant de la même façon que pour le temps consommé par les tâches RT.

Pondération de la run-queue

La pondération de la run-queue a été difficile à mettre en place et n'est toujours pas complètement vérifiée. Initialement une version plus simple avait été proposée où l'on ne décomptait pas de la longueur de la run-queue le nombre de tâches RT en cours d'exécution. Lorsque les tâches temps-réel consommaient moins de temps qu'une tâche normale, alors la queue paraissait plus grande (la longueur multipliée par un petit peu plus que 1) au lieu d'apparaître plus petite (la longueur moins le temps que les tâches RT n'ont pas consommé). Le comptage du nombre de tâches RT est fait de manière très simple en même temps que le comptage des tâches. De plus il faut aussi modifier la valeur lorsque l'ordonnancement d'une tâche change entre temps-réel et normal.

Implantation des queues à accès non-bloquant

Initialement l'implantation des FIFO avait été prévu en utilisant une version de l'algorithme de John D. Valois [10] qui permet d'être sans attente et sans blocage s'il y a qu'un écrivain et qu'un lecteur. Pour fonctionner, cet algorithme nécessite qu'il reste toujours au moins un élément dans la file. En général cela ne pose pas de problème car les éléments ne sont que des containers créés spécialement lors de l'insertion. Malheureusement dans le noyau la création de containers spéciaux n'est pas possible car l'allocation de mémoire utilise des verrous. L'utilisation directe de la structure de la tâche n'est pas non plus possible car il y a des problèmes de cohérence dès qu'une tâche qui a migré et qui est gardée dans une file est réinsérée dans une autre file.

Actuellement nous utilisons des FIFO asymétriques en se basant sur un algorithme écrit par Alexandre Hesseman. Un seul du lecteur ou de l'écrivain est assuré de ne pas avoir de verrous (en fonction de l'implantation). Dans la majorité des situations qui doivent être gérées dans le contexte nouveau d'ARTiS cela est suffisant. Pour une queue allant d'un CPU RT vers un CPU NRT on peut placer une FIFO sans verrou pour l'écrivain et réciproquement pour une queue d'un CPU NRT vers un CPU RT. Nous n'avons pas encore de solution lorsqu'il faut relier deux CPU RT. L'algorithme repose sur l'utilisation de deux files indépendantes et d'un sélecteur. Détaillons le cas

d'**écriture** sans verrou. Le lecteur lit le sélecteur puis l'inverse et tente d'accéder à la queue indiquée par le sélecteur s'il n'y a pas de verrou, sinon il attend. Une fois que le verrou est relâché il vide la queue. Indépendamment, l'écrivain commence par activer le verrou de la file (ce qui est toujours possible car il est le seul à le faire), il lit la queue indiquée par le sélecteur puis il remplit l'**autre** queue. Pour finir il relâche le verrou. La figure 3.3 présente un schéma de l'organisation de la FIFO à deux files.

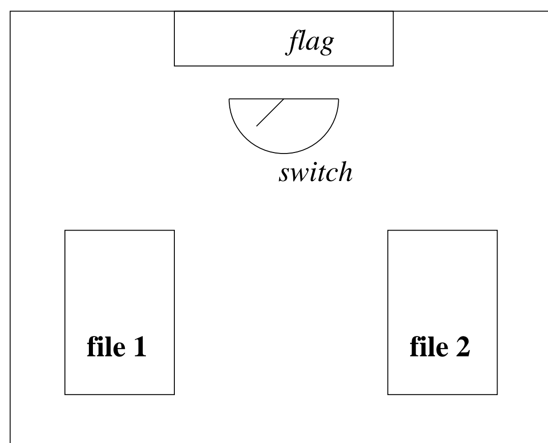

FIG. 3.3 – Schéma de la FIFO

Actuellement l'implantation est en court. Aucun des mécanismes n'a encore été entièrement validés. La spécialisation de l'équilibrage en fonction des processeurs de départ et d'arrivée n'a pas du tout été débutée. La continuation de cette étape importante permettra de confirmer le modèle et surtout de détecter des problèmes qui n'ont pu être envisager avant. Ce sera alors l'occasion de peaufiner le modèle. Faite en parallèle, l'étape de validation servira à vérifier le bon fonctionnement de l'implantation et à la comparer par rapport à d'autres systèmes.

