UNIVERSITÉ DE MONTRÉAL

LOW-IMPACT OPERATING SYSTEM TRACING

MATHIEU DESNOYERS
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION DU DIPLÔME DE
PHILOSOPHIÆ DOCTOR (Ph.D.)
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2009

© Mathieu Desnoyers, 2009.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

LOW-IMPACT OPERATING SYSTEM TRACING

présentée par : DESNOYERS Mathieu
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury constitué de :

Mme. BOUCHENEB Hanifa, Doctorat, présidente
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. BOYER François-Raymond, Ph.D., membre
M. STUMM Michael, Ph.D., membre

I dedicate this thesis to my family, to my friends,
who help me keeping balance between
the joy of sharing my work,
my quest for knowledge
and life.

Je dédie cette thèse à ma famille, à mes amis,
qui m'aident à conserver l'équilibre entre
la joie de partager mon travail,
ma quête de connaissance
et la vie.

Acknowledgements

I would like to thank Michel Dagenais, my advisor, for believing in my potential and letting me explore the field of operating systems since the beginning of my undergraduate studies.

I would also like to thank my mentors, Robert Wisniewski from IBM Research and Martin Bligh, from Google, who have been guiding me through the internships I have done in the industry. I keep a good memory of these experiences and am honored to have worked with them.

A special thanks to Paul E. McKenney, who I found has an insatiable curiosity and never-ending will to teach, discuss and learn. I really enjoyed working with him on the user-space `RCU` implementations.

Thanks to my family, Normand, Hélène and Alexandre for their support through the duration of this research. A special thanks to my friend Etienne Bergeron, who helped me making this thesis better with his thorough reviews and relevant criticism. Thanks to everyone who reviewed this thesis: Nicolas Gorse and Matthew Khouzam. Thanks to the students with whom I shared the laboratory for the discussions, always useful to explore the many faces of a problem. Thanks to the Montréal Art Café staff for feeding the writing of this thesis with their joviality and excellent Café Moka.

Thanks to Google, IBM Research, Ericsson, Autodesk, Natural Sciences and Engineering Research Council of Canada and Defence Research and Development Canada for funding this research.

Thanks to the countless industry collaborators at Fujitsu, Wind River, Nokia, Siemens, Novell and Monta Vista. Thanks to the SystemTAP team for their collaboration, especially to Frank Ch. Eigler.

Finally, I would like to thank the Linux kernel developers community for their constructive criticism and their input. Special thanks to Ingo Molnar, Steven Rostedt, Andrew Morton, Linus Torvalds, Christoph Hellwig, Christoph Lameter, H. Peter Anvin, Rusty Russell, Andi Kleen, Tim Bird, Kosaki Motohiro, Lai Jiangshan, Peter Zijlstra and James Bottomley. I would not have developed such a thorough knowledge of operating systems without all these heated discussions on the Linux Kernel Mailing list.

Abstract

Computer systems, both at the hardware and software-levels, are becoming increasingly complex. In the case of Linux, used in a large range of applications, from small embedded devices to high-end servers, the size of the operating system kernels increases, libraries are added, and major software redesign is required to benefit from multi-core architectures, which are found everywhere. As a result, the software development industry and individual developers are facing problems which resolution requires to understand the interaction between applications and all components of an operating system.

In this thesis, we propose the `LTTng` (*Linux Trace Toolkit next generation*) tracer as an answer to the industry and open source community tracing needs. The low-intrusiveness of the tracer is a key aspect to its usefulness, because we need to be able to reproduce, under tracing, problems occurring in normal conditions. In some cases, users leave tracers active at all times in production, which makes the tracer overhead definitely critical. Our approach involves the design of synchronization primitives that meet the low-impact requirements. The linearly scalable and wait-free `RCU` (*Read-Copy Update*) synchronization mechanism used by the `LTTng` tracer fulfills these requirements with respect to data read. A custom-made buffer synchronization scheme is proposed to extract tracing data while preserving linear scalability and wait-free characteristics.

By measuring the `LTTng` impact, we demonstrate that it is possible to create a tracer that satisfy all the following characteristics: low latency, deterministic real-time impact (wait-free), small impact on operating system throughput and linear scalability with the number of cores. Experiments on various architectures show that this tracer is portable.

We propose a general model for superscalar multi-core systems with weakly-ordered memory accesses to perform formal verification of the `RCU` correctness and wait-free guarantees by model-checking. The `LTTng` buffering scheme is also formally verified for safety and progress. Formal verification demonstrates that these algorithms allow reentrancy from multiple execution contexts, ranging from standard thread to non-maskable interrupts handlers, allowing a wide instrumentation coverage

of the operating system.

Résumé

Les systèmes informatiques, tant au niveau matériel que logiciel, deviennent de plus en plus complexes. En ce qui concerne Linux, système d'exploitation utilisé dans une vaste catégorie d'applications, des petits systèmes embarqués aux serveurs de haut-niveau, la taille des noyaux de systèmes d'exploitations augmente, des librairies sont ajoutées et une réingénierie logicielle est requise pour bénéficier des architectures multi-cœurs, lesquelles sont omniprésentes. Par conséquent, l'industrie du développement logiciel et les développeurs individuels font face à des problèmes dont la résolution nécessite de comprendre l'interaction entre les applications et tous les composants d'un système d'exploitation.

Dans cette thèse, nous proposons le traceur `LTTng` (*Linux Trace Toolkit next generation*) comme réponse aux besoins de traçage de l'industrie et de la communauté du logiciel libre. La faible intrusivité du traceur est un aspect clé menant à son utilisabilité, puisqu'il est nécessaire de reproduire, sous traçage, des problèmes observés sous des conditions d'exécution normales. Dans certains cas, les usagers souhaitent laisser des traceurs actifs en tout temps sur des systèmes en production, ce qui rend l'impact en performance définitivement critique. Notre approche implique l'élaboration de primitives de synchronisation qui rencontrent les requis de faible impact. Le mécanisme de synchronisation permettant la mise à l'échelle et sans attente `RCU` (*Read-Copy Update*) utilisé par le traceur `LTTng` remplit ces requis en ce qui concerne la lecture de données. Nous proposons un mécanisme de synchronisation pour extraire les données de traçage en préservant les caractéristiques de mise à l'échelle linéaire et de non-attente.

En mesurant l'impact du traceur `LTTng`, nous démontrons qu'il est possible de créer un traceur qui satisfait toutes les caractéristiques suivantes : faible latence, comportement temps-réel déterministe (sans attente), faible impact sur le débit du système d'exploitation et une mise à l'échelle linéaire par rapport au nombre de processeurs. Une expérimentation sur plusieurs architectures permet d'observer la portabilité du traceur.

Nous proposons un modèle général pour les systèmes superscalaires multi-cœurs avec accès mémoire faiblement ordonnés pour permettre la vérification formelle des

garanties quant à l'exactitude de l'exécution et l'exécution sans attente à l'aide de la vérification de modèle. Le mécanisme de tampon de `LTTng` est également vérifié formellement quant à son exactitude et son exécution sans attente. La vérification formelle permet également de démontrer que ces algorithmes permettent la réentrance de plusieurs contextes d'exécution, du fil d'exécution standard aux gestionnaires d'interruptions non-masquables, permettant une large couverture d'instrumentation du système d'exploitation.

Condensé en français

La croissance en complexité des systèmes informatiques actuels rend les tâches de déverminage et l'analyse de performance de plus en plus difficiles. Le besoin en outils d'analyse qui tiennent compte de l'ensemble du système se fait sentir, mais leur impact en performance est généralement un obstacle à leur adoption.

Le traçage est une technique éprouvée pour permettre l'étude des interactions entre composants d'un système informatique, mais son impact en performance le rend inutilisable sous plusieurs charges de calcul. Cette recherche tente de rendre le traçage largement utilisable par une vaste catégorie d'applications.

Il importe également pour un traceur de modifier le comportement du système observé de façon minimale afin de permettre la reproductibilité des problèmes observés sans traçage. Il est également important que le traceur n'utilise qu'une fraction des ressources du système et ne modifie son comportement que de façon déterministe afin de permettre une activation du traçage sur des systèmes en production.

Le problème étudié dans ce travail est l'extraction d'information de traçage d'un système d'exploitation, ce qui implique la collecte d'information à partir de l'exécution de ce système d'exploitation et le transfert de cette information hors du noyau. Les aspects de ce problème qui rendent cette étude intéressante sont l'impact du traceur sur le fonctionnement du système et la couverture d'instrumentation : quelles parties du système peuvent êtres instrumentées.

L'hypothèse servant de point de départ à cette étude est qu'il est possible de tracer un système d'exploitation qui exécute une charge de travail élevée sur des ordinateurs multiprocesseurs, en n'utilisant qu'une petite fraction des ressources systèmes, tout en permettant l'instrumentation de n'importe quel site noyau ou usager, rendant ainsi possible la modélisation du comportement original du système. Ceci implique l'utilisation d'une fraction du débit du système ainsi qu'un ajout d'une faible quantité à sa latence moyenne. La conservation des propriétés suivantes du système d'exploitation est recherchée : mise à l'échelle, réponse temps-réel, portabilité et réentrance.

Le but de cette étude est la création d'un traceur à faible impact, hautement réentrant et permettant la mise à l'échelle, pour le noyau d'un système d'exploitation largement utilisé : Linux. Cette infrastructure doit être capable de gérer un débit de

traçage généré par des charges de travail élevées sur des systèmes multiprocesseurs. Celui-ci doit préserver, ou modifier dans une faible proportion, les caractéristiques du noyau Linux.

Les objectifs de recherche sont les suivants :

- rencontrer les requis de traçage de l'industrie et de la communauté du logiciel libre,
- mettre au point de nouveaux algorithmes pour solutionner les problèmes de l'industrie identifiés,
- implanter un traceur pour Linux, un système d'exploitation largement utilisé,
- développer chaque composant du traceur afin que leur combinaison conserve les propriétés de mise à l'échelle et ait un faible impact sur le débit et la latence moyenne du système d'exploitation,
- garantir un impact temp-réel déterministe du traçage,
- obtenir des mécanismes de traçage ayant une portabilité et réentrance accrues.

La contribution scientifique principale de cette recherche est la création de mécanismes de synchronisation adaptés au traçage, incluant : un algorithme de synchronisation de tampons sans attente, pouvant être mis à l'échelle de manière linéaire et supportant les `NMIs` (*interruptions non-masquables*), l'application de techniques d'auto-modification de code pour gérer l'activation d'instrumentation statique de manière efficace, l'amélioration des mécanismes de synchronisation `RCU` en espace usager et la création d'un modèle d'architecture générique pour la vérification formelle d'algorithmes parallèles, modélisant les accès mémoire et l'ordonnancement d'instructions faiblement ordonnés. Ces contributions permettent l'atteinte des objectifs de recherche identifiés.

La méthodologie utilisée pour répondre à ces objectifs se détaille comme suit. Une interaction avec l'industrie et la communauté du logiciel libre permet initialement d'obtenir plus d'information sur le contexte d'utilisation typique et les besoins perçus en entreprise. Des stages chez IBM Research, Google, ainsi qu'une collaboration avec Autodesk et Ericsson, ont permis de mieux comprendre ces besoins. En parallèle avec cette étude de terrain, des prototypes du traceur sont réalisés et proposés aux communautés `LTT` et du noyau Linux afin de bénéficier de leurs commentaires. À travers les phases de son développement, `LTTng` est testé avec des charges de travail extrêmes, et l'utilisation de bancs d'essais en performance permettent de s'assurer que l'impact du traceur se situe dans des limites acceptables. La vérification de modèle est

utilisée afin de vérifier formellement les algorithmes de synchronisation de tampons quant à leur exactitude ainsi que d'assurer l'absence de famine.

Tôt dans le développement, j'ai identifié le besoin de découpler l'instrumentation du traceur, dans le but de permettre à des contributeurs externes et experts de procéder à l'instrumentation de chaque sous-système du noyau. C'est pourquoi j'ai créé les `Kernel Markers` et `Tracepoints` afin de permettre de gérer cette instrumentation et aider à l'ajout d'instrumentation dans le noyau Linux. Ces infrastructures sont maintenant intégrées dans le noyau Linux et utilisées largement par la communauté de développeurs Linux.

Plusieurs prototypes de traceur pour l'espace usager ont également été réalisés au cours de ce projet. Le projet `UST` (*User-Space Tracer*) présentement en cours bénéficie de l'expérience acquise via l'implantation de ces prototypes, réutilisant l'algorithme de synchronisation de tampons de `LTTng` ainsi que les mécanismes de *Kernel Markers* et *Tracepoints*. Un aspect clé de la conception du traceur `LTTng` pour permettre la mise à l'échelle et un faible impact sur les performances est l'utilisation de `RCU` (`Read-Copy Update`) pour la synchronisation de l'accès en lecture aux données de contrôle du traçage. Cependant, puisque ce mécanisme était inexistant en espace usager, nous avons conçu de nouveaux algorithmes permettant à `RCU` d'être utilisé dans ce contexte plus contraint. Ce travail a été effectué en collaboration avec Paul E. McKenney, Alan Stern et Jonathan Walpole. J'ai procédé à la mise en application de ces algorithmes en les implantant dans une librairie de synchronisation `RCU` offrant des services similaires à ceux de l'implantation du noyau Linux.

Vu la complexité des algorithmes de `RCU` et de synchronisation de tampon de `LTTng`, procéder à leur vérification formelle est souhaitable afin d'augmenter le niveau de confiance dans leur implantation. J'ai donc entrepris la tâche de créer un modèle de processeurs avec accès mémoire et exécution faiblement ordonnés, afin d'assurer l'exactitude et le progrès au plus bas niveau, tout en restant assez général pour assurer la portabilité de l'implantation des algorithmes.

Afin de s'assurer que le traceur respecte l'ensemble de ces propriétés, il faut s'attarder à chaque mécanisme qui le composent, tant au niveau du support à l'instrumentation, de la lecture de temps, du contrôle du traçage que de la synchronisation des tampons. Ces mécanismes sont coordonnés par l'exécution de la *sonde* (*probe*). Le schéma fonctionnel de celle-ci se retrouve à la Figure 0.1.

Au niveau de l'instrumentation, j'ai créé les mécanismes de *Tracepoints* et *Linux*

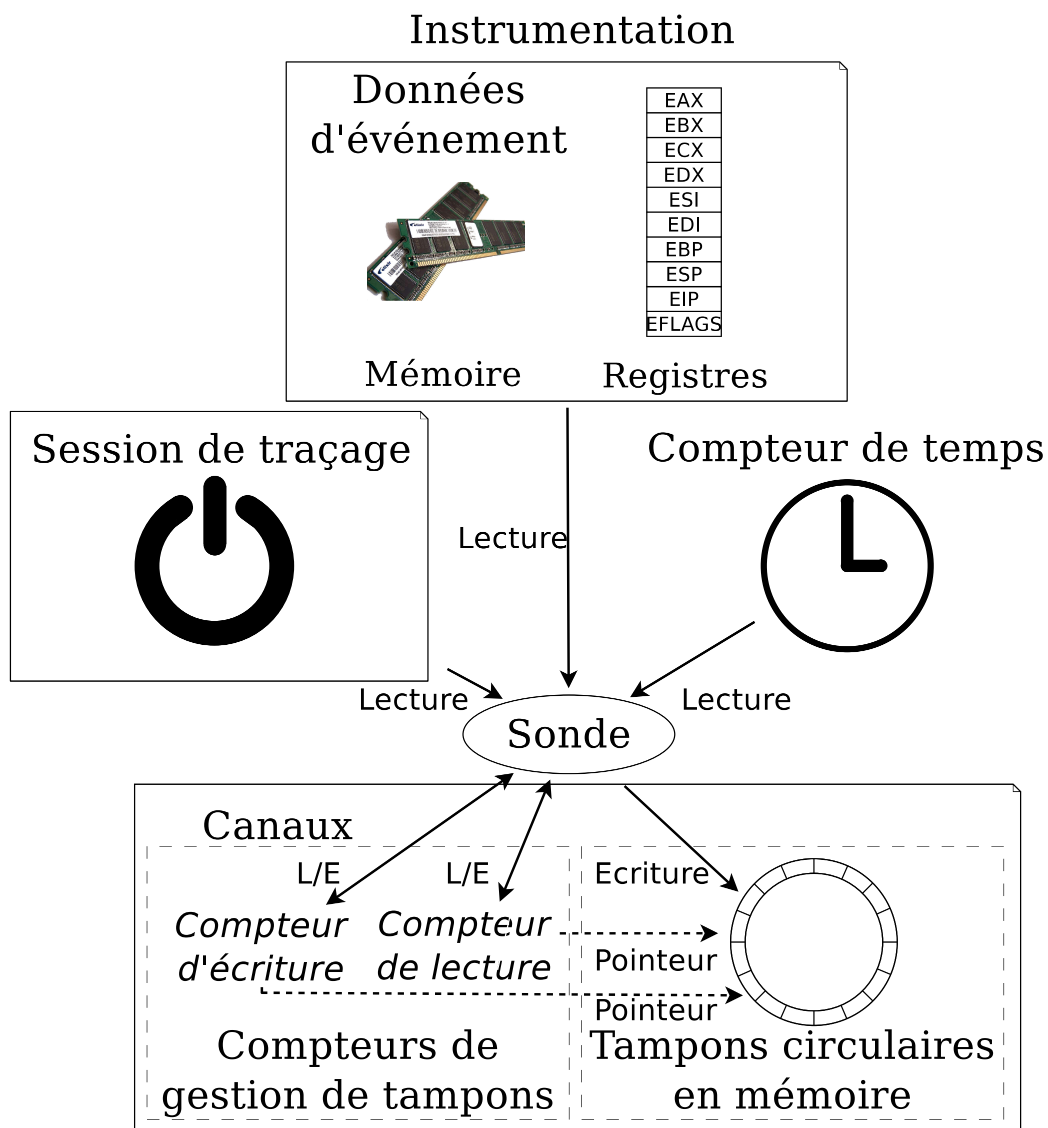

Figure 0.1 Schéma fonctionnel de la sonde LTTng

Kernel Markers. Ceux-ci dépendent du mécanisme de synchronisation `RCU` de manière extensive pour assurer la cohérence des données pointant vers les fonctions de rappels. L'exécution de ces primitives implique donc l'utilisation d'un verrou en lecture `RCU`, lequel est sans attente et pouvant être mis à l'échelle linéairement.

Afin de minimiser l'impact sur l'exécution du noyau Linux lorsque l'instrumentation est désactivée, j'ai créé le mécanisme *Immediate Values*, lequel permet l'activation de branchements en procédant à la modification dynamique d'instructions à l'exécution. Celui-ci se base sur une technique de détournement de l'exécution par *breakpoint* combiné avec une synchronisation des processeurs par interruption inter-processeurs, ce qui permet de conserver la caractéristique de réentrance.

La source de temps utilisée est une lecture directe du compteur de cycles lorsque celui-ci est disponible. Cependant, lorsqu'il est nécessaire de compléter l'information de temps lue du matériel par de l'information complémentaire dans une structure de données en mémoire, il faut également procéder à une synchronisation des accès à cette structure. J'ai donc créé un mécanisme de gestion d'horloge de traçage permettant l' extension d'un compteur de cycles limité à 32 bits vers un compteur 64-bit complet en utilisant une technique basée sur les `RCU`. Ceci permet donc d'accéder à une base de temps sans toutefois perdre les propriétés recherchées, puisqu'un simple verrou en lecture `RCU` est requis.

Les accès aux structures de données de contrôle du traçage sont également synchronisés par le mécanisme de `RCU`, ce qui permet d'altérer le comportement du traçage alors que celui-ci est en cours d'exécution tout en conservant l'ensemble des propriétés recherchées.

Finalement, un composant d'importance majeure, duquel `LTTng` est formé, est le mécanisme de synchronisation de tampons. Celui-ci gère l'écriture de données dans un tampon partagé entre contextes d'exécution. Il est donc impossible d'utiliser un verrou en lecture `RCU` pour cette synchronisation, puisqu'il s'agit ici de synchroniser des écritures.

En s'assurant que chaque composant du traceur respecte les propriétés recherchées, on démontre, par construction, que le traceur dans son ensemble les respecte également.

L'expérimentation réalisée afin de s'assurer du respect des propriétés d'impact sur la mise à l'échelle, le débit, la latence moyenne, la réentrance et le comportement temps-réel est appliquées au mécanisme `RCU` ainsi qu'à la synchronisation de tampons. Ceci permet de s'assurer du respect des propriétés pour chaque composant.

Cette expérimentation utilise deux approches : les bancs d'essai, pour les mesures de performances, et la vérification formelle par model-checking, pour procéder à la vérification des propriétés de réentrance et de comportement temps-réel.

Les bancs d'essais réalisés visent à mesurer l'impact sur la mise à l'échelle, le débit et la latence moyenne du système d'exploitation.

Un premier banc d'essai mesure la différence de débit d'une charge de travail offrant initialement de bonnes caractéristiques de mise à l'échelle avec et sans traçage. Les résultats obtenus sont que le débit augmente de manière linéaire avec et sans traçage de 1 à 8 nœuds, ce qui démontre la préservation de la caractéristique de mise à l'échelle de l'ensemble des mécanismes utilisés par le traceur. Des bancs d'essais spécialisés ont également permis de vérifier la mise à l'échelle linéaire des primitives de synchronisation en lecture `RCU` sur des configurations variant de 1 à 64 nœuds.

Un deuxième ensemble de bancs d'essais vise à déterminer l'impact sur le débit pouvant être soutenu par le système d'exploitation. L'exécution de charges de travail réseau, disque et de calcul mesurant le débit du système d'exploitation ou le temps requis pour effectuer une même tâche permet de caractériser ce débit. La comparaison des mesures avec et sans traçage permet de connaître l'impact du traceur sur ce débit. Dans des conditions d'exécution typique à haute fréquence d'événements, le ralentissement du débit système est, au plus, de 6%. Dans des cas extrêmes, le ralentissement peut atteindre 35%.

Une troisième série de bancs d'essais a pour but de mesurer l'impact du traçage d'un événement sur la latence moyenne du système d'exploitation. Cette série de bancs d'essais se subdivise en deux sous-parties : l'étude de la latence liée au traceur lorsque celui-ci est présent en antémémoire, ainsi que l'étude comparative de latence en fonctionnement normal avec une charge de travail en bruit de fond. Ceci a pour objectif de tenir compte des effets de cache dans l'analyse de l'impact en latence. Sur une architecture Intel Xeon doté d'une fréquence d'horloge de 2.0 GHz, on obtient comme résultats que le traceur a un impact sur la latence de 119 ns par événement lorsque le traceur est présent en antémémoire et de 333 ns par événement pour tracer une charge de travail standard.

Le second plan d'expérimentation est la vérification formelle d'une modélisation des algorithmes `RCU` et de synchronisation des tampons `LTTng`. Ce volet a pour objectif la vérification du bon fonctionnement de ces algorithmes (absence de corruption) ainsi que la vérification du progrès (absence de famine).

L'impact d'algorithmes sur le comportement temp-réel peut se classer selon qu'ils soient bloquants ou non-bloquants. Dans la classe des algorithmes non-bloquants, on retrouve, en ordre croissant de garanties temps-réel, les algorithmes sans obstruction, sans verrou et sans attente. Les catégories d'intérêt dans notre cas sont les algorithmes sans verrou et sans attente. Un algorithme sans verrou assure un progrès global du système, alors qu'un algorithme sans attente assure le progrès de chaque processus considéré dans le système.

La vérification de modèles de synchronisation de tampons `LTTng` impliquant plusieurs écrivains, ainsi que la présence d'écrivains et de lecteurs, permet de confirmer que l'algorithme de synchronisation de `LTTng` est sans attente pour les écrivains. C'est-à-dire que ni un lecteur, ni un autre écrivain concurrent ne peut causer la famine d'un écrivain. Cette propriété s'applique aux implantations noyau, puisqu'il est possible de désactiver l'ordonnanceur. Il est également démontré que l'impossibilité de désactiver l'ordonnanceur en espace usager élimine la garantie de type "sans attente", rendant une telle implantation seulement "sans verrou", puisqu'il devient possible pour un écrivain concurrent de causer la famine d'un autre écrivain.

La vérification d'un modèle adapté aux primitives de synchronisation `RCU` permet de s'assurer qu'un lecteur `RCU` est également sans attente.

L'impact sur la réentrance du traceur est vérifié à l'aide des modèles d'exécution de processus concurrents utilisés pour vérifier le comportement correct et l'absence d'interblocage lors de l'exécution de paires de processus. Ceux-ci couvrent les cas d'exécution de multiples interruptions entre deux étapes atomiques, ce qui permet d'assurer l'absence d'interblocage dans tous les cas. Ces mêmes modèles sont utilisés pour vérifier l'absence de corruption causée par des accès concurrents des tampons en écriture et lecture.

Des modèles similaires sont utilisés pour vérifier les propriétés des mécanismes de synchronisation `RCU`. Les résultats démontrent que tant `RCU` que le mécanisme de synchronisation d'accès aux tampons de `LTTng` supportent la réentrance de gestionnaires d'interruptions `NMI`.

Afin d'augmenter la précision du modèle `RCU` quant à la représentation des effets d'accès mémoire et d'exécution faiblement ordonnés des architectures modernes, nous avons créé deux modèles d'architectures : *OoOmem* et *OoOisched*. Ceci permet de vérifier que les barrières mémoires ainsi que les barrières restreignant les optimisations du compilateur sont adéquates en vérifiant toutes les traces possibles d'exécution

respectant les contraintes exprimées dans le modèle.

Le déploiement et l'expérimentation sur les architectures Intel 32/64-bit, PowerPC 32/64-bit, ARM, MIPS, Sparc64 ainsi que les contributions externes pour SH, S/390 ainsi que S/390x permettent de confirmer la portabilité de `LTTng`. Les prototypes de traçage rapide en espace usager et pour l'hyperviseur Xen que j'ai réalisés démontrent également la portabilité et la réutilisabilité des algorithmes créés dans des environnements d'exécution variés.

Puisque chacune des composantes satisfait les propriétés recherchées, que ce soit en utilisant `RCU` pour la synchronisation ou bien le mécanisme de synchronisation de tampons pour l'écriture, nous pouvons affirmer, par construction, que le traceur respecte toutes les propriétés recherchées.

La satisfaction de ces propriétés ouvre des champs d'utilisations diversifiés. La discussion qui suit fait le lien entre chacune des propriétés et les champs d'applications ciblés. Les propriétés de faible impact sur les performances (latence, débit et mise à l'échelle linéaire) ciblent les serveurs commerciaux qui nécessitent une surveillance constante du bon fonctionnement des systèmes en production utilisant Linux. Ceci permet à des compagnies comme Google d'identifier plus aisément la cause de problèmes de fonctionnement et de performance. Cependant, pour laisser un traceur actif en tout temps sur ces serveurs opérationnels en production, il faut avoir un impact minimal sur leurs performances. Des besoins similaires sont évoqués dans les applications à contraintes de temps douces chez Autodesk, lequels utilisent actuellement `LTTng`. Ericsson collaborent pour rendre `LTTng` utilisable dans leurs systèmes de télécommunication. Les développeurs de Siemens dépendent également, de leur côté, de `LTTng` pour certains de leurs produits sous Linux.

Les garanties temps-réel permettent également au domaine des systèmes temps-réel d'utiliser le traceur `LTTng` pour des fins de déverminage et d'analyse des systèmes comportant des contraintes de temps. C'est ce que font actuellement les distributions Linux de Wind River, Monta Vista et STLinux en intégrant `LTTng` comme outil de traçage à leur distribution visant le marché temps-réel. Ils permettent ainsi à leurs utilisateurs d'obtenir une information sur le comportement du système d'exploitation similaire à ceux retrouvés dans les autres systèmes d'exploitation temps-réels usuels. La portabilité du traceur `LTTng` permet son intégration dans la plate-forme de développement Maemo pour les téléphones cellulaires et tablettes Internet de Nokia.

La réentrance accrue du traceur bénéficie à la communauté de développeurs Linux,

leur permettant d'ajouter des points d'instrumentation sans se soucier de l'interaction du traceur avec le site instrumenté. Ceci bénéficie également aux utilisateurs Linux, les assurant que leur noyau ne cessera pas de fonctionner lors qu'ils activent le traceur. Ceci leur permet d'avoir une infrastructure de traçage en laquelle ils peuvent avoir confiance.

Toutes ces applications de `LTTng` démontrent qu'il remplit un besoin de traçage noyau à faible impact dans plusieurs champs d'application de l'industrie. Une conséquence de cette recherche est donc d'améliorer les infrastructures de déverminage pour les systèmes multi-cœurs, rendant disponible un outil qui permet de trouver les goulots d'étranglement de performance. Ceci permet une accélération des applications en trouvant toutes sortes d'utilisations inefficaces des ressources, ce qui aide à améliorer le temps de réponse, la réponse temps-réel, le débit système ainsi que l'efficacité énergétique des systèmes.

Cette recherche a eu des impacts autres que ceux directement reliés au traceur `LTTng`. Les *Local Atomic Operations*, *Kernel Markers* et *Tracepoints* ont chacun contribué à d'autres champs d'application et d'autres projets. Le traçage de l'espace usager, bien que périphérique à la cible principale de cette recherche, a été un champs dans lequel nous avons également innové. Au niveau de la contribution scientifique, 5 articles provenant d'auteurs externes ayant utilisé `LTTng` dans leur expérimentation ont été répertoriés à ce jour. Dans ces articles, `LTTng` a été utilisé à des fins variées, en permettant l'analyse de la consommation de puissance de pilotes, l'analyse des habitudes des utilisateurs et l'analyse de la précision des sources de temps.

La principale réalisation de cette recherche est la création d'algorithmes de synchronization novateurs rendant possible l'implantation du traceur `LTTng` pour le système d'exploitation Linux. Ce traceur satisfait les propriétés de faible impact sur la mise à l'échelle, le débit et la latence moyenne du système d'exploitation, d'impact déterministe sur la réponse temps-réel, de portabilité vers des architectures variées et un haut niveau de réentrance. Des bancs d'essais en performance et la vérification formelle ont démontré que chaque composant du traceur satisfait ces propriétés. Ainsi, le traceur `LTTng` répond à des requis auxquels ses prédécesseurs ne répondaient que partiellement, ce qui rend possible le traçage du système d'exploitation Linux, dont la flexibilité permet son utilisation dans un large spectre de champs d'application.

Il a été possible d'atteindre ces objectifs en procédant à un choix parcimonieux, à l'élaboration et à l'implantation de mécanismes de synchronisation : `RCU` pour la syn-

chronisation en lecture et le mécanisme de synchronisation de tampons de `LTTng` pour l'écriture. Les algorithmes utilisés dans ces deux mécanismes garantissent une mise à l'échelle linéaire et la non-attente, caractéristiques utiles pour le traçage de systèmes multi-cœurs ainsi que pour assurer un comportement temps-réel déterministe et une réentrance complète.

La réponse appropriée aux requis de traçage de l'industrie et de la communauté du logiciel libre est démontrée par le fait que divers composants de traçages que nous avons créé, les *Tracepoints* et *Linux Kernel Markers*, sont intégrés au noyau Linux et que le traceur `LTTng` bénéficie d'une large communauté d'utilisateurs et contributeurs, en particulier Google, IBM, Ericsson, Autodesk, Wind River, Fujitsu, Monta Vista, STMicroelectronics, C2 Microsystems, Sony, Siemens, Nokia et Recherche et développement pour la défense Canada.

En conclusion de cette recherche, nous pouvons affirmer que le traçage de charges de travail importantes sur un système d'exploitation à usage général s'exécutant sur des architectures multi-cœurs peut être accompli en n'utilisant qu'une fraction du débit et en n'augmentant la latence du système que faiblement, tout en conservant complètement la mise à l'échelle, la réponse temps-réel, la portabilité et la réentrance du système d'exploitation. L'implantation réalisée permet une couverture d'instrumentation du noyau dans son ensemble, incluant les gestionnaires d'interruptions non-masquables (`NMIs`).

L'analyse de traces du système dans son ensemble implique la collecte de traces à partir tant du noyau que de l'espace usager. Suite aux résultats prometteurs des prototypes de traçage usager réalisés, il est maintenant temps de stabiliser une telle infrastructure afin de permettre un traçage usager d'un niveau utilisable en production sous Linux.

Les charges de travail qui peuvent maintenant êtres tracées sur des systèmes en production permet la collecte d'information menant à l'analyse et la résolution de problèmes en comportement et en performance dans les systèmes complexes actuels. It sera intéressant d'explorer les analyses rendues possibles par une modélisation du système d'exploitation orientée par les données extraites par `LTTng`.

Vu son utilité dans le cadre de l'observation de systèmes, de l'identification de goulots d'étranglement en performance et en déverminage, la décision d'activer un traceur en tout temps sur des systèmes en production devient naturelle pour les développeurs système *si* la pénalité en performance est assez faible. Cette recherche

démontre clairement que l'impact du traceur `LTTng`, lorsqu'activé, est assez faible pour permettre son utilisation sur des systèmes en production faisant face à des charges de travail élevées, sans pour autant diminuer les performances de manière prohibitive.

Contents

Dedication . . . iii

Acknowledgements . . . iv

Abstract . . . v

Résumé . . . vii

Condensé en français . . . ix

Contents . . . xx

List of Tables . . . xxv

List of Figures . . . xxvii

List of Algorithms . . . xxix

List of Signs and Abbreviations . . . xxx

Chapter 1 Introduction . . . 1
1.1 Theoretical Framework . . . 1
1.2 Problem . . . 3
1.3 Hypothesis . . . 4
1.4 Objectives . . . 4
1.5 Claim for Originality . . . 4
1.6 Outline . . . 5

Chapter 2 State of the Art . . . 7
2.1 Computer Architecture . . . 7
2.1.1 Parallelism . . . 7
2.1.2 Memory Access . . . 8
2.1.3 Software-Level Support for Multiprocessing . . . 9
2.2 Real-time . . . 10

2.3 Distributed Systems . . . 11
2.4 Commercial Servers . . . 12

Chapter 3 Methodology . . . 13
3.1 Interaction with the community . . . 14
3.1.1 Tracing in the Industry . . . 15
3.1.2 Tracing in the Open-Source Community . . . 16
3.1.3 Authored publications . . . 16
3.1.4 Co-authored publications . . . 17
3.2 Tracer Architecture . . . 17
3.2.1 Overview . . . 17
3.2.2 Instrumentation . . . 19
3.2.3 Synchronization Primitives . . . 21
3.2.4 Buffering . . . 21
3.2.5 Read-Copy Update . . . 22
3.3 Experimentation . . . 23
3.3.1 Benchmarks . . . 23
3.3.2 Formal Verification . . . 24

Chapter 4 Paper 1: Synchronization for Fast and Reentrant Operating System Kernel Tracing . . . 25
4.1 Abstract . . . 25
4.2 Introduction . . . 25
4.3 Introduction to Tracing . . . 27
4.4 State of the Art . . . 28
4.5 Linux Trace Toolkit Next Generation . . . 34
4.6 Tracing Synchronization . . . 35
4.6.1 Atomic Primitives . . . 35
4.6.2 Recursion with the Operating System Kernel . . . 38
4.6.3 Timekeeping . . . 39
4.7 Benchmarks . . . 46
4.8 Least Priviledged Execution Contexts . . . 50
4.9 Conclusion . . . 50

Chapter 5 Paper 2: Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing . . . 57
5.1 Introduction . . . 58
5.2 State of the art . . . 59
5.3 Design of LTTng . . . 62
5.3.1 Components overview . . . 63
5.3.2 Channels . . . 65
5.3.3 Control . . . 66
5.3.4 Probe Data Flow . . . 67
5.4 Atomic Buffering Scheme . . . 69
5.4.1 Atomic data structures . . . 70
5.4.2 Equations . . . 72
5.4.3 Algorithms . . . 75
5.4.4 Memory Barriers . . . 83
5.4.5 Buffer allocation . . . 85
5.5 Experimental results . . . 86
5.5.1 Methodology . . . 86
5.5.2 Probe CPU-cycles overhead . . . 87
5.5.3 tbench . . . 88
5.5.4 Scalability . . . 89
5.5.5 dbench . . . 89
5.5.6 lmbench . . . 91
5.5.7 gcc . . . 92
5.5.8 Comparison . . . 93
5.6 Conclusion . . . 95

Chapter 6 Paper 3: User-Level Implementations of Read-Copy Update . . . 97
6.1 Introduction . . . 97
6.2 Brief Overview of RCU . . . 99
6.2.1 Conceptual View of RCU Algorithms . . . 99
6.2.2 User-Space RCU Desiderata . . . 101
6.2.3 RCU Deletion From a Linked List . . . 102
6.2.4 Overview of RCU Semantics . . . 104
6.3 User-Space RCU Usage Scenarios . . . 106

6.4 Classes of RCU Implementations 107
6.4.1 Notation 107
6.4.2 Quiescent-State-Based Reclamation RCU 109
6.4.3 General-Purpose RCU 112
6.4.4 Low-Overhead RCU Via Signal Handling 117
6.4.5 Wait-Free RCU Updates 123
6.5 Experimental Results 124
6.5.1 Scalability 125
6.5.2 Read-Side Critical Section Length 126
6.5.3 RCU Grace-Period Batch Calibration 128
6.5.4 Update Overhead 129
6.6 Conclusions 132
Acknowledgement 134
Legal Statement 134

Chapter 7 Paper 4: Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms 135
7.1 Introduction 135
7.2 Modeling Challenges 136
7.3 Modeling and Model-Checking 137
7.3.1 LTL Model-Checking 137
7.3.2 Introduction to Parallel Algorithm Modeling 139
7.4 Weakly-Ordered Memory Framework 142
7.4.1 Architecture 142
7.4.2 Testing 145
7.5 Out-of-Order Instruction Scheduling Framework 146
7.5.1 Architecture 146
7.5.2 Testing 149
7.6 Read-Copy Update Algorithm Modeling 155
7.6.1 State-Space Compression 160
7.6.2 RCU Model-Checking Results 161
7.6.3 RCU Verification Discussion 166
7.7 Framework Discussion 167
7.8 Conclusion 168

Acknowledgement 169
Legal Statement 169

Chapter 8 Complementary Results 170
8.1 Kernel Markers 170
8.2 Tracepoints 171
8.3 Immediate Values 172
8.4 Analysis of LTTng Latency Impact 173
8.5 Analysis of LTTng Real-Time Impact 175
8.6 Formal verification of LTTng 177
8.6.1 Modeling 178
8.6.2 Correctness 179
8.6.3 Real-Time Impact 180
8.7 Reentrancy 182

Chapter 9 General Discussion 184
9.1 Tracer Properties 184
9.2 Tracer Application Domains 185
9.3 Contributions 187
9.3.1 Local Atomic Operations 187
9.3.2 Kernel Markers 188
9.3.3 Tracepoints 188
9.3.4 Fast User-space Tracing 189
9.4 Scientific Studies Using LTTng 190