3.3.2 Validation

Le principe de la validation est d'une part de vérifier que les modifications améliorent effectivement le comportement de l'application (et indirectement de détecter les cas où il y aurait régression) et d'autre part de vérifier que le comportement implanté est conforme aux spécifications du modèle. Dans le cadre du travail nous avons créé un programme qui observe l'équilibrage de charge du système sur lequel il s'exécute nommé *lollobrigida*. Son fonctionnement est simple, il lance un scénario défini par l'utilisateur qui comporte un certain nombre de tâches ayant des propriétés définies. Une fois l'exécution terminée il affiche des statistiques sur le fonctionnement du système. En lançant le même scénario sur plusieurs systèmes, *lollobrigida* permet de comparer les équilibreurs de charge entre eux.

Scénarios

Des programmes de tests existaient déjà mais ils fonctionnent en général en exécutant des tâches réelles (une compilation du noyau en parallèle par exemple). Bien que cela permette de tester des cas très réalistes, cela laisse peu de possibilité de tester explicitement des cas particuliers où des problèmes peuvent être mis en évidence. De plus les programmes sont trop complexes pour rendre visible l'association d'une combinaison d'actions particulière avec un comportement spécial de l'ordonnanceur. Enfin, les tests habituels ne sont pas facilement reproductives étant donné le nombre de programmes sur lesquels ils dépendent. C'est pourquoi les scénarios de *lollobrigida* sont basés sur un fonctionnement très simple. Les paramètres disponibles sont uniquement ceux strictement nécessaires pour pouvoir simuler les comportements typiques de tâches.

Définitions des scénarios Un scénario est un ensemble de tâches aux propriétés strictement définies et dont les exécutions débutent en même temps. Concrètement, c'est un fichier texte qui contient à la suite la description de chacune des tâches. Il existe sept paramètres disponibles pour représenter une tâches, la plupart sont optionnels, auquel cas ils prennent des valeurs par défaut. Les paramètres disponibles sont :

- le nombre d'itérations de programme à faire (contrôle la quantité de calcul) ;
- la période entre deux endormissements de la tâche ;
- la durée d'attente à chaque endormissement (contrôle la densité moyenne de calcul de la tâche) ;
- la période entre deux appels systèmes qui bloquent les interruptions (contrôle la fréquence de migration sur un noyau ARTiS) ;
- le type d'ordonnancement de la tâche (normal ou temps-réel) ;
- la priorité d'ordonnancement de la tâche (dépend du type d'ordonnancement) ;
- le masque des processeurs autorisés (contrôle l'affinité de la tâche à certain processeurs).

Chacun de ces paramètres est décrit en détail, avec les valeurs qu'il peut prendre, dans la documentation de *lollobrigida*. Un exemple simple de scénario est disponible en annexe B.

Reproductibilité des mesures L'un des aspects majeur du programme que nous avons fixé dès le début du développement était la possibilité de relancer un scénario de manière quasi identique quelque soient le système et les conditions d'exécution. Pour assurer ce déterminisme, le comportement des tâches factices dépend exclusivement des paramètres spécifiés dans le scénario. La lecture du scénario est faite lors de la phase d'initialisation. Pour garantir que le départ des mesures se fait de manière toujours identique, toutes les tâches se bloquent après leur initialisation. La transmission d'un seul signal émit à toutes les tâches factices au début de la mesure assure qu'elles commencerons au même moment.

Information de sortie

Pour laisser décider l'utilisateur des qualités de l'équilibrage de charge *lollobrigida* affiche à la fin de l'exécution des tâches un certain nombre de statistiques. Notamment on y trouve le temps d'exécution de chaque tâche, ainsi qu'un un résumé pour chaque groupe de tâche ayant les mêmes propriétés. Cela permet d'estimer l'équité entre tâches, elle est bonne si l'écart type est faible. Pour chaque tâche, la proportion de temps passé sur chaque processeur est aussi indiquée. Cela permet de vérifier le respect de l'équilibreur de charge vis-à-vis de l'affinité des tâches. Sur un noyau ARTiS il est aussi indiqué le nombre de migrations faites pour éviter un blocage des interruptions sur un CPU RT. Un nombre faible indique en général que l'ordonnanceur a privilégié la localité des tâches. Enfin, la proportion de temps passé oisif par chaque processeurs au cours du scénario permet d'estimer l'efficacité du l'équilibrage.

Conclusion

Un système Linux capable de garantir des contraintes temps-réel fournit l'aisance d'utilisation et de développement pour laquelle Linux est réputé tout en répondant aux besoins de réactivité desquels dépendent de nombreuses applications principalement industrielles et multimédia. Plusieurs approches existent déjà pour permettre un tel système. Cependant les approches co-noyau obligent une programmation asymétrique tandis que si cette asymétrie est déplacée au niveau des processeurs on limite la puissance de l'architecture SMP sur laquelle repose cette approche. ARTiS se base sur ce dernier type d'approche mais contourne la principale limite en autorisant les tâches normales à s'exécuter aussi sur un processeur RT tant qu'elles ne désactivent pas les interruptions.

Outre la détection et la migration des tâches mettant en danger le temps-réel, ARTiS nécessite aussi un équilibrage de charge particulier qui puisse prendre en compte l'asymétrie des processeurs. Le travail du stage de DEA a consisté à définir le nouveau modèle d'équilibrage. En se basant sur l'implantation existante nous avons modifié le comportement de certains mécanismes, notamment en éliminant les verrous interprocesseurs qui étaient utilisés pour migrer des tâches mais aussi en spécifiant un type de sélection de tâche en fonction des processeurs émetteur et récepteur. Nous avons aussi introduit de nouvelles notions telles que le temps processeur consommé par les tâches temps-réel et l'estimation de la prochaine tentative de migration d'une tâche.

Perspectives À court terme, nous devons terminer l'implantation qui est actuellement en cours. Tout d'abord une version minimale sera développée en parallèle de la validation. Puis au fur et à mesure que le fonctionnement correct sera vérifié, nous pourrons nous attacher à lever les restrictions de cette l'implantation comme par exemple la communication sans aucun verrou entre deux processeurs RT. Alors pourra avoir lieu une comparaison détaillée des performances et des comportements entre les différents équilibreurs de charge et en fonction des tâches exécutées.

À plus long terme, une adaptation à l'ordonnanceur amélioré de la version 2.6.7 sera à envisager. Cette version introduit la notion d'affinité hiérarchique entre tâche et processeur. Il faudra voir si la représentation des domaines RT et NRT via une hiérarchie permet de prendre en compte moins spécifiquement certains des aspects de l'équilibrage de charge ARTiS. Finalement une importante étape sera de valider le bon fonctionnement de l'équilibrage dans le cadre du projet HYADES, lors de l'utilisation d'une

application réelle qui ait à la fois besoin de puissance de calcul et de garanties temps-réel.

Bibliographie

[1] Kevin Morgan. Preemptible Linux : A reality check. White paper, MontaVista Software, Inc., 2001.

[2] Pierre Cloutier, Paolo Montegazza, Steve Papacharalambous, Ian Soanes, Stuart Hughes, and Karim Yaghmour. DIAPM-RTAI position paper. In *Second Real Time Linux Workshop*, Orlando, FL, novembre 2000.

[3] Finite State Machine Labs, Inc. RealTime Linux (RTLinux). `http://www.fsmlabs.com/`.