Chapter 10 Conclusion and Recommendations 191

List of References 193

List of Tables

Table 4.1 Benchmark comparison between locking primitives and added inner operations, on Intel Xeon E5405 52
Table 4.2 Cycles taken to execute `CAS` compared to interrupt disabling . 53
Table 4.3 Breakdown of cycles taken for spin lock disabling interrupts . 54
Table 4.4 Breakdown of cycles taken for using a read *seqlock* and using a synchronized `CAS` 55
Table 4.5 Breakdown of cycles taken for disabling preemption and using a local `CAS` 56
Table 4.6 Speedup of tracing synchronization primitives compared to disabling interrupts and spin lock 56
Table 5.1 Cycles taken to execute a `LTTng` 0.140 probe, Linux 2.6.30 . . 87
Table 5.2 `tbench` client network throughput tracing overhead 88
Table 5.3 `tbench` localhost client/server throughput tracing overhead . . 89
Table 5.4 `dbench` disk write throughput tracing overhead 90
Table 5.5 Linux kernel compilation tracing overhead 93
Table 5.6 Comparison of lockless and interrupt disabling `LTTng` probe execution time overhead, Linux 2.6.30 94
Table 7.1 General-purpose `RCU` verification results for the Alpha architecture 162
Table 7.2 General-purpose `RCU` verification results for the Intel/PowerPC architectures 163
Table 7.3 Signal-based `RCU` verification results for the Alpha architecture 163
Table 7.4 Signal-based `RCU` verification results for the Intel/PowerPC architectures 164
Table 7.5 General-purpose `RCU` signal-handler reader nested over reader verification (no instruction scheduling) 165
Table 7.6 General-purpose `RCU` signal-handler reader nested over updater verification (no instruction scheduling) 165
Table 8.1 Tracer latency overhead for a ping round-trip. Local host, Linux 2.6.30.9, 100,000 requests sample, at 2 ms interval . . . 174

Table 8.2 Tracer latency overhead for a ping round-trip. 100 Mb/s network, tracing receiver host only, Linux 2.6.30.9, 100,000 requests sample, at 2 ms interval 175

List of Figures

Figure 0.1 Schéma fonctionnel de la sonde LTTng xii
Figure 3.1 Tracer architecture diagram 18
Figure 4.1 Trace clock read (*no* 32nd bit overflow) 42
Figure 4.2 Trace clock read (32nd bit overflow) 42
Figure 4.3 Trace clock update (1, 3, 4) interrupted by a read (2) 42
Figure 4.4 Synthetic clock read-side 43
Figure 4.5 Synthetic clock periodic update 45
Figure 4.6 Assembly listings for Intel Xeon benchmarks (`CAS` loop content) 52
Figure 4.7 Assembly listings for Intel Xeon benchmarks (interrupt save/restore loop content) 53
Figure 4.8 Assembly listings for Intel Xeon benchmarks (spin lock loop content) 54
Figure 4.9 Assembly listings for Intel Xeon benchmarks (sequence lock and preemption disabling loop content) 55
Figure 5.1 Tracer components overview 64
Figure 5.2 Channel components 66
Figure 5.3 Probe data flow 68
Figure 5.4 Producer-consumer synchronization 71
Figure 5.5 Write and read counter masks 74
Figure 5.6 Commit counter masks 75
Figure 5.7 Impact of tracing overhead on localhost `tbench` workload scalability 90
Figure 6.1 Linux-kernel usage of RCU 99
Figure 6.2 Schematic of RCU grace period and read-side critical sections 100
Figure 6.3 RCU linked-list deletion 103
Figure 6.4 RCU read side using quiescent states 108
Figure 6.5 RCU update side using quiescent states 110
Figure 6.6 RCU read side using memory barriers 113
Figure 6.7 RCU update side using memory barriers 114
Figure 6.8 RCU read side using signals 118

Figure 6.9	RCU signal handling	119
Figure 6.10	RCU update side using signals	121
Figure 6.11	Avoiding update-side blocking by RCU	123
Figure 6.12	Read-side scalability of various synchronization primitives, 64-core POWER5+	125
Figure 6.13	Read-side scalability of mutex and reader-writer lock, 64-core POWER5+	126
Figure 6.14	Impact of read-side critical section length, 8-core Intel Xeon, logarithmic scale	127
Figure 6.15	Impact of read-side critical section length, 8 reader threads on POWER5+, logarithmic scale	128
Figure 6.16	Impact of read-side critical section length, 64 reader threads on POWER5+, logarithmic scale	129
Figure 6.17	Impact of grace-period batch-size on number of update operations, 8-core Intel Xeon, logarithmic scale	130
Figure 6.18	Impact of grace-period batch-size on number of update operations, 64-core POWER5+, logarithmic scale	130
Figure 6.19	Update overhead, 8-core Intel Xeon, logarithmic scale	131
Figure 6.20	Impact of pointer exchange on update overhead, 8-core Intel Xeon, logarithmic scale	132
Figure 6.21	Update overhead, 64-core POWER5+, logarithmic scale	133
Figure 6.22	Impact of pointer exchange on update overhead, 64-core POWER5+, logarithmic scale	134
Figure 7.1	Promela model for PowerPC spinlock	140
Figure 7.2	Diagram representation of PowerPC spinlock model	141
Figure 7.3	Diagram representation of Intel ticket spinlock model	143
Figure 7.4	Out-of-order instruction scheduling and memory frameworks promela test code, Processor A	151
Figure 7.5	Instruction dependencies of out-of-order instruction scheduling and memory framework test	152
Figure 7.6	Instruction dependencies of out-of-order instruction scheduling and memory framework test (error injection)	153
Figure 7.7	Schematic of RCU grace period and read side critical sections	157

List of Algorithms

Algorithm 5.1 TryReserveSlot(*payload_size*) 77
Algorithm 5.2 ReserveSlot(*payload_size*) 78
Algorithm 5.3 CommitSlot(*slot_size, slot_offset*) 78
Algorithm 5.4 ForceSwitch() 80
Algorithm 5.5 ReadPoll() 81
Algorithm 5.6 ReadGetSubbuf() 82
Algorithm 5.7 ReadPutSubbuf(*arg_read_count*) 82
Algorithm 5.8 AsyncWakeupReadersTimer() 84

List of Signs and Abbreviations

API	Application Programming Interface
BIOS	Basic Input/Output System
BDD	Binary Decision Diagram
CAS	Compare-And-Swap
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DFA	Deterministic Finite Automaton
GNU	GNU's Not Unix
GPL	GNU General Public License
GPS	Global Positioning System
HTM	Hardware Transactional Memory
IBM	International Business Machines
IRQ	Interrupt Request
IPI	Interprocessor Interrupt
I/O	Input/Output
KVM	Kernel-based Virtual Machine
LGPL	GNU Lesser General Public License
LTL	Linear Temporal Logic
LTTng	Linux Trace Toolkit Next Generation
LTTV	Linux Trace Toolkit Viewer
Mutex	Mutual exclusion
MPI	Message Passing Interface
NAND	Not AND
NMI	Non-Maskable Interrupt
NOR	Not OR
NUMA	Non-Uniform Memory Access
OS	Operating System
POSIX	Portable Operating System Interface for Unix
QSBR	Quiescent-State Based Reclamation
RAM	Random Access Memory

RCU	Read-Copy Update
RT	Real-Time
RTAI	Real-Time Application Interface
RISC	Reduced Instruction Set Computer
RP	Relativistic Programming
RPC	Remote Procedure Call
SMP	Symmetric Multiprocessing
SPE	Synergistic Processing Element
STM	Software Transactional Memory
TLS	Thread-Local Storage
UST	User-Space Tracer

Chapter 1

Introduction

Growth in hardware and software complexity that industry of software development must cope with now turns debugging and performance monitoring into increasingly challenging tasks. The need for global system-wide problem diagnosis facilities emerges, but their associated high performance impact is usually frowned upon. This compromises their deployment on production systems, where processor time and input/output bandwidth must be fully utilized by the workload.

This increase in complexity and size has been a steady trend in computers since their appearance. At the hardware-level, multiprocessor systems become increasingly large and interconnection buses become more complex. At the software-level, operating system (OS), library, virtual machine and application layers contribute to this complexity. As a result, the effective complexity that software developers have to deal with, represented by the number of interactions between components, grows even faster.

1.1 Theoretical Framework

Debugging and performance monitoring tools are needed to provide insight into component interactions, to help diagnosing issues in those large complex systems. Tracing provides insight into interactions within system components. This technique can diagnose the most difficult performance problems and bugs in software development. It can provide both high-level and detailed views of the operating system (OS) as a whole and of each of its individual subsystems. It consists in collecting event records associated with time stamps, allowing to reconstruct the trace (a sequence of events) in the order events occurred. The major drawback of this technique is its heavy use of computer resources. It provides detailed system activity information, at the expense of system resources required to extract large amounts of data. As an observer, the challenge for a tracer is to minimally alter the original system behavior,

hence permitting its reproducibility when the observer is active. This technique is therefore inherently limited by the portion of system resources required to perform tracing. Using faster computers does not solve the problem: newer architectures will be expected to execute workloads faster and will create new timing conditions by allowing execution of larger workloads. As a result, faster hardware does not improve tracer's ability to trace workloads, because the workload grows to fully use the available resources, mostly due to the evolution of application complexity. When supplementary computational resources are available, they will be allocated to the system's primary purpose rather than tracing.

This research is conducted within the GNU/Linux open-source operating system, which becomes increasingly popular in the industry. It is used on a wide variety of computer systems, from small embedded devices to large multiprocessor servers. Supporting a broad hardware diversity makes it subject to subtle compatibility breakages with specific hardware combinations. In order to deal with the complexity of software development, the operating system is the fruit of interaction of many individuals and companies part of an active developer community. Due to the increasing complexity of the operating system, the people who understand it globally are very rare. Even kernel developers specialize in areas (e.g. memory management, thread scheduling, block device layer, ...). Solving complex subsystem interaction problems therefore becomes a hard problem not only for the Linux users, but also for its own developer community.

This research addresses the need for system-wide problem diagnosis tools, grown from the increased hardware and software complexity. It focuses on the Linux operating system, a candidate already heavily used by the industry and presenting the complexity characteristics requiring such tools. The software development industry currently lacks the appropriate tools to diagnose problems requiring system-wide observability of the system. Tracing fills this need due to some of its unique features. It permits diagnosing performance problems and bugs introduced by the interactions between execution layers and different subsystems. Furthermore, a post-processing analysis approach, where all collected information is made available for a posteriori analysis, is elected to permit diagnosing bugs occurring rarely, which are amongst the hardest to identify.

1.2 Problem

Tracing the Linux operating system brings many challenges, which define the major problems inherent to tracing. These challenges are caused by the need for a system observer that minimally impact the system behavior.

As an initial point, tracer CPU usage must be kept low, so the majority of the CPU time can still be available for the traced workload. Also, exporting data outside of its producing execution context involves consuming bandwidth at many levels: various levels of caches, memory, and optionally disk and network input/output. This bandwidth consumption must also be kept within fractions of the amount of bandwidth used by the operating system traced.

Characteristics met by the operating system, such as support for real-time (increasingly supported by Linux Preempt-RT tree) and scalability to large number of processors (4096) must also be supported by a tracer aiming to meet the requirements faced by this operating system.

At the processor level, reentrancy of tracer operations with respect with concurrent execution contexts is required to ensure coherent data is collected. Moreover, data gathered from multiple processors must be synchronized to keep event ordering information for a posteriori analysis.

Instrumented parts of the operating system are used as calling sites to execute tracer code. This means that instrumentation coverage is directly limited by reentrancy of the tracer code. For example, instrumentation of kernel code executed from non-maskable interrupt (NMI) context require the tracer to be reentrant with respect to these interrupts.

Given that the kernel code executed, and thus the tracer usage, directly depends on the type of workload, it makes sense to principally consider the tracer impact on workloads representing various typical heavy system usage.

The problem studied in this work is the extraction of tracing data from an operating system, which involves collecting data from the operating system execution and streaming this information outside of the kernel. The aspects of the problem making its study worthwhile are the impacts of tracing (workload disruption) and the instrumentation coverage: which parts of the system can be instrumented.

1.3 Hypothesis

The hypothesis serving as starting point for this research is that it is possible to trace an operating system, which runs intensive workloads on large multiprocessor machines, using a small fraction of the resources, allowing to instrument any kernel and user-space code location and permitting to model the original workload behavior. This involves using a fraction of the operating system throughput and adding a constant small amount to its average latency, as well as preserving the following properties: scalability, real-time response, portability and reentrancy.

1.4 Objectives

The purpose of this study is to create a low-overhead, highly-reentrant and scalable tracer for Linux, a widely used operating system. It must be able to handle trace throughput generated by heavy workloads on multiprocessor systems. It must preserve, or modify within a small proportion, the Linux kernel characteristics.

The research objectives are detailed as follow:

- meet the industry and open source community tracing requirements,
- create new algorithms to solve the problems identified in industry,
- implement a tracer for Linux, an existent mainstream operating system,
- develop each tracer component so the tracer meets properties of preserving scalability and having a low-impact on the operating system throughput and average latency,
- guarantee a deterministic impact of tracing on real-time response,
- provide high portability and reentrancy of tracer mechanisms.

1.5 Claim for Originality

The main scientific contribution of this research is the creation of original synchronization algorithms suitable for tracing. The new algorithms created and the application of these algorithms to tracing is detailed as follows:

- creation of a wait-free, linearly scalable, and `NMI`-safe kernel buffering scheme,
- creation of an `RCU`-based trace clock permitting to atomically update current time information, which is larger than a word, atomically with respect to the

reader execution contexts, thus offering a wait-free, linearly scalable and `NMI`-safe time-source,

- design of a complete kernel tracer based on wait-free, linearly scalable and `NMI`-safe algorithms,
- application of self-modifying code techniques to dynamically enable and disable static instrumentation, with non-measurable overhead when disabled and low overhead when enabled, allowing `NMI` context instrumentation (presented in Section 8.3),
- improvements to the `RCU` (*Read-Copy Update*) synchronization mechanisms for efficient execution in user-space context, specifically proposing:
  1. using signal-handlers to execute memory barriers only when waiting for a grace period, allowing fast read-side,
  2. using `TLS` (*Thread-Local Storage*) to perform local `RCU` state access efficiently for reader threads.
  3. chaining `TLS` data within a doubly linked-list to create a reader thread registry, allowing to detect quiescent state with $O(1)$ thread registration and unregistration,
- creation of a generic architecture model for formal verification of parallel algorithms, modeling weakly-ordered memory accesses and instruction scheduling,

The careful design of synchronization primitives enables the creation of the `LTTng` tracer and of its components.

1.6 Outline

The state of the art is presented in Chapter 2, which focuses on the high-level aspects of tracing. The state of the art refers to each article's state-of-the-art section for in-depth per-subject literature review. Each contains a detailed review for its specific topic.

The methodology organizing this research is presented in Chapter 3. It explains how the tracing requirements were first identified by collaborating with the industry. It presents the tracer architecture developed to fulfill these requirements. The instrumentation requirements are then detailed, referring to Chapter 8 for the detailed presentation of this work not part of the four main articles. It then depicts an

overview of the four articles.

The core of this thesis consists in four articles. Synchronization primitive choices for the kernel tracer are presented in the article "Synchronization for Fast and Reentrant Operating System Kernel Tracing" [1], in Chapters 4. This article is currently re-submitted after revision to Software – Practice and Experience. The reviewers recommended its publication, although acceptation depends on approval of modifications performed in the re-submitted version. The presentation of a lock-less buffering scheme using the results of the first article follows in the article "Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing" [2] in Chapter 5. It is currently submitted to ACM Transactions on Computer Systems (TOCS). Then, our implementations of the `RCU` (Read-Copy Update) synchronization mechanism for user-space are presented in the article "User-Level Implementations of Read-Copy Update" [3], in Chapter 6. This article is currently submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS). An architecture-level model for formal parallel algorithm verification is presented in "Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms" [4], in Chapter 7. This last article is also currently submitted to IEEE TPDS.

Chapter 8 presents additional results and discusses impacts of this research which are not presented in the four main articles. This includes conference publications, contributions to open source projects and external work based on the research results.

Chapter 9 discusses the results obtained from a high-level perspective. It shows links between the main results obtained in the individual articles.

Finally, the conclusion recall the progress realized, the principal research contributions and ends by proposing future work leveraging this research.

Chapter 2

State of the Art

This chapter first presents the major computer architecture aspects which impact the current research. Aiming at portability to different architectures implies taking into account multiprocessor system effects. Then, the main tracing systems available today will be classified in categories related to their target systems: parallel, distributed or real-time systems.

The actual usage of these systems is less strictly compartmented. Parallel and distributed systems have been traditionally used for scientific and engineering purposes. However, commercial uses have surpassed these fields in terms of market volume [5]. Real-time systems, on the other hand, have evolved somewhat separately from high-performance systems, filling the embedded system market. This trend is however changing, as these same tools become increasingly useful to observe high-throughput, low-latency systems, which are now required for applications such as stock exchange and search engines, which must deal with high volume of transactions having low-latency requirements. This forms the fourth category: commercial servers.

2.1 Computer Architecture

At one point, there was a large dissimilarity between general-purpose hardware and high-end multiprocessor computers. However, as commodity computers become increasingly multi-core and multi-processor, the scalability concerns become pervasive.

2.1.1 Parallelism

Between 1990 and 2009, the parallelism level of general-purpose processors has increased significantly. From 1990 to 2000, instruction-level parallelism (pipelined and superscalar architectures [6]) increased the instruction execution throughput while still allowing a sequential programming paradigm, by allowing instruction execution

to complete in an order different from which they were issued as long as data dependencies are met.

During this phase, the complexity of operating systems grew to allow multiple tasks to share resources. Amongst them: the processor, memory, devices and shared libraries [7].

Before 2000, multiprocessor systems were less common in general-purpose computers. They were mostly used for servers and mainframes, which ran specialized applications. Given the important market represented by these high-end computers, most operating systems were designed to support multiple processors. Then, the 2000-2009 period showed an increasing availability of multiprocessor systems in commodity hardware and an increasing use of such hardware in distributed systems, principally due to commercial applications [5, 8, 9, 10].

Hence, in 2009, an application must be multiprocessor-aware to fully use the computing power of today's commodity hardware present in both general-purpose computers and distributed systems. The same applies to newer multi-core embedded systems such as the ARM11 MPCORE [11], which indicates that multiprocessing is increasingly pervasive. However, this comes at the cost of additional complexity.

2.1.2 Memory Access

In recent years, we have also seen the gap between the available processing power and memory bandwidth widen, the former evolving much faster than the latter, as presented in [12].

Various architectural solutions have been deployed to deal with this kind of problem. The addition of an increasing number of cache layers being the primary method to help data accesses keeping up with the processor speed. Level-1, level-2 and level-3 caches are now found in processors. They accelerate data access under the condition that temporal and space locality is preserved. Given the increasing performance gap filled by these caches, locality of reference becomes an increasingly important consideration.

Increasing the number of cache levels is one way to deal with the ever increasing number of processors trying to access the same memory. It performs well as long as applications are providing good cache locality. An alternative solution to connect multiple cores to shared memory is Non-Uniform Memory Access (`NUMA`). This type

of architecture subdivides memory into regions closer to specific processors. Access to local memory is faster, but the cost is that access to remote memory is slower than machines with standard shared memory. Getting good performance on these architectures requires to adapt the operating system and applications to allocate data in memory regions close to the processor, and to take memory access locality in consideration when performing thread migration between processors.

Hence, being able to identify inefficient memory access patterns becomes increasingly important as memory access locality effect increases and memory layout becomes intrinsically tied to thread scheduling.

Because current architectures allow reordering of memory accesses to increase instruction-level parallelism, verification of parallel algorithms is required to ensure proper execution on parallel systems. Memory models and formal verification methods are one way to achieve such verification. The background will be presented in the state of the art of the article "Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms", found in Chapter 7.

2.1.3 Software-Level Support for Multiprocessing

Some of the widely used multiprocessing libraries to develop applications include the `pthread` library, part of the `glibc`, and `OpenMP`, now included in the `gcc` compiler. Outside of multiprocessing support added to intrinsically sequential languages like C or Fortran by added libraries, `SMP` support becomes increasingly integrated in languages like Java, where thread support is included in the language.

The K42 operating system is an experiment on highly-scalable OS, with built-in tracing support. Its tracing features will be detailed in the state of the art of the article "Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing", in Chapter 5.

KTAU is a kernel tracer for the Linux operating system developed for multi-processor computers. It provides detailed per-process tracing information, but only aggregated information for system-wide tracing. It will be detailed in the state of the art of the same Chapter.

2.2 Real-time

Increase of architecture-level complexity caused by addition of parallelism and increased impact of memory access locality is a concern for real-time applications.

An application is considered as having real-time requirements when it needs a computation to be performed in a deterministically known amount of time. The throughput provided by the application, in this case, comes second to the worst-case delay the computation can afford [13, 14].