[4] Victor Yodaiken. The RTLinux manifesto. In *Proc. of the 5th Linux Expo*, Raleigh, NC, mars 1999.

[5] Steve Brosky and Steve Rotolo. Shielded processors : Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03*, Nice, France, avril 2003.

[6] Stephen Brosky. Symmetric multiprocessing and real-time in PowerMAX OS. White paper, Concurrent Computer Corporation, Fort Lauderdale, FL, 2002.

[7] Sillicon Graphics, Inc. REACT : Real-time in IRIX. Rapport technique, Sillicon Graphics, Inc., Mountain View, CA, 1997.

[8] Momtchil Momtchev and Philippe Marquet. An asymmetric real-time scheduling for Linux. In *Tenth International Workshop on Parallel and Distributed Real-Time Systems*, Fort Lauderdale, FL, avril 2002.

[9] Philippe Marquet, Julien Soula, Éric Piel, and Jean-Luc Dekeyser. An asymmetric model for real-time and load-balancing on Linux SMP. Rapport de recherche 2004-04, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, avril 2004.

[10] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, octobre 1994.

[11] Cyril Fonlupt. *Distribution Dynamique de Données sur Machines SIMD*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 1994.

[12] Robert Love. *Linux Kernel Development*. Sams Publishing, août 2003.

[13] Jeff Roberson. ULE : A modern scheduler for FreeBSD. In *USENIX BSDCon'03*, San Mateo, CA, septembre 2003.

[14] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Load-balancing for a real-time system based on asymmetric multi-processing. In *16th Euromicro Conference on Real-Time Systems, WIP session*, Catania, Italy, juin 2004.

Annexe A

Extrait du code d'implantation des statistiques

```
#ifdef ARTIS_LB

#define FSHIFT_P 11 /* nr of bits of precision */
#define FIXED_P_1 (1<<FSHIFT_P) /* 1.0 as fixed-point */
#define EXP_PERIOD 2028 /* 1/exp(1attempt/100attempt) as fixed-point */

#define CALC_PERIOD(period,n) \
*period *= EXP_PERIOD; \
*period += (n)*(FIXED_P_1-EXP_PERIOD); \
*period >>= FSHIFT_P;

/*
 * update the statistics about the migration attempts by saving the
 * new timestamp of the attempt and re-computing the average period.
 * Warning: this function must be lock-free
 */
void artis_migration_stat(unsigned long long *la, unsigned long long *ap)
{
unsigned long long old = *la;
unsigned long long current_period;

*la = sched_clock(); // on ia64 needs patch sched-clock-overflow.patch or > 2.6.7

current_period = (*la - old) * FIXED_P_1;

CALC_PERIOD(ap, current_period);
}

#define FSHIFT_RT 15 /* nr of bits of precision */
#define FIXED_RT_1 (1<<FSHIFT_RT) /* 1.0 as fixed-point */
#define EXP_RT 32703 /* 1/exp(1tick/500ticks) as fixed-point */
#define MAX_RT (FIXED_RT_1 - (FIXED_RT_1/(FIXED_RT_1 - EXP_RT))) /* the maximum possible
```

```
#define CALC_RT(load,n) \
*load *= EXP_RT; \
*load += (n)*(FIXED_RT_1-EXP_RT); \
*load >>= FSHIFT_RT;
/*
 * update the statistics about the RT task CPU consumption and the time
 * they donot consume. We always insert 1 or 0 in the mean so it should
 * always be between 0 and 1.
 */
void artis_rt_stat(int ticks, task_t *p, struct cpu_usage_stat *cpustat)
{
int rt_ticks = 0;

if (unlikely(rt_task(p)))
rt_ticks = FIXED_RT_1;

for (; ticks > 0; ticks--)
CALC_RT(&cpustat->rt, rt_ticks);
}

/*
 * It is called RT ratio but actually it is the inverse of the cpu time NOT consumed
 * by RT tasks.
 * Returns 1, the minimum, when no RT process was ran.
 * The maximum value of the rt stat is MAX_RT so it returns 504
 * when only RT tasks are running.
 */
unsigned int get_rt_ratio(int cpu)
{
unsigned int nrt;
struct cpu_usage_stat *cpustat = &(kstat_cpu(cpu)).cpustat;

// the divisor is never null because cpustat->rt <= MAX_RT < FIXED_RT_1
return FIXED_RT_1 / (FIXED_RT_1 - cpustat->rt);
}
#else /* ARTIS_LB */

#define get_rt_ratio(cpu) 1
#define artis_rt_stat(ticks, p, cpustat)
#define artis_migration_stat(la, ap)

#endif /* ARTIS_LB */
```

Annexe B

Exemple de scénario lollobrigida

```
# this scenario should theoritically be better with the load-balancing
# taking into account the time consumed by RT tasks. With normal lb, there
# should be a balance like (4n0rt:3n1rt:3n1rt:3n1rt)
# As the RT tasks take most of the power of the CPU, the aware lb should give:
# (10n0rt:1n1rt:1n1rt:1n1rt)

# the normal tasks
{
cpu_mask = 0xffff
loop = 10000000
}
#2
{
cpu_mask = 0xffff
loop = 10000000
}
#3
{
cpu_mask = 0xffff
loop = 10000000
}
#4
{
cpu_mask = 0xffff
loop = 10000000
}
#5
{
cpu_mask = 0xffff
loop = 10000000
}
#6
{
cpu_mask = 0xffff
loop = 10000000
```

```
}
#7
{
cpu_mask = 0xffff
loop = 10000000
}
#8
{
cpu_mask = 0xffff
loop = 10000000
}
#9
{
cpu_mask = 0xffff
loop = 10000000
}
#10
{
cpu_mask = 0xffff
loop = 10000000
}
#11
{
cpu_mask = 0xffff
loop = 10000000
}
#12
{
cpu_mask = 0xffff
loop = 10000000
}
#13
{
cpu_mask = 0xffff
loop = 10000000
}

# the RT tasks
{
cpu_mask = 0x2
sched = FIFO
priority = 99
loop = 110000000
sloop = 5000
sleep = 5000
}
#2
{
cpu_mask = 0x4
sched = FIFO
priority = 99
```

```
loop = 110000000
sloop = 5000
sleep = 5000
}
#3
{
cpu_mask = 0x8
sched = FIFO
priority = 99
loop = 110000000
sloop = 5000
sleep = 5000
}
```

Annexe C

Load-balancing for a real-time system based on asymmetric multi-processing

LIFL Report # 2004-06

Load-Balancing for a Real-Time System Based on Asymmetric Multi-Processing*

Éric PIEL
Eric.Piel@lifl.fr

Philippe MARQUET
Philippe.Marquet@lifl.fr

Julien SOULA
Julien.Soula@lifl.fr

Jean-Luc DEKEYSER
Jean-Luc.Dekeyser@lifl.fr

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France

April 2004

Abstract

ARTiS is a project that aims at enhancing the Linux kernel with better real-time properties. It allows to retain the flexibility and ease of development of a normal application for the real-time applications while keeping the whole power of SMP (Symmetric Multi-Processors) systems for their execution.

Based on the introduction of an asymmetry between the processors, distinguishing real-time and non real-time processors, the system can insure low interrupt latencies to real-time tasks. Furthermore, every processor can execute all the tasks, excepted when they request real-time endangering functions. In this case the task is moved before continuing to be executed. A first version of ARTiS has demonstrated this is technically possible. Unfortunately, the original load-balancing mechanism of Linux is not aware of this enhanced design.

We have studied all the types of migration possible between the combinations of a real-time specialized processor and a general one. From the deducted requirements, we have specified special mechanisms and policies taking into account both performances and real-time specificities. We are currently working on implementing those particular load-balancing functions within the ARTiS system.