There is a major problem in terms of real-time response caused by these architecture trends. While increasing the size of caches and the number of processors can help increasing the computing throughput, the side-effect of this complexity increase is that real-time response becomes harder to certify. It is affected by pipeline stalls, interactions between processors, and the fact that memory access delay becomes harder to predict.

Real-time requirements are usually classified under the banners “hard real-time” and “soft real-time”. At one extreme, hard real-time applications simply can’t afford to miss a deadline. This imposes strict requirements on verification and architecture behavior, where throughput is likely to be sacrificed in favor of increased determinism. At the other side of the spectrum, soft real-time applications also have real-time requirements, and would thus consider a missed deadline as a bug in the sense that it deteriorate service quality. Some of these applications could not afford to sacrifice performance because high throughput, and potentially low average latency, are parts of their requirements. Therefore, these applications will typically provide guarantees in terms of number of deadlines missed per time-period [15].

There is currently an increasing amount of such soft real-time applications used in the field. For instance, Autodesk is using a slightly customized Linux distribution to run their audio and video acquisition and edition applications [16]. Soft real-time and low-latency must however not be confused. For instance, the Google servers, answering to web search requests, must provide a low average latency response-time to their users in an effort to make their experience enjoyable. This is, in fact, slightly different from soft real-time: one slow answer once in a while can be acceptable, but their aim is to have very low average response time [17].

Some operating systems are specialized to meet hard real-time requirements [18]. This is the case of VxWorks and μC OS-II, for instance. These operating systems

are meant to provide a deterministic response time. In the hard real-time systems category, RTAI (*Real-Time Application Interface*) and Xenomai provide hard real-time support for user-space applications. They collaborates with a Linux kernel, but provide limited application interfaces to the real-time applications. The RT-preempt Linux kernel is derived from the standard Linux kernel. It aims at providing real-time guarantees to standard Linux applications. It modifies extensively locking primitives, interrupt handling and thread scheduling. The aim of this kernel is to provide real-time guarantees close to hard real-time, where missing a deadline is inadmissible. This comes at the cost of a significant performance regression compared to the standard mainline kernel, and is hence not yet suitable for high-throughput soft real-time applications.

Real-time systems requirements being heavily tied to the execution time, tracers can easily provide counter-examples showing where the system does not meet deadlines along with the faulty execution trace. This category of tool is therefore extremely useful for these systems.

Tracing tools are already widely used in the real-time field, such as the closed-source Wind River Tornado, a tracing tool embedded in the VxWorks real-time operating system.

Amongst the open-source tracers, the original Linux Trace Toolkit [19] is a tracer for the Linux operating system which has been made available for embedded real-time systems, including RTAI. The Ftrace Linux mainline kernel tracer evolved from the IRQ latency tracer, part of the Linux RT (*Real-Time*) kernel tree before the latter was integrated in the Linux mainline kernel. This tracer's goal was specifically to identify long latencies caused by interrupt disabling. The tracers for which source-code availability allows their study are presented in the article "Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing", in Chapter 5.

2.3 Distributed Systems

The growing trend of distributed systems, predicted in [5], is confirmed by their increasing use between 2000 and 2009 [9, 10], especially for large-scale indexing and query workloads.

In the scientific and engineering fields, the MPI library allows to divide a task across a group of computers using a standard message-passing programming interface

to exchange data at synchronization points. The Sun Studio Performance Analyzer and PIP [20] tracers instrument the `MPI` synchronization primitives and provide an insight into the workload by monitoring the exchanges between the nodes.

Commercial distributed systems are however less likely to rely on a `MPI`-style communication between the nodes. Algorithms like map-reduce [8], which reserve different roles to various nodes, including redundancy, are better suited to a `RPC` (*Remote Procedure Call*) interface. As discussed in Chapter 8, tracing the data exchanges is only part of the equation; knowing what is happening at the system-level on a per-node basis is also required to identify the causes of performance degradations.

2.4 Commercial Servers

Linux distribution packaging companies such as Redhat aim to provide tools, to help system administrators identifying performance issues on their servers running the Redhat Enterprise distribution. SystemTAP, based on the Kprobes, Kernel Markers and Tracepoint instrumentations, is Redhat's response to this demand. Solaris, from Sun, targets a similar market by enhancing their operating system with Dtrace, a system-wide tracer.

The main characteristic of this use-case category is to have servers running various services, which makes the task of pinpointing the source of performance slowdown across the kernel, library and application layers difficult.

This category of tracer will be studied in greater depth in the state of the art of Chapter 5, within the article "Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing".

Chapter 3

Methodology

This chapter presents the methodology used to proceed towards fulfillment of all research objectives, explaining the rationale for each research phase. Initially, interaction with industry partners (Autodesk, IBM Research, Google), the `LTT`, and Linux open source communities made possible to collect information about their requirements, computer architectures and software environment.

In parallel with this interaction, initial prototypes of the `LTTng` tracer were developed. These enabled us to gather real-world feedback on the tracer implementation and its usability. During this phase, initial contributions were made to the Linux kernel in order to publicize my work through the open-source community. For instance, extending the internal Linux kernel `API` with our synchronization primitives, *Local Atomic Operations*, permitted us to contribute an infrastructure identified as useful for the `LTTng` tracer very early in the development process. Being known in a community circle to be an active contributor for many years accounts very positively in the amount of help and feedback received in return.

Through its design and development, the `LTTng` tracing infrastructure has been stress-tested with very demanding workloads, benchmarked to ensure the tracer impact was within acceptable limits. It subsequently had its wait-free algorithms formally verified using model-checking, both for correctness and starvation-free behavior.

Still in the early phases of development, I identified the need to decouple instrumentation from the tracer. The idea is that adding instrumentation to the Linux kernel is a task better performed by contributors and experts of each individual kernel subsystem. Consequently, I created the *Kernel Markers* and *Tracepoints* infrastructures to manage this instrumentation and help the addition of instrumentation to the Linux kernel. These infrastructures are now integrated into the mainline Linux kernel and used extensively by the kernel developers community.

A few user-space `LTTng` tracer prototypes have been developed as part of this project. However, given the pace of kernel-side development, I decided to discontinue

support for the user-space tracer to eliminate a maintainability burden. However, this experience has been very useful in designing the kernel tracer in a way that would allow the algorithms to be easily migrated to user-space. In 2009, Pierre-Marc Fournier, research associate with the DORSAL research team at École Polytechnique de Montréal, started a port of the LTTng kernel tracer to a user-space library, which he called UST (*User-Space Tracer*). Its design benefits from the experience acquired with the past LTTng user-space tracers, and most of the synchronization is reusing the LTTng kernel buffering algorithms, the *Markers* and *Tracepoints*.

A key aspect of the LTTng design for scalability and low performance overhead has been the use of RCU (*Read-Copy Update*) to synchronize trace-control data read-side accesses. However, this mechanism did not exist in user-space. I ergo volunteered to develop new algorithms enabling RCU usage within the more constrained user-space context. This led to the implementation of a user-space RCU library which provides similar services to the library present in the Linux kernel. With IBM's approval to release their contribution under the LGPL license, the resulting library has been made widely available and already has active users and contributors.

A fine-grained model for formal verification of these complex algorithms was needed to provide a high confidence level in the implementation. Hence, I undertook the task of creating a model of modern weakly-ordered processors to ensure algorithmic correctness and progress at the lowest level, although generic enough to ensure portability of the algorithm implementations.

3.1 Interaction with the community

Nowadays, many companies are doing business peripheral to operating systems, and thus need them, but do not earn money directly from selling operating systems. Amongst them, Google, IBM, Intel, Fujitsu, Autodesk, Ericsson, Siemens, Nokia; all these collaborate through the Linux open source community to create and maintain the GNU/Linux operating system.

Reaching for feedback from industry partners and from the open source community has improved my understanding of today's computer system ecosystem. Establishing contact with such enterprises and groups of developers was initiated early in our research, with the objective to benefit as quickly as possible from the community expertise and feedback. An important part of this collaboration-oriented research has

been the interaction with the industry, as well as with the Linux and LTT communities.

3.1.1 Tracing in the Industry

The first phase of this research consists in meeting with industry partners, users of tracing solutions, as well as participating in conferences. It enabled the identification of tracing requirements from the industry. Internships and partnerships with Autodesk, IBM Research, and Google, as well as meetings and discussions with Wind River helped gather this information from experienced industry partners. The Montreal Tracing Summits held in 2008 [21] and 2009 [22], as well as discussions on tracing at the Kernel Summit 2008 [23] helped us gather further input from industry.

An important opportunity I had through my research is to do an internship at the IBM T.J. Watson research center under the mentorship of Robert Wisniewski, one of the K42 authors, an expert in highly-scalable systems and lock-less algorithms. I have been exposed to different architectures, mainly PowerPC, and to Commercial Scale-Out workloads, which ended up badly needing tracer tools to find problem culprits. I contributed, during my presence at IBM Research, to the paper "Experiences Understanding Performance in a Commercial Scale-Out Environment" [10]. It outlines the requirements for a system-wide tracer able to analyze cluster nodes.

Another rich learning experience was an internship at Google, Mountain View, with the Platform Team, under the supervision of Martin J. Bligh. He is a very active and well-known Linux kernel developer, with contributions to the Linux NUMA support and to memory management. I have been allowed to learn about the Google infrastructure and their requirements. We wrote a conference paper, "Linux Kernel Debugging on Google-sized clusters" [24], presented at the Ottawa Linux Symposium in 2008. I contributed various use-cases for which tracers have been useful at Autodesk and IBM Research.

The LTTng project also had impact in the security field. I presented debugger circumvention techniques at the Recon 2006 conference, and showed how the low-impact LTTng kernel tracer can be used to study elusive pieces of software, such as viruses. This resulted in a conference paper, "OS Tracing for Hardware, Driver and Binary Reverse Engineering in Linux" [25], for which I am the principal author. It is published in the CodeBreakers journal. This paper, originally part of the Recon 2006 conference proceedings [26], was later added to a standard journal issue.

3.1.2 Tracing in the Open-Source Community

An initial version of `LTTng` was designed to meet the requirements from the industry and the open-source community. The `LTTng` prototypes were iteratively proposed to the community, and revised to fit their requirements. Noticing, from early benchmarks, that per-`CPU` atomic operations were a very promising synchronization mechanism in terms of performance, I contributed a complete `API` to the Linux kernel, called *local atomic operations*. This first contribution helped us learn how the community works, helped us publicize the `LTTng` project, and helped facilitate feedback and exchange with the community through the rest of the project.

Implementing the tracer indicated a very high coupling between the Linux kernel and its instrumentation. I thus created *Linux Kernel Markers*, which evolved into *Tracepoints*, today integrated and used widely in the Linux kernel. This permitted us to decouple kernel instrumentation from the tracer.

3.1.3 Authored publications

Initially, conference articles were presented to discuss early results with the Linux community. The article presented in [27] describes the `LTTng` architecture in its early development phase. The article [28] outlines work done to diminish instrumentation overhead (*Immediate Values*) and to instrument the Xen hypervisor. This article was followed by one presented at the Linux Foundation Collaboration Summit [29] which discusses the difference between tracing requirements for kernel developers and Linux end-users.

A second community, real-time users, has also been targeted. Conference articles were presented at the Embedded Linux Conference in 2006 [30] and 2009 [31]. The first one focused on a use-case of the `LTTng` tracer to identify long latencies caused by interrupt disabling. The second article focused more on a developer-level audience, showing what architecture-specific components are needed to port the `LTTng` tracer to new architectures.

The paper presented at Recon 2006 [26], afterward presented in an issue of the Code Breakers Journal [25], targets the computer security community. It presents how the `LTTng` kernel tracer can be used to understand the behavior of hostile executables, taking as an example a Linux virus.

3.1.4 Co-authored publications

Industrial internships at IBM Research and Google resulted in publication of papers co-authored with the respective teams. Both papers discussed use-cases requiring high-throughput tracing solutions. The paper published with IBM Research [10] presents how a tracing solution can help understanding and debugging commercial scale-out systems. On the other hand, the paper co-authored with the Google team [24] is more specifically aimed at showing various tracing use-cases to the Linux community. It gathers use-cases from experience at Google, IBM and Autodesk.

3.2 Tracer Architecture

This section presents the solution I propose to address the constraints imposed by my research hypothesis. The proposed design aims at preserving the following operating system's aspects: scalability, portability, real-time response and, within limits permitting workload reproducibility, low latency and throughput.

3.2.1 Overview

Pursuing the objective of allowing multiple analysis to be performed on a single collected trace, enabling in-depth analysis of hard to reproduce bugs, we extract trace data from the kernel.

In order to answer the various industry requirements, the architecture depicted in Figure 3.1 is proposed. Grey rectangles represent major phases of tracing. Within these rectangles, ellipses represent the tracing phases, linked with arrows showing the trace data flow direction. Between the tracing and post-processing phases, a dotted *Input/Output* arrow represents extraction of trace data through `I/O` mechanisms: disk, network, serial port, etc.

The *tracing* phases are performed on the traced system, using the processor, memory bandwidth and `I/O` resources required to extract the data out of the kernel. Initially, *instrumentation* is inserted into the operating system kernel. When the kernel executes and reaches an instrumentation site, it verifies if the tracing site is activated, and calls the attached probes. These probes perform all synchronization required to write events into the trace buffers.

Writing to circular memory buffers without live trace data extraction is called *flight recorder* tracing. This is one tracing mode available. The other mode consists in extracting the trace data while tracing is active. This latter phase is named *buffered data extraction*. Events are gathered in memory buffers to ensure that costly `I/O` operations are not used by the probe execution. The `I/O` phase is performed by specialized threads. It can be either done live while the trace is being recorded, or after tracing activity is over. In the latter case, only the very last buffers written will be available for analysis.

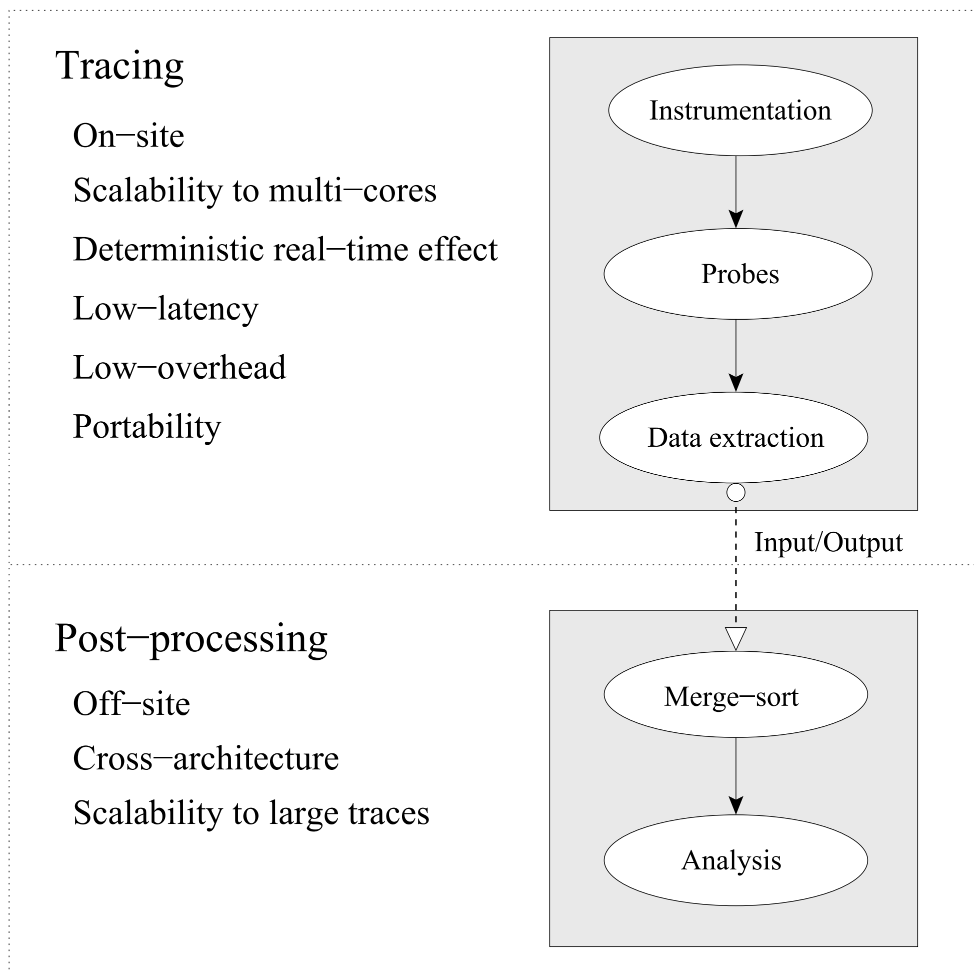

Figure 3.1 Tracer architecture diagram

Extracting large amounts of data, albeit having a small impact on system performances, involves applying very strict implementation constraints. We deal with this

at the design-level by minimizing the amount of trace data extraction synchronization required between processors and execution contexts. A zero-copy approach, involving no trace data copy between memory locations through the tracing phases, ensures efficient use of memory bandwidth.

Event reordering at *post-processing* is made possible by gathering a time-stamp value from the traced processor, written by the *probe* at each event header. The time-stamp is typically the value of a time-source synchronized across processors. When provided by the architecture, a cycle count register synchronized across processors can be used as time-source. This allows a posteriori reordering of events based on their time-stamps.

The *post-processing* phase can be performed either in the same environment as the traced kernel or on a completely different computer architecture. It may not be assumed that the traced and post-processing machines are the same architecture. Thus, trace data extracted must be readable by the post-processor. We propose self-described binary traces, written by the traced kernel in its native binary format, to extract compact trace data efficiently and portably.

3.2.2 Instrumentation

As a result from the tracing requirement study, we noticed that some work needed to be done to allow kernel-wide instrumentation without hurting performance when tracing is not active.

The `Kprobes` infrastructure, already available in the mainline Linux kernel, is a hardware breakpoint-based instrumentation approach. It dynamically replaces each kernel instruction to instrument with a breakpoint, which generates a trap each time the instruction is executed. A tracing probe can then be executed by the trap handler. However, due to the heavy performance impact of breakpoints, the inability to extract local variables anywhere in a function due to compiler optimizations, and the maintenance burden of keeping instrumentation separate from the kernel code, a more elaborate solution was needed.

I present, in Chapter 8, the *Linux Kernel Markers* and *Tracepoints* infrastructure, which I created to decouple kernel instrumentation from the tracer probes. The markers and tracepoints allow us to declare instrumentation statically at the source-code level without affecting performance significantly and without adding the cost of a

function call when instrumentation is disabled. Having extremely low-overhead when instrumentation is dynamically disabled is crucially important to provide Linux distributions the incentive to ship instrumented programs to their customers. Markers and tracepoints consist in a branch skipping a stack setup and function call when instrumentation is dynamically disabled (dormant). These individual instrumentation sites can be enabled dynamically at runtime by dynamic code modification, and only add low overhead when tracing. The typical overhead of a *dormant* marker or tracepoint has been measured to be below 1 cycle [28] when cache-hot. Static declaration of tracepoints helps manage this instrumentation within the Linux kernel source-code. Given that the Linux kernel is a distributed collaborative project, enabling each kernel subsystem instrumentation to be maintained by separate maintainers helps distributing the burden of managing kernel-wide instrumentation.

However, statically declaring an instrumentation site for dynamic activation typically incurs a non-zero performance overhead due to the test and branch required to skip the instrumentation call. To overcome this limitation, I created the concept of activating statically compiled code efficiently by dynamically modifying an immediate operand within an instruction, which I called *Immediate Values*. With these optimizations, we have been able to show the Linux kernel community that a static instrumentation approach is viable and could have near-zero overhead when disabled. The better performing prototypes realized within this infrastructure replace a standard load, test and branch by a static jump which skips over the instrumentation unconditionally. Runtime code-patching allows us to either activate or deactivate an unconditional jump to dynamically enable branches. This results in completely unmeasurable performance impact for the *Tracepoints* and *Kernel Markers*. These prototypes initiated an effort at Redhat to implement the required compiler support for *static jump patching*. More details about these infrastructures are presented in Chapter 8.