1 Real-Time and Load-Balancing on SMP

There exists two types of approach to obtain real-time properties from the Linux kernel. One consists in running the RT tasks in a special designed kernel running in parallel, this is what does RTAI [2]. The drawback is that the programming model and configuration methods are different from the usual one: Linux tasks are not real-time tasks and real-time activities can not benefit of the Linux services. The second approach relies on the shielded processors or asymmetric multiprocessing principle. On a multiprocessor machine, the processors are specialized to real-time or not. Concurrent Computer Corporation RedHawk Linux variant [1] and SGI REACT IRIX variant [6] follow this principle. However, since only RT tasks are allowed to run on shielded CPUs, if those tasks are not consuming all the available power then there is free CPU time which

*This work is partially supported by the ITEA project 01010, HYADES

is lost. ARTiS extends this second approach by also allowing normal tasks to be executed on those processors as long as they are not endangering the real-time properties.

ARTiS insures a possible processor preemption when the system has to schedule a real-time process [5, 4]. The main principle is to distinguish two kinds of CPUs in the multi-processor system: specialized CPUs, a part oriented toward real-time (the so-called RT CPUs) and another part serving all the other tasks (the so-called non real-time CPUs, NRT CPUs). Two types of RT tasks are characterized, the RT0 which have the highest priority and are binded to a processor and the RT1+ which have lower priority and may migrate between processors. Every task is allowed to run on an RT CPU but tasks which are not RT0 will not be allowed to perform real-time endangering functions on this particular CPU. We have implemented ARTiS in the 2.6 Linux kernel and tested it on x86 and IA-64 platforms. The implantation of the ARTiS system relies on the fact that any call to `preemt_disable()` or `local_irq_disable()` from a task without the highest RT priority leads systematically to the migration of this task to an NRT CPU. Thus the RT CPUs remain able to face to real-time activities without long latency.

Nevertheless, the system has to insure a migration mechanism: a NRT task must leave a RT CPU if it jeopardizes the real-time response time on this RT CPU. Furthermore, to maximize the utilization of the whole of the CPU, those NRT tasks may have to come back on the RT CPU latter. The standard Linux kernel already provides a migration mechanism. Unfortunately, this standard mechanism is not sufficient because it is not aware of the specialization of the different processors and also because it holds in the same time locks of two different CPUs (possibly one being a RT CPU) to insure the task migration: the RT CPU may have to wait after the completion of an operation on a NRT CPU which is unacceptable. In order to address those issues we have to design a specialized load-balancing mechanism.

Usually, the load-balancing mechanism aim is to move the running tasks across the CPUs in order to insure that no CPU is idle while some tasks are waiting to be scheduled on other CPUs. It should minimize the total running time by a set of tasks. The characteristics of a load-balancing can be enumerated as follow:

- information update policy: how to renew statistics about the entire system,
- trigger policy: how to decide it is time to redistribute the tasks,
- selection policy: method to select unbalanced nodes,
- local designation policy: method to select the tasks that will move,
- pairing policy: method to select the destination node for a given task.

The trigger policy can be either of type “pull” –the low loaded CPUs initiate the load-balancing and pull the tasks from another CPU– or “push” –over-loaded CPUs initiate the load-balancing in order to push some of their tasks– or a mix of both.

2 ARTiS Migrations

In order to specify a more advanced load-balancing mechanism it is necessary to distinguish the different types of migration according to the specialization of the CPUs involved: RT and NRT CPUs. There may be several mechanisms associated to a given kind of migration in order to fit all the scenarii involving this migration.

NRT CPU to NRT CPU This migration is used for load-balancing between non real-time CPUs. As no RT CPU is involved there is not particular requirement to take care. The original Linux mechanism [3] can be kept for this kind of load-balancing.

RT CPU to NRT CPU This type of migration is mainly called when a task on a RT CPU tries to disable the preemption (endangering the RT properties). This is the core mechanism of ARTiS and it is already implemented. There is also a load-balancing mechanism related to this migration. When a NRT CPU has less load than a RT CPU, some of the NRT tasks should be moved to

the NRT CPU. RT tasks should not be moved as there is better response time on the RT CPUs. In practice, most of the tasks trigger preemption disabling code often enough so that this load-balancing is not needed. Still, it is necessary to handle this case in order to guaranty the best use of all the CPUs in every configuration (for instance with tasks doing only computational work). In this kind of migration it is important that the RT CPU does not take a lock shared with the NRT CPU.

NRT CPU to RT CPU This migration is used in two different contexts. First, it is used to move back as soon as possible to a RT CPU the RT tasks which have reenabled the preemption. In the ARTiS model, the highest priority RT tasks are always on a RT CPU but other RT tasks may migrate to a NRT CPU. Therefore, to provide the smallest latencies to them it is necessary to bring them back to any RT CPU as soon as they are allowed to. However, as the migration process costs some time it is also important not to move back a task that may need soon to migrate again. Although it is obviously impossible to exactly know in advance the future behavior of a given task, the local designation part of the load-balancing algorithm has to approximately predict the next time of migration.

Second, this migration is required to load-balance the NRT tasks if a RT CPU has free time available. Even if the most important point is to keep the RT properties on the RT CPUs, this load-balancing represent a major advantage of ARTiS and so it must not be neglected.

Both moves between the CPUs implies the modification of the runqueue of each processor. If the NRT CPU locks the runqueue of the RT CPU (which is the standard mechanism) then the RT properties of this later cannot be guaranteed. It is therefore necessary to find a transfer mechanism which is lock free at least on the RT CPU side.

RT CPU to RT CPU The migration between two RT CPUs is solely used to balance their load. The algorithm can be very similar to the NRT to NRT algorithm with the exception that it has to avoid locks between two CPUs that will possibly lead to a jitter on the expected latencies.

3 Migration Implementation

Starting from the specific requirements described above we have defined the algorithms and implementations of the dedicated load-balancing mechanisms in ARTiS.

Lock-free queues One of the main change which is required from the original load-balancing mechanism is the removal of inter-CPU locks. In order to be able to insure the RT properties of the RT CPUs, there should not be locks that can be taken both by NRT and RT CPUs. If a RT CPU tries to take a lock already taken by a NRT CPU it will have to wait after it. To avoid this particular sequence we decided to avoid taking shared locks. The only shared lock is on the runqueue which has to be modified from the other CPU when inserting a migrating task. This is why this mechanism has to be changed to an indirect one. Instead of removing the task from a runqueue and adding it to another runqueue, we insert it to a special FIFO connecting the two CPUs. The task is dequeued later and asynchronously by the second CPU. Because this FIFO has only one writer and one reader, it is possible to access it without lock, as described in [7]. We are not expecting machines with more than 32 CPUs, so even if there are two queues per couple of CPUs, the size of these data structure should never be excessive.

Trigger policy By default, the Linux kernel [3] uses a "pull" policy for the load-balancing. However, with the FIFO mechanism which is required to avoid the locks, this policy has a more complex implementation (leading to longer delays) than the "push" policy. The use of a queue with this second policy is straightforward. Consequently, for the new load-balancing functions, we invert the default policy. A CPU will look for the less busy CPU and then select tasks to send from its own runqueue to the second CPU. In order to lower the latency between the moment a

task is inserted and the moment the second CPU reads the FIFO, a signal (an IPI - Inter-Processor Interrupt) is sent, warning about a new task in the queue. In addition, concerning the RT tasks on a NRT CPU, the time spent on this processor must be minimized so this special load-balancing function is triggered more often than the other functions, at every scheduling (typically every millisecond).