Once the static instrumentation problem is covered, allowing kernel developers to add instrumentation to their own subsystems, we are allowed to turn our focus on efficiently extracting the trace data outside of the kernel.

3.2.3 Synchronization Primitives

During the initial phase of the `LTTng` implementation, I noticed that the number of cycles required to execute synchronization instructions was much higher than for standard instructions, at least on x86 and PowerPC. Because a kernel tracer has reentrancy requirements from various execution contexts, protection is required in order to collect coherent, non-corrupted, traces.

I identified the synchronization primitives needed for efficient tracing on some of the most common architectures. In the first paper, presented in Chapter 4, I discuss the reentrancy question: which concurrent execution contexts can write trace data to a shared memory buffer. I propose a new clock management infrastructure using appropriate locking for tracing and then benchmark each possible choice of synchronization primitive suitable to deal with the identified concurrency. I subsequently propose to use the identified set of synchronization primitives as key synchronization components in a kernel tracer.

3.2.4 Buffering

Following the discovery of the elected synchronization primitives to perform reentrant and fast kernel tracing, I created the `LTTng` buffering scheme to meet the low-overhead, low-latency, real-time, scalability, portability and reentrancy requirements targeted by this research. The design rules I imposed on the algorithm development were that no synchronization primitive impacting the overall traced system behavior can be used, and that each loop depending on external conditions to complete should be carefully verified for deadlocks and possible unexpected delays and starvation that could be caused by the probe sharing a data structure with a lower-priority execution context.

I verified these properties with benchmarks and formal verification to demonstrate that the research objectives are met. It showed linear scalability when increasing the number of cores, wait-free guarantees for the kernel implementation, cache-hot performance overhead between 119–314 ns per probe execution on the x86 processor family, and 1014 ns for ARMv7. The equations and algorithms, as well as the benchmarks, are detailed in the second paper, presented in Chapter 5. Formal verification of the algorithm correctness and wait-free guarantees is presented in Section 8.6 The latency added by a `LTTng` probe, as presented in Section 8.4, has been measured to be in a

95% confidence interval between 328 and 338 ns on an Intel Xeon.

3.2.5 Read-Copy Update

The Read-Copy Update synchronization mechanism makes RP (Relativistic Programming) more accessible. This class of programming techniques targets shared-memory multiprocessor architectures, achieving linear read-side scalability by tolerating that different processors see memory operations in different orders and by tolerating concurrent accesses to a memory location. RCU provides very efficient and wait-free read-side. It has been used extensively in the Linux kernel since its integration in 2002, and counts 2500 uses in the 2.6.30 kernel as of June 2009 [32].

However, although it has been made widely available in the Linux kernel, no algorithm as efficient as the ones used for kernel RCU existed for user-space execution. This execution context poses specific problems in terms of support for multi-threaded execution, support for synchronization of library data and lack of direct access to internal kernel API, hence limiting the control over the scheduler and interrupts. Considering the pervasiveness of multi-core architectures in commodity hardware, the need for highly efficient synchronization can be expected to grow. Consequently, we can expect many applications to benefit from making RCU more widely available. One of these is, as a matter of fact, the UST (*User-Space Tracer*) library. To fill the need for low-overhead synchronization of UST, I thereafter created the *Userspace RCU* library. This library aims at providing a very low-overhead and highly-scalable read-side to the UST tracer, a dynamic shared object loaded with the applications. Great care was taken not to require any application modifications when loading the UST tracer. This requirement equally applied to the *Userspace RCU* library.

Paul E. McKenney, author of the kernel RCU and Distinguished Engineer at the IBM Linux Technology Center, and I proceeded to benchmark various alternative user-space RCU implementations on Intel and PowerPC computers, with respectively 8 and 64 cores, showing linear read-side scalability. These user-space experiments contributed to Paul E. McKenney's simplifications of the kernel-side preemptible RCU. These kernel RCU synchronization primitives are used by the LTTng kernel tracer. I am the main author of an article on "User-Level Implementations of Read-Copy Update", presented in Chapter 6. I led the implementation effort with the collaboration of Paul E. McKenney. This resulted in the *Userspace RCU* library, of which I am the author

and maintainer.

3.3 Experimentation

Various approaches are used to ensure that the implementation realized meets the scalability, low-overhead, low-latency and deterministic real-time characteristics expected. Benchmarks are used to quantify the tracing overhead in terms of fraction of system resources used, scalability and latency impacts. It is also used to compare the performance overhead to other existing tracing solutions. Correctness of the synchronization primitives is ensured by stress-testing and formal verification.

3.3.1 Benchmarks

We demonstrate tracer properties using latency, throughput and scalability benchmarks. Section 5.5 presents `LTTng` benchmarks on systems running various workloads.

To characterize the latency induced by the tracer probes, we demonstrate that the cache-hot `CPU`-time overhead per probe executed ranges from 119 to 1014 ns, depending on the architecture. This is complemented by latency benchmarks in Section 8.4, performed on the Intel Xeon, which shows that the 95% confidence interval of the average tracer probe execution time on a workload more representative of normal cache conditions is between 328 and 338 ns. This confirms that latency impact of the tracer is low.

To measure the throughput impact of the `LTTng` tracer, we ran benchmarks measuring network and disk traffic throughput. The overhead in terms of fraction of system throughput used is typically below 3 % for `flight recorder` mode and 6 % for trace data extraction to disk for event throughput of 0.8 million events per second, which confirms that the typical tracer throughput overhead is low. Given that the tracer is expected to be used in the field on production machines running industry-level workloads, throughput stress-tests were performed to push the tracer to its limits to measure upper-bounds to its capacity. Under the heaviest stress-tests, tracer throughput overhead is 28 % for 10 million events per second gathered in `flight recorder` mode. A 35 % stress-test impact has been measured for 2.5 million events per second when writing events to disk. This confirms that, even on heavy workloads, the tracer throughput overhead is only a fraction of the system throughput.

We also performed a comparative study with existing tracing solutions. Section 5.5.8 presents a comparison between LTTng and Dtrace. The results obtained on an x86 architecture shows an acceleration factor of 6.42 to 1 in favor of LTTng, which demonstrates that LTTng performs significantly better than the existing tracing solutions.

In this same section, scalability of the LTTng tracer is studied by comparing the throughput of an intrinsically parallel workload with various number of nodes, with and without tracing. On a 8-core Intel Xeon, we observe that system linear throughput scalability is conserved under tracing. This confirms that the LTTng tracer scales linearly with the number of CPU.

Benchmarking is not suitable to study the impact of the tracer on system real-time determinism. It is rather verified with a formal method, as presented in the following section.

3.3.2 Formal Verification

Developing correct synchronization algorithms becomes an increasingly complex task with all reordering freedom taken by some architectures. Hence, only resorting to test does not permit to validate that these synchronization primitives will behave as expected in every context. For instance, an incorrect RCU implementation with a single grace-period phase could result in a sequence of events causing a grace-period guarantee failure, as presented in Section 6.4.3.

Section 8.6 presents the formal verification performed on the LTTng wait-free buffering algorithm. A model of the LTTng wait-free algorithm verified with the Spin model-checker, enabled to identify an incorrectness in the original sub-buffer full and empty equations, which led to design modifications leading to the equations presented in Section 5.4.2. Formal verification allows to ensure algorithm correctness; in other words, that no race condition nor data corruption can occur. It also ensures that the algorithm is wait-free by performing progress verification.

The final paper, presented in Chapter 7, details a framework to model multi-core architectures with weak instruction and memory ordering, enabling verification of low-level synchronization algorithms such as RCU.

We will proceed to the detailed explanation of each of these research phases contained within the articles presented in the next chapters.

Chapter 4

Paper 1: Synchronization for Fast and Reentrant Operating System Kernel Tracing

4.1 Abstract

To effectively trace an operating system, a performance monitoring and debugging infrastructure needs the ability to trace various execution contexts. These contexts range from kernel running as a thread to `NMI` (*Non-Maskable Interrupt*) contexts.

Given that any part of the kernel infrastructure used by a kernel tracer could lead to infinite recursion if traced, and because most kernel primitives require synchronization unsuitable for some execution contexts, all interactions of the tracing code with the existing kernel infrastructure must be considered in order to correctly inter-operate with the existing operating system kernel.

This paper presents a new low overhead tracing mechanism and motivates the choice of synchronization sequences suitable for operating system kernel tracing, namely *local atomic instructions* as main buffer synchronization primitive and the `RCU` (*Read-Copy Update*) mechanism to control tracing. It also proposes a wait-free algorithm extending the time-base needed by the tracer to 64-bit on architectures that lack hardware 64-bit time-base support.

4.2 Introduction

As computers are becoming increasingly multi-core, the need for debugging facilities that will help identify timing related problems, across execution layers and between processes, is rapidly increasing [24, 33, 34]. Such facilities must allow information gathering about problematic workloads without affecting the behavior being

investigated in order to be useful for problem analysis purposes.

The variety of execution contexts reached during kernel execution complicates efficient exportation of trace data out of the instrumented kernel. The general approach used to deal with this level of concurrency is to either provide good protection against other execution contexts, for instance by disabling interrupts, or to adopt a fast, but limited mechanism by shrinking the instrumentation coverage (e.g. by disallowing interrupt handler instrumentation). This trade-off often means that either performance or instrumentation coverage is sacrificed. However, as this paper will show, this trade-off is not required if the appropriate synchronization primitives are chosen.

We propose to extend the OS[1] kernel instrumentation coverage compared to other existing tracers by dealing with the variety of kernel execution contexts. Our approach is to consider reentrancy from NMI[2] execution context, which presents particular constraints regarding execution atomicity due to the inability to create critical sections by disabling interrupts. In this article, we show that in a multiprocessor OS, the combination of synchronized time-stamp counters, cheap single-CPU atomic operations and trace merging, provides an effective and efficient tracing mechanism which supports tracing in NMI contexts.

Section 4.4 will first present the synchronization primitives used by the state-of-the-art open source tracers. In Section 4.5, we outline the LTTng[3] design requirements which aim at high scalability and minimal real-time response disruption.

Recursion between the tracer probe and the kernel will be discussed in Section 4.6, where the reasons why a kernel tracer cannot call standard kernel primitives without limiting the instrumentation coverage will be illustrated by concrete examples in Sections 4.6.1 and 4.6.2. We then propose an algorithm that synchronizes data structures to provide a 64-bit tracer clock on architectures that lack hardware 64-bit time-base support in Section 4.6.3.

In Section 4.7 we show how to achieve both good performance and instrumentation coverage by choosing primitives that provide the right reentrancy characteristics and high performance. Overhead measurements comparing the RCU[4] [35] mechanism and *local atomic operations* primitives to alternative synchronization methods will support this proposal.

1. OS: Operating System
2. NMI: Non-Maskable Interrupt
3. LTTng: Linux Trace Toolkit Next Generation
4. RCU: Read-Copy Update.

This opens a wide perspective for the design of fully reentrant, wait-free, high-performance buffering schemes.

4.3 Introduction to Tracing

This section introduces the background required to understand the tracing state of the art. The tracer impact on real-time response is first discussed, followed by a description of tracer concepts.

The real-time impact of algorithms can be categorized following the guarantees they provide. The terms used to identify such guarantees evolved through time in the literature [36, 37]. The terminology used in this article will follow [37]. The strongest non-blocking guarantee is wait-free, which ensures each thread can always make progress and is therefore never starved. A non-blocking lock-free algorithm only ensures that the system as a whole always makes progress. This is made possible by ensuring that at least one thread is progressing when concurrent access to a data structure is performed. An obstruction-free algorithm offers an even weaker guarantee than lock-free: it only guarantees progress of any thread executing in isolation, meaning that all competing concurrent accesses may be aborted. Finally, a blocking algorithm does not provide any of these guarantees.

This article specifically discusses operating system kernel tracers. A tracer consists of a mechanism collecting an execution trace from a running system. A trace is a sequence of event records, each identifying that the kernel executed a pre-identified portion of its code.

Mapping between execution sites and events is made possible by instrumentation of the kernel. Instrumentation can be either declared statically at the source-code level or dynamically added to the running kernel. Statically declared instrumentation can be enabled dynamically.

A tracer probe is the tracer component called when enabled instrumentation is executed. This probe is responsible for fetching all the data to write in the event record, namely an event identifier, a time-stamp and, optionally, an event-specific payload. Time-stamps are monotonically increasing values representing the time-flow on the system. They are typically derived from an external timer or from a `TSC`[5] register on the processor.

5. `TSC`: Time-Stamp Counter

To amortize the impact of I/O [6] communication, event records are saved in memory buffers. Their extraction through an I/O device is therefore delayed. To ensure continuous availability of free buffer space, a ring buffer with at least two sub-buffers can be used. One is used by tracer probes to write events while the other is extracted through I/O devices.

4.4 State of the Art

This section reviews the synchronization primitives used in the state-of-the-art open source tracers currently available, namely: the original LTT tracer [19], the wait-free write-side tracing solution found in K42 [38, 39], a highly-scalable research operating system created by IBM, DTrace [33] from Sun's OpenSolaris, SystemTAP [40] from RedHat, providing scripting hooks for the Linux kernel built as external modules, KTAU [41] from University of Oregon and Ftrace, started by Linux kernel maintainer Ingo Molnar. The following study details the synchronisation mechanisms used in each of these projects.

The original LTT (Linux Trace Toolkit) [19] project started back in 1999. Karim Yaghmour, its author, aimed at creating a kernel tracer suitable for the Linux kernel with a static instrumentation set targeting the most useful kernel execution sites. LTT uses the architecture time-stamp counter register when available to interpolate the time between the time-stamps taken at sub-buffer boundaries with *do_gettimeofday()*. This leads to problems with NTP (Network Time Protocol) correction, where the time-base at the sub-buffer boundaries could appear to go backward. Regarding synchronization, the *do_gettimeofday()* function uses a sequence counter locking on the read-side for ensuring time-base data consistency, which can cause deadlocks if NMI handlers were instrumented. One of the early buffering scheme used was based on spin lock (busy-waiting lock) disabling interrupts for buffers shared between the CPUs. Per-CPU buffers support, provided by RelayFS, uses interrupt disabling to protect from interrupt handlers. Karim worked, in collaboration with Tom Zanussi and Robert Wisniewski from IBM, on the integration of some lockless buffering ideas from the K42 tracer into RelayFS.

K42 [38, 39] is a research operating system developed by IBM, mostly between 1999 and 2006. According to the authors, its code-base should be considered as a

6. I/O: Input/Output

prototype. It focuses on large multi-processor machine scalability and therefore uses data structures and operations local to each CPU as much as possible. It brings some very interesting ideas for tracing synchronization, namely the use of atomic operations to synchronize buffer space reservation. The tracing facility found in K42 is built into the kernel. It uses per-CPU buffers to log tracing data and limits the consumption of data to user-space threads tied to the local CPU. This first design constraint could be problematic in a production OS, because if the workload is not equally shared amongst all CPUs, those with the most idle time will not be able to collaborate with the busier CPUs to help them extract the trace streams to disk or over the network. It uses a wait-free algorithm based on the *CAS* (*compare-and-swap*) operation to manage space reservation from multiple concurrent writers. It adds compiler optimisation restriction barriers to order instructions with respect to the local instruction stream, but does not add memory barriers, since all data accesses are local. Once the space reservation is performed, the data writes to the buffer and the commit count increments are done out-of-order. A *buffers produced* count and a *buffers consumed* count are updated to keep track of the buffers available for consumption by the user-space thread. For time-base synchronization, K42 only supports architectures with 64-bit time-stamp counters for the PowerPC and AMD64, and assumes that those counters are synchronized across all CPUs. Therefore, a simple register read is sufficient to provide the time-base at the tracing site, and no synchronization is required after system boot.

The DTrace [33] tracer has first been made available in 2003, and formally released as part of Sun's Solaris 10 in 2005 under the CDDL [7] license. It aims at providing information to users about their system's behavior by executing scripts at the kernel level when instrumentation sites are reached. It has since then been ported to FreeBSD and Apple Mac OS X 10.5 Leopard. A port to the Linux kernel is under development, but involves license issues between CDDL and GPL [8].

The DTrace tracer [9] disables interrupts around iteration on the probe array before proceeding to their invocation. Therefore, the whole tracer site execution is protected from interrupts coming on the local CPU. Trace control synchronization is based on a RCU-like [35] mechanism. After trace control data modification, the updater thread

7. CDDL: Common Development and Distribution License.
8. GPL: General Public License.
9. Version reviewed: OpenSolaris 20090330.

waits for a grace period before all previously executing tracing sites can be considered to have completed their execution, thus reaching a quiescent state. This is performed by executing a thread on every CPU which is only scheduled when the currently active tracing sites have completed their execution. Disabling interrupts serves as a means to mark the tracing site execution, which therefore permits detection of tracing sites quiescent states.

DTrace also uses a per-thread flag, *T_DONTDTRACE*, ensuring that critical kernel code dealing with page mappings does not call the tracer. It does not seem, however, to apply any thread flag to NMI handler execution. In OpenSolaris, NMIs are primarly used to enter the kernel debugger, which is not allowed to run at the same time as DTrace. Therefore, the following discussion applies to a situation where the same algorithms and structures would be used in an operating system like Linux, where NMIs can execute code contained in various subsystems, including the Oprofile [42] profiler.

DTrace calls the *dtrace_gethrtime()* primitive to read the time source. On the x86 architecture, this primitive uses a locking mechanism similar to the sequence lock in Linux. A sequence lock is a type of lock which lets the reader retry the read operation until the writer exits its critical section. The particularity of the sequence lock found in DTrace is that if it spins twice waiting for the lock, it assumes that it is nested over the write lock, so a time value previously copied by the time-base tick update will be returned. This shadow value is protected by its own sequence lock. In the x86 implementation, this leaves room for a 4-way deadlock on 2 CPUs involving the NTP correction update routines, *tsc_tick()* and two nesting *dtrace_gethrtime()* calls in interrupt handlers.

Although this deadlock should never cause harm due to specific and controlled use of NMIs in OpenSolaris, porting this tracer to a different operating system or loading specific drivers using NMIs could become a problem. Discussion with Bryan Cantrill, author of DTrace, with Mike Shapiro and Adam Leventhal, led us to notice that an appropriate NMI-safe implementation based on two sequence locks taken successively from a **single** thread already exists in *dtrace_gethrestime()*, but is not used in the lower-level x86 primitive. It requires that only a single execution thread takes the two sequence locks successively. Using it in the lower-level code would require modification of the NTP adjustment code [10]. This example taken from a widely distributed tracer

10. Based on review of the DTrace code-base, we recommend using a standard mutex to ensure

shows that it is far from trivial to design tracing clock source synchronization properly, especially for a flexible open source operating system like Linux.

Considering real-time guarantees, a sequence lock should be categorized as a blocking algorithm. If an updater thread is stopped in the middle of an update, no reader thread can progress. Therefore, a sequence lock does not provide non-blocking guarantees. This means real-time behavior can be affected significantly by the execution of `DTrace`.

The `SystemTAP` [40] project from Redhat, first made available in 2006, aims at letting system administrators run scripts connected at specific kernel sites to gather information and statistics about the system behavior and investigate problems at a system-wide level. It aims at providing features similar to `DTrace`[11] in the Linux operating system. Its first aim is not to export the whole trace information flow, but rather to execute scripts which can either aggregate the information, perform filtering on the data input or write data into buffers along with time-stamps. The focus is therefore not to have a very high-performance capable data extraction mechanism, given this is not their main target use-case. `SystemTAP` uses a heavy locking mechanism at the probe site. It disables interrupts and takes a global spin lock[12] twice in the write path. The first critical section surrounded by interrupt disabling and locking is used to manage the free buffer pool. The second critical section, similar to the former but using a distinct lock, is needed to add the buffer ready for consumption to a *ready queue*.