Local designation criteria The designation mechanism of the load-balancing which is charged to decide which task is better to migrate has to be adapted to each type of load-balancing. For the symmetric load-balancing (NRT to NRT and RT to RT) the original criteria can be kept as the requirements are the same than in a normal configuration. For the RT to NRT CPUs the only additional criterion is to avoid RT tasks (it is better to let them on the RT CPU). Concerning the NRT to RT CPUs migrations, there are two kinds of load-balancing. One is to move back as soon as possible the RT tasks and a second is to use the free cycles of the RT CPUs. Both functions have to predict if a task will soon have to migrate again to a NRT CPU because it requires preemption disabled. We propose to estimate the likelihood of another migration by the frequency of the previous migration attempts: we suppose a task that has not disabled preemption for a long time is less likely to disable it in a close future. The implementation of this prediction can be done by saving the time of the last two migration attempts of each task. From those times, we obtain the time weighted mean of the time elapsed between two attempts. Then the selection criterion do not move tasks if the time since their last migration attempt multiplied by a constant K is smaller than the mean. K has to be specified later following the results of experiments, it is expected to be between 2 and 100.

Pairing policy The original kernel mechanism compares the load of processors simply by using the number of tasks ready to run on each processor (runqueue length). Although this method is sufficient for a system executing nearly exclusively NRT tasks, it gives unexpected results if real-time tasks consume a significant amount of time because they do not share time with lower priority tasks. Therefore, the pairing policy has to be enhanced to take into account, in addition to the runqueue length, the time consumed by RT tasks. In the pairing procedure, we ponderate the runqueue length by $(1 - RT)$, where RT is the ratio of CPU time used by the RT tasks.

4 Conclusion

In this paper, we have described the specific migration types and their requirements for a real-time system based on an asymmetry of the processors. We distinguish load-balancing strategies according to RT and NRT tasks. We also point out that inter-CPU locks must be avoided. We have then presented the techniques on which our solution are based. Using lock-free FIFOs and an trigger policy of type "push", it is possible to implement lock-free load-balancing functions. The real-time aware behavior is achieved via the introduction of new running statistics (elapsed time between two migration attempts, CPU time consumed by RT tasks) and the specialization of the functions according to the migration type.

For now, a first version of ARTiS has been developed. It integrates the migration of the tasks which disable preemption from an RT to NRT CPU via lock-free queues. The normal load-balancing from the RT to NRT CPUs has been deactivated. However, this standard load-balancing from NRT to RT CPUs is still active (otherwise, tasks will never come back to the RT CPUs) and leads to high latencies. In the same time, measurements have been done on a simulated ARTiS system with a static configuration (tasks attached by hand to the processors). A huge improvement of the latencies could be noticed over a normal kernel: 99.999999% of the latencies where under 40μs [4].

The future work will first consist of the full implementation of the proposed algorithms. The existing version of ARTiS will be the starting point for a version including the implementation of the load-balancing functions. Then the major part of the work will be spent on designing

measurement and verification procedures. They will first be used to check and fine-tune the implemented code and later to perform comparisons with other kernels and configurations.

References

[1] Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03*, Nice, France, April 2003.

[2] Pierre Cloutier, Paolo Montegazza, Steve Papacharalambous, Ian Soanes, Stuart Hughes, and Karim Yaghmour. DIAPM-RTAI position paper. In *Second Real Time Linux Workshop*, Orlando, FL, November 2000.

[3] Robert Love. *Linux Kernel Development*. Sams Publishing, August 2003.

[4] Philippe Marquet, Julien Soula, Éric Piel, and Jean-Luc Dekeyser. An asymmetric model for real-time and load-balancing on Linux SMP. Research Report 2004-04, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, April 2004.

[5] Momtchil Momtchev and Philippe Marquet. An asymmetric real-time scheduling for Linux. In *Tenth International Workshop on Parallel and Distributed Real-Time Systems*, Fort Lauderdale, FL, April 2002.

[6] Sillicon Graphics, Inc. REACT: Real-time in IRIX. Technical report, Sillicon Graphics, Inc., Mountain View, CA, 1997.

[7] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.