`SystemTAP` assumes it is called from `Kprobes` [43], a Linux kernel infrastructure permitting connection of breakpoint-based probes at arbitrary addresses in the kernel. Kprobes disables interrupts around handler execution. Therefore, `SystemTAP` assumes interrupt disabling is done by the caller, which is not the case for static instrumentation mechanisms like the Linux Kernel Markers and `Tracepoints`. In those cases, if events come nested over the tracing code, caused by recursion or coming from `NMIs`, `SystemTAP` will consider this as an error condition and will silently discard the event until the number of events discarded reaches a threshold. At that point, it will stop tracing entirely. `SystemTAP` modules can use the *gettimeofday()* primitive

mutual exclusion around the two write sequence locks should allow to permit using the same locking mechanism for both *dtrace_gethrestime()* and *dtrace_gethrtime()*, which would allow updates from `NTP` and from the *tsc_tick()* routine.

11. According to `http://sourceware.org/systemtap/wiki/SystemtapDtraceComparison`.

12. A spin lock is a type of busy-waiting lock in the Linux kernel.

exported by the Linux kernel as time source. It uses a sequence lock to ensure the time-base coherency. This fails in a NMI context because it would cause a deadlock if a probe in a NMI nests over a sequence writer lock. Therefore, SystemTAP's internals disallows instrumentation of code reached from NMI context. It also depends on interruptions being disabled by the lower-level instrumentation mechanism.

The KTAU (Kernel Tuning and Analysis Utilities) [41] project, available since 2006, allows either the profiling or tracing the Linux kernel on a system-wide or per-process basis. It allows detailed per-process collection of trace events to memory buffers, but deals with kernel system-wide data collection by aggregating performance information of the entire system. The motivation for using aggregation to deal with system-wide data collection is that exporting the full information flow into tracing buffers would consume too much system resources. Conversely, the hypothesis the LTTng approach is trying to verify is that it is possible to trace a significant useful subset of operating system's execution in a detailed manner without prohibitive impact on the workload behavior. Therefore, we have to consider if the KTAU process-centric tracing approach would deal with system-wide tracing appropriately.

Some design decisions indicate that detailed process tracing is not meant to be used for system-wide tracing. KTAU keeps buffers and data structures local to each thread, which can lead to significant memory usage on workloads containing multiple threads. Workloads consisting of many mostly inactive threads and few very active threads risk overflowing the buffers if they are too small, or consuming a lot of memory if all buffers are made larger. KTAU allows tweaking the size of specific thread's buffers, but it can be difficult to tune if the threads are short-lived. We can also notice that the kernel idle loop, which includes swap activity, and all interrupts and bottom halves nested over this idle loop, are not covered by the tracer, which silently drops the events.

For synchronization, KTAU permits choosing at compilation time between IRQ or bottom half (lower priority interrupts) disabling and uses a per-thread spin lock to protect its data structures. The fact that the data can stay local to each thread ensures that no unnecessary cache-line bouncing between the CPUs will occur. Those spin locks are used therefore mainly to synchronize the data producer with the consumer. This protection mechanism is thus not intended to trace NMIs because the handler could deadlock when taking a spin lock if it nests over code already holding the lock.

Regarding kernel reentrancy, KTAU uses vmalloc (kernel virtual memory) to allocate the trace buffers. Given that the Linux kernel populates the TLB (Translation Lookaside Buffer) entries of those pages lazily on x86, the tracing code will trigger page faults the first time those pages are accessed. Therefore, the page fault handler should be instrumented with great care. KTAU only supports x86 and PowerPC and uses the time-stamp counter register as a time source, which does not require any synchronization per se. On the performance impact side, allocation of tracing buffers at each thread creation could be problematic on workloads consisting of many short-lived threads, because thread creation is normally not expected to be slowed down by multiple page allocations, since threads usually share the same memory pages.

Ftrace, a project started in 2009 by Ingo Molnar, grew from the IRQ tracer, which traces long interrupt latencies, to incrementally integrate the wake up tracer, providing information about the scheduler activity, the function tracer, which instruments the kernel function entry at low-cost and an actively augmented list of tracers. Its goal is to provide system-wide, but subsystem-oriented tracing information primarily useful to kernel developers. It uses the Tracepoint mechanism, which comes from the LTTng project, as primary instrumentation mechanism.

Ftrace, in its current implementation, disables interrupts and takes per-buffer (and thus per-CPU) spin locks. The advantage of taking a per-CPU spin lock over a global spin lock is that it does not require to transfer the spin lock cache-line between CPUs when the lock has to be taken, which improves the scalability when the number of CPUs increases. Ftrace, as of its Linux 2.6.29 implementation, does not handle NMIs gracefully. If instrumentation is added in a code path reached by NMI context, a deadlock may occur due to the use of spin locks. Ftrace relies on the scheduler clock for timekeeping, which does not provide any locking against non-atomic *jiffies*[13] counter updates. Although this time source is statistically correct for scheduler purposes, it can result in incorrect timing data when the tracer races with the *jiffies* update. Dropping events coming from nested NMI handlers will be the solution integrated in the 2.6.30 kernels. Improvement is expected in a near future regarding tracing buffer ability to handle NMIs gracefully using a lock-free kernel-specific buffering scheme submitted for U.S. and international patent in early 2009

13. The *jiffies* counter increments at each timer tick, at a frequency typically between 100 and 1000 Hz.

by Steven Rostedt [14].

4.5 Linux Trace Toolkit Next Generation

The purpose of the study presented in this paper is to be used as a basis for developing the LTTng kernel tracer. This tracer aims at tracing the Linux kernel while providing these guarantees:

– Provide a wide instrumentation coverage.
– Provide probe reentrancy for all kernel execution contexts, including NMIs and MCE (Machine Check Exception) handlers.
– Record very high-frequency kernel events.
– Impose small overhead to typical workloads.
– Scale to large multiprocessor systems.
– Change the system real-time response in a predictable way.

Earlier work presented an overview of the LTTng tracer design [27] and industry use-case scenarios in the industry [24, 10, 30, 25]. That work presents an in-depth analysis of synchronization primitives and new algorithms required to deal with some widely used 32-bit architectures.

The LTTng tracer probe needs, as input, a clock source to provide timestamps, trace control information to know if tracing is enabled or if filters must be applied, and the input data identified by the instrumentation. The result of its execution is to combine its inputs to generate an event written to a ring buffer.

In order to provide good scalability when the number of CPU increases, LTTng uses per-CPU buffers and buffer management counters to eliminate cache misses and false-sharing. This diminishes the impact of the tracer on the overall system performance. Nevertheless, cross-CPU synchronization is still required when information is exchanged from a producer to a consumer CPU.

This paper will justify LTTng's use of the RCU mechanism to synchronize control information read from the probe, local CAS and proper memory barriers to synchronize ring buffer output and present a custom trace clock scheme used to deal with architectures lacking a 64-bit hardware clock source.

14. As stated in the Ftrace presentation at the Linux Foundation Collaboration Summit 2009.

4.6 Tracing Synchronization

In this section we describe the *atomic primitives* and `RCU` mechanisms used by the `LTTng` [27, 44] tracer to deal with the constraints associated with *synchronization* of data structures while running in any *execution context*, avoiding *kernel recursion*. We then present an `RCU`-like trace clock infrastructure required to provide 64-bit time-base on many 32-bit architectures. The associated *performance impact* of the synchronization primitives will be studied thereafter, which will lead to the subsequent benchmark section.

4.6.1 Atomic Primitives

This section presents synchronization considerations for kernel data read from the tracing probe, followed by inner tracer synchronization for the control data structures read using `RCU` and buffer space reservation performed with atomic operations.

Because any execution context, including `NMIs`, can execute the probe, any data accessed from the probe must be consistent when it runs. Kernel data identified by the instrumentation site is expected to be coherent when read by all execution contexts associated with the given site. It is therefore the instrumentation site's responsibility to correctly synchronize with those kernel data structures.

Data read by the probe can be classified into two types. The first type contains global and static shared variables read from kernel memory. The second type includes data accessed locally by the processor, contained either in registers, on the thread or interrupt stack, or in per-CPU data structures when preemption [15] is temporarily disabled.

Synchronization of shared data structures is ensured by static instrumentation because the data input identification is located within the source code which carries the correct locking semantic. Conversely, dynamic instrumentation offers no guarantee that global or static variables read by the probe will be appropriately synchronized. For instance, Kprobes [43] do not export specific data at a given instrumentation site. Therefore, it does not guarantee locking other than what is being done in the kernel

15. User space preemption naturally occurs when the scheduler interrupts a thread executing in user space context and replaces it by another runnable thread. At kernel-level, with fully-preemptible Linux kernels (*CONFIG_PREEMPT=y*), the scheduler can preempt threads running in kernel context as well.

around the breakpoint instruction. Given that there are not necessarily any data dependency between the instruction being instrumented and the data accessed within the probe, subtle race conditions may occur if locking is not performed appropriately within the probe.

Local data accessed by its owner execution context, however, do not have such locking requirements because it is normally modified only by the local execution context. The probe which accesses this data executes either in the same execution context owning this data or in a trap generated by instructions within the owner context. However, compiler optimizations do not guarantee to keep local variables live at the probe execution site with Kprobes. Static instrumentation can make sure that the compiler keeps the data accessed live at a specific instruction.

Information controlling tracing behavior is accessed directly from the probe, without any consideration regarding the context in which it is executed. This information includes the buffer location, produced and consumed data counters and a flag to specify if a specific set of buffers is active for tracing. This provides flexibility so users can tune the tracer following their system's workload.

`LTTng` uses the `RCU` mechanism to manage the trace-control data structure. This synchronization mechanism provides very fast and scalable data structure read access by keeping copies of the protected data structure when a modification is performed. It gradually removes an outdated data structure by first replacing all pointers to it by pointers to the new version. It keeps all data copies in place until a grace period has passed, which identifies a read-side quiescent state and therefore permits reclamation of the data structure. A `RCU` read-side is wait-free, but the write-side can block if no more free memory is available. Moreover, the write-side may either block waiting for a grace period to end, or queue memory reclamation as a `RCU` callback to execute after the current grace period. In this latter case, reclamation is performed in batch after the current grace period ends. It therefore provides very predictable read-side real-time response. Given that the trace control data structure updates are rare, this operation can afford to block. `LTTng` marks the read-side critical sections by disabling preemption because this technique is self-contained (it does not use other kernel primitives) and due to its low overhead. The `LTTng` trace-control write-side waits for readers to complete execution to provide guarantees to the trace-control caller. Therefore, when the operation *start trace* completes, the caller knows all current and new tracer probes are seeing an active trace. The opposite applies when

tracing stops.

The tracing information is organized as a RCU list of trace structures, and is only read by the probe to control its behavior. Since the probe is executed with preemption disabled, updates to this structure can be done on a copy of the original while the two versions are presented to the probes when the list is updated: probes holding a pointer to the old structure still use the old one, while the newly executing probes use the new one. A quiescent state is reached when all processors have executed the scheduler. It guarantees that all preemption-disabled sections holding a pointer to the old structure finished their execution. It is thus safe, from that point, to free the old data structure.

With the RCU mechanism, the write-side must use preemptible mutexes to exclude other writers and has to wait for quiescent states. Luckily, such trace data structure updates are rare (e.g. starting a trace session), so update performance is not an issue.

Because the RCU mechanism wait-free guarantees apply only for the read-side, LTTng cannot leverage RCU primitives to deal with reentrancy coming from any execution context to synchronize memory buffer space reservation, which includes updating a data structure. Primitives, allowing protection from concurrent execution contexts performing buffer space reservation on the local CPU, need to execute atomically with respect to interrupts and NMIs, which implies that *atomic operations* must be used to perform atomic data accesses.

Given that the cross-CPU synchronization points are clearly identified and occur only when sub-buffers can pass from a producer CPU to a consumer CPU at sub-buffer boundaries, the performance impact of synchronization primitives required for each event should be characterized to find out which set of primitives are adequate to protect the tracer data structures from use in concurrent execution contexts.

On modern architectures such as Intel Pentium and above, AMD, PowerPC and MIPS, using atomic instructions, synchronized to modify shared variables in a SMP (Symmetric Multi-Processor) system, incurs a prohibitive performance degradation due to the synchronized variant of the instructions used (for Intel and AMD) or to the memory barriers which must be used on PowerPC, MIPS and modern ARM processors. Given that several atomic operations are often required to perform the equivalent synchronization of what would otherwise be done by disabling interrupts a single time, the latter method is often preferred. The wait-free write-side tracing

algorithm used in LTTng[16] needs a single CAS operation to update the write count (amount of space reserved for writing) and an atomic increment to update the commit count (amount of information written in a particular sub-buffer).

Given that the tracing operations happen, by design, only on per-CPU data, their single-CPU atomic primitives can be safely used. This means Intel and AMD x86 do not need *LOCK* prefix to synchronize these atomic operations with concurrent CPU access, while PowerPC, MIPS and modern ARM processors do not require them to be surrounded by memory barriers to ensure correct memory order, since the only order that matters is from the point of view of a single CPU. Therefore, those lightweight primitives, faster than disabling interrupts on many architectures, can be used. Section 4.7 will present benchmarks supporting these claims.

4.6.2 Recursion with the Operating System Kernel

The instrumentation coverage depends directly on the amount of interaction the probe has with the rest of the kernel. In fact, the tracer code itself cannot be instrumented because it would lead to infinite probe recursion. The same applies to any kernel function used by the probe[17].

In the Linux kernel, the x86 32 and 64-bit architectures rely on page faults to populate the page table entries of the virtual memory mappings created with *vmalloc()* or *vmap()*. Since the kernel modules are allocated in this area, any access to module instructions and data might cause a minor kernel page fault. Care must therefore be taken to call the *vmalloc_sync_all()* primitive which populates all the kernel virtual address space reserved for virtual mappings with the correct page table entries between module load and use of this module at the tracing site. This ensures that no recursive page fault will be triggered by the page fault handler instrumentation.

In the context of the probe, the most important limitation regarding operating system recursion is the inability to wake up a process when the buffers are ready to be read. Instrumenting thread wake-ups provides very useful information about the inner scheduler behavior. However, instrumentation of this scheduler primitive forbids using it in the tracer probe. This problem is solved by adding a periodic timer which samples the buffer state and wakes up the consumers appropriately. Given that the

16. LTTng kernel tracing algorithm with wait-free write-side will be presented in a forthcoming paper.

17. A particularly unobvious example is the page fault handler instrumentation.

operating system already executes a periodic timer interrupt to perform scheduling and manage its internal state, the performance impact of this approach is in the same order of magnitude as adding a callback to the timer interrupt. The impact on low power-consumption modes is kept small by ensuring that these per-processor polling timers are delayed while the system is in these low-power modes. Therefore, polling is only performed when the system is active, and thus generating trace data.

As a general guideline, the probe site only touches its own variables atomically, so it requires no higher-level synchronization with the OS. On the OS side, any operation done on those shared variables is also performed atomically. It results in an hermetic interface between the probe and the kernel which makes sure the probe calls no OS primitive.

Because preemption must be disabled around probe execution, primarily to allow the `RCU`-based data structures reads, care must be taken not to use an instrumented version of the preemption disabling macros. It can be done by using the untraced implementation *preempt_disable_notrace()*.

4.6.3 Timekeeping

Time-stamping must also be done by the probe. It therefore has to read a time-base. In the Linux kernel, the standard *gettimeofday()* or other clock sources are synchronized with a sequence lock (*seqlock*), which consists of a busy loop on the read-side, waiting for writers to finish modifying the data structure and checking for a sequence counter modification prior to and after reading the data structure. However, this is problematic when `NMI`s need to execute the read-side, because nesting over the write lock would result in a deadlock; the `NMI` would wait endlessly for the writer to complete its modification, but would do so while being nested over the writer. Normal use of this synchronization primitive requires interrupt disabling, which explains why it is generally correct, except in this specific case. Another issue is that the sequence lock is a blocking synchronization algorithm, because the updater threads have the ability to inhibit reader progress for an arbitrarily long period of time. Therefore, the CPU time-stamp register, when available, is used to read the time-base rather than accessing any kernel infrastructure.

Some architectures provide a 64-bit time-base. This is the case for the cycle counter read with *rdtsc* on x86 32 and 64-bit [45], the PowerPC time-base register [46]

and the 64-bit MIPS [47]. A simple atomic register read permit reading a full 64-bit time-base. However, architectures like the 32-bit MIPS and ARM OMAP3 [48] only provide a 32-bit cycle counter. Other architecture which lack proper cycle counter register support must read external timers. For instance, earlier ARM processors must read the time-base from an external timer through memory mapped I/O. Memory-mapped I/O timers usually overflow every 32-bit count or even more often, although some exceptions, like the Intel `HPET` [49], permits reading a 64-bit value atomically in some modes.

The number of bits used to encode time has a direct impact on the ability of the time-base to accurately keep track of time during a trace session. A 64-bit time-base is guaranteed not to overflow for 3 thousand years at 4 GHz, which should be enough for any foreseeable use. However, at a 500 MHz frequency, typical for embedded systems, 32-bit overflows occur every 8 seconds.

Tracing-specific approaches to deal with time-stamp overflow has been explored in the past, all presenting their own limitations. The sequence lock inability to deal with `NMI` context has been presented above, although one could imagine porting the `DTrace` double-sequence lock implementation to address this problem. This approach is however slower than the `RCU` read-side and implies using a blocking sequence lock, which fails to provide good real-time guarantees.

Alternatively, an approach based on a posteriori analysis of the event sequence presented in the buffers could permit detecting overflows, but this requires a guaranteed maximum time delta between two events, which could be hard to meet due to its dependency on the workload and events traced. Low-power consumption systems with deep sleep states are good examples of such workloads. Periodically writing a resynchronization time-stamp read from a lower-frequency time-source would diminish the precision of time-stamps to the precision of the external time-source.

If, instead of writing such a resynchronization event periodically, it was written in a header to the buffer containing the events, this would again either impose limits on the slowest event flow expected, otherwise a buffer covering too long a time period could contain undetectable 32-bit overflows. Also, given that the buffer is naturally expected to present the events in an order in which time monotonically increases, performing adjustments based on a different time-source at the buffer boundary can make time go backward because the two clocks are not perfectly synchronized. One clock going too fast could make the last buffer events overlap the time window of

the following buffer. Simply using a `CAS` instruction would not solve the issue, given that the architectures we are dealing with only have a 32-bit cycle counter and are typically limited to 32-bit atomic operations.

There is already an existing approach in the Linux kernel, created initially for the ARM architecture, to extend a 32-bit counter to 63 bits. This infrastructure, named *cnt32_to_63*, keeps a 32-bit (thus atomically updated) value in memory. Its lower 31 bits are used to represent the extended counter top 31 bits. A single bit is used to detect overflow by keeping track of the low-order 32nd bit. Update is performed atomically in the reader context when a 32-bit overflow is detected. Assuming the code is run at least twice per low-order 32-bit overflow period, this algorithm detects the 32-bit overflows and updates the high-order 31-bit count accordingly. This approach has the benefit of requiring a very small amount of memory data (only 32 bits) and being fast: given the snapshot is updated on the reader-side as soon as the overflow is detected, the branch verifying this condition only needs to be taken very rarely. This approach, however, has some limitations: it only permits us to keep an amount of data smaller than the architecture word size. Therefore, it is not extensible: it would not be possible to return the full 64-bit, because the top bit must be cleared to zero, and it could not support addition of `NTP` or CPU frequency scaling information. This infrastructure assumes that the hardware time-source will always appear to go forward. Therefore, with slightly buggy timers or if the execution or memory accesses are not performed in order, this would cause time to jump forward a whole 32-bit period if the time-source appears to slightly decrement at the same time an overflow occurs. This could be fixed by reserving one more bit to also keep track of the low-order 31st bit and require the code to be called 4 times per counter overflow period. Those two bits could be used together to distinguish between overflow and underflow. This would however be at the expense of yet another high-order information bit and only permit returning a 62-bit time-base.

Therefore, a new mechanism would be welcome to generically extend these 32-bit counters to 64-bit while still allowing a time-base read from `NMI` context. The algorithm we created to solve this problem extends a counter containing an arbitrary number of bits to 64 bits. The data structure used is a per-CPU array containing two 64-bit counts. A pointer to either the first or the second array entry is updated atomically, which permits to atomically read the odd counter while the even is being updated and conversely (as shown in Figure 4.1). The reader atomically reads the

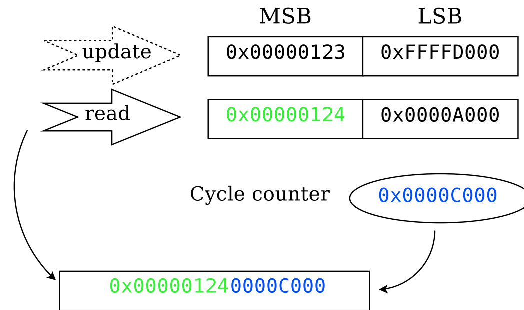

Figure 4.1 Trace clock read (*no* 32nd bit overflow)

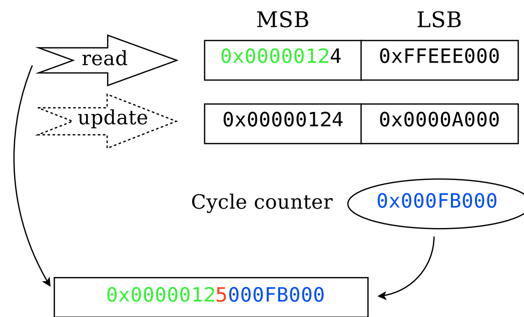

Figure 4.2 Trace clock read (32nd bit overflow)

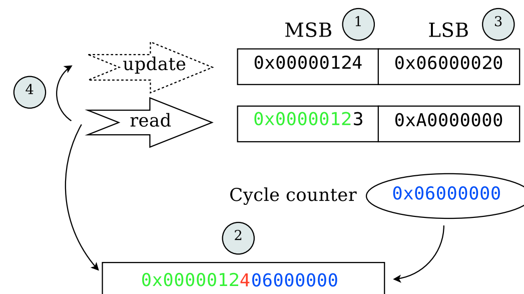

Figure 4.3 Trace clock update (1, 3, 4) interrupted by a read (2)

pointer to the current array parity and then reads the last 64-bit value updated by the periodic timer. It then detects the possible overflows by comparing the current value of the time source least significant bits with the low-order bits of the 64-bit value. It returns the 64 bits corresponding to the current count, with high-order bits incremented if a low-order bit overflow is detected (as shown in Figure 4.2).

The algorithm for synthetic clock read-side is shown in Figure 4.4. At line 1, `TC_HW_BITS` is defined as the number of bits provided by the clock source, represented by a call to `hw_clock_read()`. The main limitation on the minimum number of bits required from the clock source is that it must be larger than the sum of timer interrupt period and maximum interrupt latency. This ensures that a timer interrupt is

```
 1 #define HW_BITMASK            ((1ULL << TC_HW_BITS) - 1)
 2 #define HW_LS(hw)             ((hw) & HW_BITMASK)
 3 #define SW_MS(sw)             ((sw) & ~HW_BITMASK)
 4
 5 struct synthetic_tsc_struct {
 6   u64 tsc[2];
 7   unsigned int index;
 8 };
 9
10 static DEFINE_PER_CPU(struct synthetic_tsc_struct, synthetic_tsc);
11
12 static inline notrace u64 sw_tsc_read(u64 old_sw_tsc)
13 {
14       u64 hw_tsc, new_sw_tsc;
15
16       hw_tsc = (u64)hw_clock_read();
17       new_sw_tsc = SW_MS(old_sw_tsc) | hw_tsc;
18
19       if (unlikely(hw_tsc < HW_LS(old_sw_tsc)))
20         new_sw_tsc += 1ULL << TC_HW_BITS;
21
22       return new_sw_tsc;
23 }
24
25 u64 notrace trace_clock_read_synthetic_tsc(void)
26 {
27       struct synthetic_tsc_struct *cpu_synth;
28       unsigned int index;
29       u64 sw_tsc;
30
31       preempt_disable_notrace();
32       cpu_synth = &per_cpu(synthetic_tsc, smp_processor_id());
33       index = ACCESS_ONCE(cpu_synth->index);
34       sw_tsc = sw_tsc_read(cpu_synth->tsc[index]);
35       preempt_enable_notrace();
36
37       return sw_tsc;
38 }
```

Figure 4.4 Synthetic clock read-side

executed at least once per counter overflow period. Lines 2–3 present the `HW_LS()` and `HW_MS()` macros, to select the least and most significant bits of a counter, respectively corresponding to the hardware clock source and the bits counting the clock-source overflows. Lines 5–8 declare a structure containing two 64-bit `tsc` values and an index to the current `tsc` value to read. Line 10 defines a per-CPU variable, `synthetic_tsc`, holding the current `tsc` value for each processor.

The inline function `sw_tsc_read` is detailed at lines 12–23. The `notrace` keyword is a macro expanding to a gcc attribute indicating that the function must not be traced, in the unlikely event gcc decides not to inline the function. It receives as parameter the last 64-bit clock value saved in the data structure and returns the current 64-bit clock value. The current source clock value is read at line 16. The current 64-bit clock value is then derived from the old 64-bit clock most significant bits and the source clock bits. If an overflow is detected by line 19, the 64-bit clock value is incremented of the power of two value corresponding to the overflow at line 20.

Lines 25–37 show the execution context considerations taken around the execution of the trace clock read. Lines 31 and 35 disable and re-enable preemption, therefore inhibiting the scheduler during this execution phase. This ensures that no thread migration occurs, therefore ensuring local access to per-CPU data. It also ensures that the thread is not scheduled out for a long period of time between the moment it reads the index, reads the clock source and accesses the array. Long preemption between these operations could cause the current clock value to be more than a clock-source overflow apart from the previously read last 64-bit clock value when the thread resumes. To overcome this problem, the maximum duration for which this code can be interrupted is bounded by the maximum interrupt handler execution time, which must be an order of magnitude lower than the overflow period. Line 32 uses the `per_cpu` inline, a primitive which gets a pointer to the CPU-local instance of `synthetic_tsc`. `ACCESS_ONCE()` is used at line 33 to read the current index through a volatile access, which informs the compiler to treat this as an access to a memory-mapped hardware device, therefore not permitting re-fetching nor reading in multiple segments. Line 34 invokes the `sw_tsc_read()` inline explained above, which returns the current 64-bit clock value.

The update is performed periodically, at least once per overflow period, by a per-CPU interrupt timer. It detects the low-order bits overflows and increments the upper bits, and then flips the current array entry parity (as shown in Figure 4.3). Readers

still use the previous 64-bit value while the update is done until the update completes with the parity flip.

As pointed out earlier, the read-side must disable preemption to ensure that it only holds a reference to the current array parity for a bounded amount of cycles, much lower than the periodic timer period. This upper bound is provided by the maximum number of cycles spent in this short code path increased by the worse interrupt response time expected on the system. It is assumed that no interrupt flood will hold the code path active for a whole timer period. If this assumption is eventually proven to be wrong, disabling interrupts around the algorithm execution could help not experiencing this type of problem, but delaying of timer interrupt would still leave room for overflow miss.

The update-side algorithm is detailed in Figure 4.5. The function `update_synthetic_tsc()` must be executed periodically on each processor. It is expected to be executed in interrupt context (therefore with preemption already disabled) at least once per overflow period. Line 6 gets a pointer to the CPU-local `synthetic_tsc`. Line 7 flips the current index back and forth between 0 and 1 at each invocation. Lines 8–9 invoke `sw_tsc_read()` to read the current 64-bit `TSC` value, using the last synthetic TSC value saved in the data structure by the previous `update_synthetic_tsc()` execution. The current 64-bit `TSC` value is saved in the free array entry, unused at that moment. Line 10 is a compiler barrier, ensuring that the index update performed on line 11 is not reordered before line 8 by the compiler. This makes sure concurrent interrupts and `NMI`s are never exposed to corrupted data.

If processors need to be kept in low-power mode to save energy, the per-processor interrupt needed to update the current 64-bit synthetic `TSC` value can be disabled in

```
 1 static void update_synthetic_tsc(void)
 2 {
 3       struct synthetic_tsc_struct *cpu_synth;
 4       unsigned int new_index;
 5
 6       cpu_synth = &per_cpu(synthetic_tsc, smp_processor_id());
 7       new_index = 1 - cpu_synth->index;
 8       cpu_synth->tsc[new_index] =
 9         sw_tsc_read(cpu_synth->tsc[cpu_synth->index]);
10       barrier();
11       cpu_synth->index = new_index;
12 }
```

Figure 4.5 Synthetic clock periodic update

such low-power mode, replaced by a resynchronization on an external timer counter upon return to normal processor operation.

The amount of data which can be placed in the per-CPU array is not limited by the architecture size. This could therefore be extended to support time-base correction for CPU frequency scaling and `NTP` correction. If the hardware time-source is expected to appear to run slightly backward (due to hardware bugs or out-of-order execution), the algorithm presented above could additionally check the 31st bit to differentiate between overflow or underflow in order to support non-perfectly monotonic time-sources and still keep the ability to return the full 64 bits.

Given that each read-side and write-side thread will complete in a bounded amount of cycles without waiting, this time-base enhancement algorithm can be considered as wait-free, which ensures that no thread starvation can be caused by this algorithm.

It must be understood, however, that this proposed algorithm does not replace a proper 64-bit time-stamp counter implemented by hardware. Indeed, if a faulty device holds the bus or if a driver disables interrupts for more than a cycle-counter overflow period, it would lead to time-base inaccuracy due to miss of one (or more) cycle-counter overflow. Making sure that this situation does not happen would imply reading an external clock source in addition to the cycle counter, which does not meet our efficiency constraints. Therefore, given that it is of utmost importance to be able to rely on core debugging facilities like kernel tracers, it is highly recommended to use hardware providing full 64-bit cycle counters. However, given that software must often adapt to hardware limitations rather than the opposite, the algorithm proposed should work correctly, unless some hardware or driver is doing something *really* bad like holding the bus or disabling interrupts for a few seconds.

4.7 Benchmarks

This section will present the benchmarks used to choose the right synchronization primitives for tracing, given their respective performance impact on many of the mainstream architectures, namely Intel and AMD x86, PowerPC, ARM, Itanium and SPARC. The goal of the present section is to show that it is possible to use local atomic operations without adding prohibitive overhead relative to interrupt disabling. It will be demonstrated that, on most architectures, it is even faster to use local atomic operations than to disable interrupts.

The benchmarks presented consist of 20,000 executions of the cache-hot synchronization primitive. The overall execution time is determined by sampling the cycle counter once before and once after the 20,000 executions. The average time per iteration is therefore obtained. Given that these synchronization primitives typically bring only a single cache line from memory to the processor, we assume cache miss cost to be almost the same for each synchronization primitives. We study their cache-hot performance impact, rather than their cache-cold impact, because combination of synchronization primitives will typically still only require a single cache line for synchronization data.

A comparison between benchmarks realized only with synchronization primitives, and with added operations within the synchronization is presented at Table 4.1. The added operation consists of 10 word-sized reads and one word write. For each locking primitive, the columns present the number of cycles required to execute only the synchronization, and the synchronization with the added operation, respectively. The last column presents the difference from the expected baseline, which is assume is caused by pipeline effects. It shows that the simple operations account for a negligible number of cycles compared to the synchronization cost, and that costly synchronization primitives such as synchronized `CAS` are made even slower by the added operations, probably due to pipeline stalls caused by the serializing instruction. Therefore, the following benchmarks only take into account the synchronization primitive execution time.

The assembly listings for the following Intel Xeon benchmarks are presented in Figures 4.6, 4.7, 4.8 and 4.9. The term *speedup* is used to represent the acceleration of one synchronization primitive compared to another.

Let's first focus on performance testing of the `CAS` operation. Table 4.2 presents benchmarks comparing disabling interrupts to local `CAS` on various architectures. The columns present, in this order, the architecture on which the test is done, the speedup obtained (cost of disabling and enabling interrupts divided by the overhead of a local `CAS`), the overhead, in cycles, of local `CAS`, synchronized `CAS`, enabling and disabling interrupts. When comparing the synchronization done with local `CAS` to disabling local interrupts alone, a speedup between 4.60 and 5.37 is reached on x86 architectures. On PowerPC, the speedup range is between 1.77 and 4.00. Newer PowerPC generations seems to provide better interrupt disabling performance than the older ones. Itanium, for both older single-core and newer dual-core 9050 processor,

has a small speedup of 1.33. Conversely, UltraSPARC atomic CAS seems inefficient compared to interrupt disabling, which makes the latter option about twice faster. As we will discuss below, besides the performance considerations, all those architectures allow `NMI`s to execute. Those are, by design, unprotected by interrupt disabling. Therefore, unless the macroscopic impact of atomic operations becomes prohibitive, the tracer robustness, and ability to instrument code executing in `NMI` context, favors use of atomic operations.

Tables 4.3, 4.4 and 4.5 present the different synchronization schemes that could be used at the tracing site. Table 4.3 shows the individual elementary operations performed when taking a spin lock (busy-waiting loop) with interrupts disabled. For each architecture, the overhead of spin locks, and interrupt disabling/enabling pairs is shown, as well as the sum of these overhead. These numbers are a "best case", because they do not consider the non-scalability of this approach. Indeed, the spin lock atomic variable must be shared between all CPUs, which leads to performance degradation when the variable must be alternately owned by different CPU's caches, a phenomenon known as cache-line bouncing.

Table 4.4 presents the equivalent synchronization performed using a sequence counter lock and a fully synchronized atomic operation. The *Seqlock* column presents the number of cycles taken for a sequence counter lock. These locks are used in the kernel time-keeping infrastructure to make sure reading the 64-bit jiffies is consistent on 32-bit architectures and also to ensure the monotonic clock and the clock adjustment are read consistently. The Sync. `CAS` column presents the number of cycles taken to perform synchronized `CAS` operation. This operation is needed because preemption is kept enabled, which allows migration. Therefore, given that the probe could be preempted and migrated between the moment it reads the processor ID and the moment it performs the atomic data access, concurrency between CPUs must be addressed by a SMP-aware atomic operation. If preemption is left enabled, per-CPU data would be accessed by the local CPU most of the time, so it would statistically provide a good cache locality, but, in cases where a thread is migrated to a different CPU between reading the pointer to the data structure and the write to the reserve or commit counters, we could have concurrent writes in the same structure from two processors. Therefore, the synchronized version of `CAS` and increment should be used if preemption is left enabled. It is interesting to note that ARMv7 OMAP3 shows a significant slowdown for the sequence lock. This is caused by the requirement for

read barriers before and after the sequence number read due to lack of address [48] or control dependency between the sequence lock and the data to access. ARMv7 does not have weaker read-side only memory barriers and therefore requires two *dmb* (Data Memory Barrier) instructions, which decreases performance significantly.

Table 4.5 presents a `RCU` approach to synchronization. It involves disabling preemption around the read-side critical section, keeping a copy of the old data structures upon update and making sure the write-side waits for a grace-period to pass before the old data structure can be considered private and memory can be reclaimed. Disabling preemption, in this scheme, also has an effect on the scheduler: it ensures that the whole critical section is not preempted nor migrated to a different CPU, which permits to use the faster local `CAS`. For each architecture, column 2 presents the number of cycles taken to disable and re-enable preemption. Column 3 presents the time taken for a local `CAS`.

Table 4.6 presents the overall speedup of each synchronization approach compared to the baseline: Spin lock disabling interrupts. For each architecture, the baseline speedup is presented in column 2, followed by the sequence lock and `CAS` speedup. Finally, column 3 presents the preemption disabling and local `CAS` speedup.

If we would only care about the read-side, the sequence counter lock approach is the fastest: it only takes 3-4 cycles on the x86 architecture family to read the sequence counter and to compare it after the data structure read. This is faster than disabling preemption, which takes 8-9 cycles on x86. Preemption disabling is the cost of `RCU` read-side synchronization. Therefore, in preemptible kernels, a `RCU` read-side could be slightly slower than a sequence lock. On non-preemptible kernels, however, the performance cost of `RCU` falls down to zero and outperforms the sequence lock. But the synchronization requirements we have also involve synchronizing concurrent writes to data structures.

In our specific tracing case, in addition to read tracing control information, we also have to synchronize for writing to buffers by using `CAS` to update the write counter and by using an atomic increment to keep track of the number of bytes committed in each sub-buffer. Choosing between a *seqlock* and `RCU` has a supplementary implication: the seqlock outperforms `RCU` on preemptible kernels only because preemption is left enabled. However, this implies that a fully synchronized `CAS` and atomic add must be used to touch per-CPU data to prevent migration.

The speedup obtained by using the `RCU` approach rather than the sequence lock

ranges between 1.2 and 2.53 depending on the architectures, as presented in Table 4.6. This is why, overall, the `RCU` and local atomic operations solution is preferred over the solution based on read-side sequence lock and synchronized atomic operations. Moreover, in addition to execute faster, the `RCU` approach is reentrant with respect to `NMI`s. The read sequence lock would deadlock if an `NMI` nests over the write lock.

4.8 Least Priviledged Execution Contexts

The discussion presented above focused on tracing the kernel execution contexts. It it however important to keep in mind that different execution contexts, namely user-space, have different constraints. The main distinction comes from the fact that it is a bad practice to let user-space code modify data structures shared with the kernel without going through a system call, because this would pose a security threat and lead to potential privilege escalation.

If we were to port the tracing probe to perform user-space tracing, the trade-off would differ. The main downside of the `RCU` approach, for both the scheduler-based and preemptible versions, is that it requires the writer to wait for reader quiescent state before the old memory can be reclaimed. This could be a problem when exporting data from kernel-space to user-space, (e.g. time-keeping data structures) where the write-side is the kernel and the reader is user-space. When synchronizing between different privilege levels (kernel vs user-space), the highest privilege level must never wait or synchronize on the least-privileged execution context, otherwise resource exhaustion could be triggered by the lower privilege context.

4.9 Conclusion

As this paper has demonstrated, the current state of the art in tracing involves either instrumentation coverage limitations, synchronization flaws or limitation of the architectures supported to those which have synchronized 64-bit time-stamp counters.

A set of synchronization primitives has been proposed which fulfill the instrumentation coverage requirements of kernel tracing, adding code executed in `NMI` handler context, which was not properly handled by state-of-the-art tracers. Those primitives are the local `CAS` instruction and the `RCU` mechanism along with preemption disabling around the tracing code execution.

A wait-free algorithm, to extend a time-base providing less than 64-bit (which overflows periodically during the trace) to a full 64-bit counter by software, has been detailed. It should help tracers implement time-bases without the flaws caused by incorrect use of the sequence lock and improving the real-time guarantees compared to the sequence lock.

Finally, benchmarks have demonstrated that, on almost all architectures (except SPARC), using local `CAS` for synchronization rather than disabling interrupts is actually faster. It shows that using atomic primitives over interrupt disabling allows to grow the instrumentation coverage, including code executed from `NMI` handler context, without sacrificing performance.

This will open the door to the design of fully reentrant, wait-free, high-performance buffering schemes and to speedups in kernel primitives currently using interrupt disabling to protect their execution fast path, such as the memory allocator.

Acknowledgements

We would like to thank the Linux Trace Toolkit, Linux Kernel and SystemTAP communities for their feedback, as well as NSERC, Google, IBM Research and Autodesk for funding parts of this work.

We are indebted to Robert Wisniewski, Robert Roy, Paul E. McKenney, Bryan Cantrill, Simone Winkler and Etienne Bergeron for their constructive comments on this paper. Special thanks to David Miller, Josh Boyer, Alan D. Brunelle and Andika Triwidada for helping running benchmarks on various architectures.

Table 4.1 Benchmark comparison between locking primitives and added inner operations, on Intel Xeon E5405

Locking primitive	Sync. only (cycles)	Sync. and operations (cycles)	Pipeline effect (cycles)
Baseline (no locking)	1	60	0
Local `CAS`	8	60	-7
Sync. `CAS`	24	94	11
`IRQ` save/restore	39	97	-1
Spin lock/unlock	46	99	-6
seqlock	3	60	-2
Preemption disable/enable	12	60	-11

Synchronized `CAS`:

```
110:   48 89 c8                mov    %rcx,%rax
113:   f0 0f b1 0d 00 00 00    lock cmpxchg %ecx,0x0(%rip)
11a:   00
11b:   ff c2                   inc    %edx
11d:   81 fa 20 4e 00 00       cmp    $0x4e20,%edx
123:   75 eb                   jne    110
```

Local `CAS`:

```
1e8:   48 89 c8                mov    %rcx,%rax
1eb:   0f b1 0d 00 00 00 00    cmpxchg %ecx,0x0(%rip)
1f2:   ff c2                   inc    %edx
1f4:   81 fa 20 4e 00 00       cmp    $0x4e20,%edx
1fa:   75 ec                   jne    1e8
```

Figure 4.6 Assembly listings for Intel Xeon benchmarks (`CAS` loop content)

Interrupt restore:

```
468:   56                    push   %rsi
469:   9d                    popfq
46a:   ff c0                 inc    %eax
46c:   3d 20 4e 00 00        cmp    $0x4e20,%eax
471:   75 f5                 jne    468
```

Interrupt save (and disable):

```
530:   9c                    pushfq
531:   59                    pop    %rcx
532:   fa                    cli
533:   ff c0                 inc    %eax
535:   3d 20 4e 00 00        cmp    $0x4e20,%eax
53a:   75 f4                 jne    530
```

Interrupt save/restore:

```
600:   51                    push   %rcx
601:   9d                    popfq
602:   9c                    pushfq
603:   59                    pop    %rcx
604:   fa                    cli
605:   ff c0                 inc    %eax
607:   3d 20 4e 00 00        cmp    $0x4e20,%eax
60c:   75 f2                 jne    600
```

Figure 4.7 Assembly listings for Intel Xeon benchmarks (interrupt save/restore loop content)

Table 4.2 Cycles taken to execute CAS compared to interrupt disabling

Architecture	Speedup	CAS		Interrupts	
	(cli + sti) / local CAS	local	sync	Enable (sti)	Disable (cli)
Intel Pentium 4	5.24	25	81	70	61
AMD Athlon(tm)64 X2	4.60	6	24	12	11
Intel Core2	5.37	8	24	21	22
Intel Xeon E5405	5.25	8	24	20	22
PowerPC G5	4.00	1	2	3	1
PowerPC POWER6	1.77	9	17	14	2
ARMv7 OMAP3[a]	4.09	11	71	25	20
Itanium 2	1.33	3	3	2	2
UltraSPARC-IIIi [b]	0.64	0.394	0.394	0.094	0.159

a. Forced SMP configuration for test module. Missing barriers for SMP support added in these tests and reported to ARM Linux maintainers.

b. In system bus clock cycles.

Spin lock:

```
ffffffff814d6c00 <_spin_lock>:
ffffffff814d6c00:       65 48 8b 04 25 08 b5    mov    %gs:0xb508,%rax
ffffffff814d6c07:       00 00
ffffffff814d6c09:       ff 80 44 e0 ff ff       incl   -0x1fbc(%rax)
ffffffff814d6c0f:       b8 00 01 00 00          mov    $0x100,%eax
ffffffff814d6c14:       f0 66 0f c1 07          lock xadd %ax,(%rdi)
ffffffff814d6c19:       38 e0                   cmp    %ah,%al
ffffffff814d6c1b:       74 06                   je     ffffffff814d6c23 <_spin_lock+0x23>
ffffffff814d6c1d:       f3 90                   pause
ffffffff814d6c1f:       8a 07                   mov    (%rdi),%al
ffffffff814d6c21:       eb f6                   jmp    ffffffff814d6c19 <_spin_lock+0x19>
ffffffff814d6c23:       c3                      retq
```

Spin unlock:

```
spin_unlock:
ffffffff814d6f10 <_spin_unlock>:
ffffffff814d6f10:       fe 07                   incb   (%rdi)
ffffffff814d6f12:       65 48 8b 04 25 08 b5    mov    %gs:0xb508,%rax
ffffffff814d6f19:       00 00
ffffffff814d6f1b:       ff 88 44 e0 ff ff       decl   -0x1fbc(%rax)
ffffffff814d6f21:       f6 80 38 e0 ff ff 08    testb  $0x8,-0x1fc8(%rax)
ffffffff814d6f28:       75 06                   jne    ffffffff814d6f30 <_spin_unlock+0x20>
ffffffff814d6f2a:       f3 c3                   repz retq
ffffffff814d6f2c:       0f 1f 40 00             nopl   0x0(%rax)
ffffffff814d6f30:       e9 fb e1 ff ff          jmpq   ffffffff814d5130 <preempt_schedule>
ffffffff814d6f35:       66 66 2e 0f 1f 84 00    nopw   %cs:0x0(%rax,%rax,1)
```

Benchmark loop for spin_lock()/spin_unlock():

```
 140:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
 147:   ff c3                   inc    %ebx
 149:   e8 00 00 00 00          callq  ffffffff814d6c00 <_spin_lock>
 14e:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
 155:   e8 00 00 00 00          callq  ffffffff814d6f10 <_spin_unlock>
 15a:   81 fb 20 4e 00 00       cmp    $0x4e20,%ebx
 160:   75 de                   jne    140
```

Figure 4.8 Assembly listings for Intel Xeon benchmarks (spin lock loop content)

Table 4.3 Breakdown of cycles taken for spin lock disabling interrupts

Architecture	Spin lock (cycles)	IRQ save/restore (cycles)	Total (cycles)
Pentium 4	144	131	275
AMD Athlon(tm)64 X2	67	23	90
Intel Core2	57	43	100
Intel Xeon E5405	46	39	85
ARMv7 OMAP3[a]	132	45	177

a. Forced SMP configuration for test module.

Sequence read lock:

```
 330:   f3 90                   pause
 332:   89 f2                   mov    %esi,%edx
 334:   48 89 c8                mov    %rcx,%rax
 337:   a8 01                   test   $0x1,%al
 339:   75 f5                   jne    330
 33b:   39 15 00 00 00 00       cmp    %edx,0x0(%rip)
 341:   75 ef                   jne    332
 343:   ff c7                   inc    %edi
 345:   81 ff 20 4e 00 00       cmp    $0x4e20,%edi
 34b:   75 ea                   jne    337
```

Preemption disabling/enabling:

```
 3f8:   ff 43 1c                incl   0x1c(%rbx)
 3fb:   ff 4b 1c                decl   0x1c(%rbx)
 3fe:   41 f6 84 24 38 e0 ff    testb  $0x8,-0x1fc8(%r12)
 405:   ff 08
 407:   0f 85 a4 00 00 00       jne    4b1
 40d:   ff c5                   inc    %ebp
 40f:   81 fd 20 4e 00 00       cmp    $0x4e20,%ebp
 415:   75 e1                   jne    3f8 <init_module+0x3e8>
[...]
 4b1:   e8 00 00 00 00          callq  4b6 <preempt_schedule>
 4b6:   e9 52 ff ff ff          jmpq   40d
```

Figure 4.9 Assembly listings for Intel Xeon benchmarks (sequence lock and preemption disabling loop content)

Table 4.4 Breakdown of cycles taken for using a read *seqlock* and using a synchronized `CAS`

Architecture	*Seqlock* (cycles)	Sync. `CAS` (cycles)	Total (cycles)
Pentium 4	4	81	85
AMD Athlon(tm)64 X2	4	24	28
Intel Core2	3	24	27
Intel Xeon E5405	3	24	27
ARMv7 OMAP3[a]	73	71	144

a. Forced `SMP` configuration for test module.

Table 4.5 Breakdown of cycles taken for disabling preemption and using a local CAS

Architecture	Preemption disable/enable (cycles)	Local CAS (cycles)	Total (cycles)
Pentium 4	9	25	34
AMD Athlon(tm)64 X2	12	5	17
Intel Core2	12	8	20
Intel Xeon E5405	12	8	20
ARMv7 OMAP3[a]	10	11	21

a. Forced SMP configuration for test module.

Table 4.6 Speedup of tracing synchronization primitives compared to disabling interrupts and spin lock

Architecture	Spin lock disabling interrupts (speedup)	Sequence lock and CAS (speedup)	Preempt disabled and local CAS (speedup)
Pentium 4	1	3.2	8.1
AMD Athlon(tm)64 X2	1	3.2	5.3
Intel Core2	1	3.7	5.0
Intel Xeon E5405	1	3.1	4.3
ARMv7 OMAP3[a]	1	1.2	8.4

a. Forced SMP configuration for test module.

Chapter 5

Paper 2: Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing

Abstract

Studying execution of concurrent real-time online systems, to identify far-reaching and hard to reproduce latency and performance problems, requires a mechanism that is able to cope with large amounts of information extracted from execution traces, without disturbing the workload thereby causing the problematic behavior to become unreproducible.

In order to meet this low-disturbance characteristic, we created the `LTTng` kernel tracer. It is designed to make it possible, safe, and race-free to attach probes virtually anywhere in the operating system, including sites executed in non-maskable interrupt context.

In addition to being reentrant with respect to all kernel execution contexts, `LTTng` provides good performance and scalability mainly due to its use of per-CPU data structures, *local atomic operations* as main buffer synchronization primitive, and `RCU` (*Read-Copy Update*) mechanism to control tracing.

Given that kernel infrastructure used by the tracer could lead to infinite recursion if traced and typically require non-atomic synchronization, this paper proposes an asynchronous mechanism to inform the kernel that a buffer is ready to be read. This ensures that the tracing site does not require any kernel primitive and therefore protects from infinite recursion.

This paper presents the core of `LTTng`'s buffering algorithms and benchmarks its performance.

5.1 Introduction

Performance monitoring of multiprocessor high-performance computers deployed as production systems (e.g. Google platform), requires tools to report what is being executed on the system. This provides better understanding of complex multi-threaded and multi-processes application interactions with the kernel.

Tracing the most important kernel events has been done for decades in the embedded field to reveal useful information about program behavior and performance. The main distinctive aspect of multiprocessor system tracing is the complexity added by time-synchronization across cores. Additionally, tracing of interactions between processes and the kernel generates a high volume of information.

Allowing wide instrumentation coverage of the kernel code can prove to be especially tricky, given the concurrency of multiple execution contexts and multiple processors. In addition to being able to trace a large portion of the executable code, another key element expected from a kernel tracer is to be low-overhead and not disturb the normal system behavior. Ideally, a problematic workload should be repeatable both under normal conditions and under tracing, without suffering from the observer effect caused by the tracer. The `LTTng` [27] tracer (available at: `http://www.lttng.org`) has been developed with these two principal goals in mind: provide good instrumentation coverage and minimize observer effect on the traced system.

A state of the art review is first presented, showing how the various tracer requirements bring their respective design and core synchronization primitive choice in different directions and how `LTTng` differs. The `K42` tracer will be studied in detail, given the significant contribution of this research operating system. This paper will discuss some limitations present in the `K42` lockless algorithm, which will bring us to the need for a new buffer management model. The algorithms and equations required to manage the buffers, ensuring complete atomicity of the probe, will then be detailed. The scalability of the approach will also be discussed, explaining the motivations behind the choice of per-CPU data structures to provide good processor cache locality. Performance tests will show how the tracer performs under various workloads at the macro-benchmark and micro-benchmark levels.

5.2 State of the art

In this section, we will first present a review of the requirements from the target LTTng user-base in terms of tracing. This is a summary of field work done to identify those requirements from real-world Linux users. Then, we will present the state-of-the-art open source tracers. For each of these, their target usage scenarios will be presented along with the requirements imposed. Finally, we will study in detail the tracer in K42, which is the closest to LTTng requirements, explaining where LTTng brings new contributions.

Previous work published in 2007 at the Linux Symposium [24] and Europar [10] presented the user-requirements for kernel tracing that are driving the LTTng effort. They explain how tracing is expected to be used by Linux end-users, developers, technical support providers and system administrators. The following list summarizes this information and lists which Linux distributions integrate LTTng:

- Large online service companies such as Google need a tool to monitor their production servers and to help them solve hard to reproduce problems. Google have had success with such tracing approaches to fix rarely occuring disk delay issues and virtual memory related issues. They need the tracer to have a minimal performance footprint.
- IBM Research looked into the debugging of commercial scale-out applications, which are being increasingly used to split large server workloads. They used LTTng successfully to solve a distributed filesystem-related issue.
- Autodesk, in the development of their next-generation of Linux audio/video edition applications, used LTTng extensively to solve soft real-time issues they faced.
- Wind River includes LTTng in their Linux distribution so their clients, already familiar with Wind River VxWorks tracing solutions, can benefit from the same kind of features they have relied on for a long time.
- Montavista has integrated LTTng in their Carrier Grade Linux Edition 5.0 for the same reasons.
- SuSE is currently integrating LTTng in their next SLES real-time distribution, because their clients, asking for solutions supporting a kernel closer to real-time, need such tools to debug their problems.
- A project between Ericsson, Defence R&D Canada, NSERC and various univer-

sities is just starting. It aims at monitoring and debugging multi-core systems, providing tools to automate system behavior analysis.

– Siemens has been using `LTTng` internally for quite some time now [50].

We will now look at the existing tracing solutions for which detailed design and implementation documentation is publicly available. This study will focus on tracers available under an open-source license, given that closed-source tracers do not provide such detailed documentation. The requirements fulfilled by each tracer as well as their design choices will be exposed. Areas in which `LTTng` requirements differ from these tracers will be outlined.

`DTrace` [33], first made available in 2003 and formally released as part of Sun's Solaris 10 in 2005, aims at providing information to users about the way their operating system and applications behave by executing scripts performing specialized analysis. It also provides the infrastructure to collect the event trace into memory buffers, but aims at moderate event production rates. It disables interrupts to protect the tracer from concurrent execution contexts on the same processor and a sequence lock to protect the clock source from concurrent modifications.

`SystemTAP` [40] provides scriptable probes which can be connected on top of Markers, Tracepoints or Kprobes [43]. It is designed to provide a safe language to express the scripts to run at the instrumentation site, but does not aim at optimizing probe performance for high data volume, since it was originally designed to gather information exclusively from Kprobes breakpoints and therefore expects the user to carefully filter out the unneeded information to diminish the probe effect. It disables interrupts and takes a busy-spinning lock to synchronize concurrent tracing site execution. The `LKET` project (Linux Kernel Event Tracer) re-used the `SystemTAP` infrastructure to trace events, but reached limited performance results given the fact that it shared much of `SystemTAP`'s heavy synchronization.

Ftrace, started in 2009 by Ingo Molnar, aims primarily at kernel tracing suited for kernel developer's needs. It primarily lets specialized trace analysis modules run in kernel-space to generate either a trace or analysis output, available to the user in text format. It also integrates binary buffer data extraction which aims at providing efficient data output. It is currently based on per-cpu busy-spinning locks and interrupt disabling to protect the tracer against concurrent execution contexts. It is currently evolving to a lockless buffering scheme. In comparison, the work on `LTTng` presented in this paper started back in 2005.

The K42 [38] project is a research operating system developed mostly between 1999 and 2006 by IBM Research. It targeted primarily large multiprocessor machines with high scalability and performance requirements. It contained a built-in tracer simply named "trace", which was an element integrated to the kernel design per se. The systems targeted by K42 and use of lockless buffering algorithms with atomic operations are similar to LTTng.

From a design point of view, a major difference between this research-oriented tracer and LTTng is that the latter aims at being deployed on multi-user Linux systems, where security is a concern. Therefore, simply sharing a per-cpu buffer, available both for reading and writing by the kernel and any user process, would not be acceptable on production systems. Also, in terms of synchronization, K42's tracer implementation ties trace extraction user-space threads to the processor on which the information is collected. Although this removes the need for synchronization, it also implies that a relatively idle processor cannot contribute to the overall tracing effort when some processors are busier. Regarding CPU hotplug support, which is present in Linux, an approach where the only threads able to extract the buffer data would be tied to the local processor would not allow trace extraction in the event a processor would go offline. Adding support for cross-CPU data reader support would involve adding the proper memory barriers to the tracer.

Then, more importantly for the focus of this paper, studying in depth the lockless atomic buffering scheme found in K42 indicates the presence of a race condition where data corruption is possible. It must be pointed out that, given the fact that the K42 tracer uses large buffers compared to the typical event size, this race is unlikely to happen, but could become more frequent if the buffer size is made smaller or larger events were written, which LTTng tracer's flexibility permits.

The K42 tracer [39] divides the memory reserved for tracing a particular CPU into buffers. This maps to the sub-buffer concept presented in the LTTng design. In comparison, LTTng uses the "buffer" name to identify the set of sub-buffers which are parts of the circular buffer. In the present discussion, the term "buffer" will have the K42 semantic, but the rest of the paper will use the LTTng semantic. The K42 scheme uses a lockless buffer-space management algorithm based on a reserve-commit semantic. Space is first reserved atomically in the buffer, and then the data write and commit are done out-of-order with respect to local interrupts. It uses a *buffersProduced* count, which counts the number of buffers produced by the tracer, a

buffersConsumed count, to keep track of the number of buffers read and a per-buffer *bufferCount*, to keep track of the amount of information committed into each buffer.

In the `K42` scheme, the *buffersProduced* count is incremented upon buffer space *reservation* for an event crossing a buffer boundary. If other out-of-order writes are causing the current and previous sub-buffer's commit counts to be a modulo of buffer size (because they would still be fully uncommitted), the user-space data consumption thread can read non-committed (invalid) data because the *buffersProduced* would make an uncommitted buffer appear as fully committed. This is a basic algorithmic flaw that `LTTng` fixes by using a free-running per sub-buffer *commit count* and by using a different *buffer full* criterion which depends on the difference between the *write count* (global to the whole buffer) and its associated per-subbuffer *commit count*, as detailed in Equation 5.1 in Section 5.4.2.

The formal verification performed by modeling the `LTTng` algorithms and using the Spin model-checker increases the level of confidence that such corner-cases are correctly handled.

5.3 Design of LTTng

Tracing an operating system kernel poses interesting problems related to the *observer effect*. In fact, tracing performed at the software level requires modifying the execution flow of the traced system and therefore modifies its behavior and performance. When deciding what code will be executed when the instrumentation is reached, each execution context concerned must be taken into account.

This section describes how `LTTng` is designed to deal with kernel tracing, satisfying the constraints associated with *synchronization* of data structures while running in any *execution context*, avoiding *kernel recursion* and inducing a very small *performance impact*. It details a complete buffering synchronization scheme.

This section starts with a high-level overview of the tracer design. It is followed by a more detailed presentation of the *Channel* component, an highly-efficient data transport pipe. Synchronization of trace *Control* data structures, allowing tracing configuration, is then exposed. This leads us to the *Data Flow* presentation as seen from the tracing probe perspective. Finally, the *Atomic Buffering Scheme* section details the core of `LTTng` concurrency management, which brings innovative algorithms to deal with write concurrency in circular memory buffers.

5.3.1 Components overview

Starting with a high-level perspective on the tracer design, Figure 5.1 presents the component interactions across the boundary between kernel-space and user-space.

Kernel core and kernel modules are *instrumented* either statically at the source-code level with the Linux Kernel Markers and Tracepoints or dynamically with Kprobes. Each instrumentation site identifies kernel code and module code which must be traced upon execution. Both static and dynamic instrumentation can be activated at runtime on a per-site basis to individually enable each event type. An event maps to a set of functionally equivalent instrumentation sites.

When an instrumented code site is executed, the `LTTng` *probe* is called if the instrumentation site is activated. The probe reads the *trace session* status and writes an event to the *channels*.

Trace sessions contains the tracing configuration data and pointers to multiple *channels*. Although only one session is represented in Figure 5.1, there can be many `trace sessions` concurrently active, each with its own trace configuration and its own set of `channels`. Configuration data determines if the trace session is active or not and which event filters should be applied.

From a high-level perspective, a channel can be seen as an information pipe with specific characteristics configured at trace session creation time. Buffer size, tracing mode (*flight recorder* or *non-overwrite*) and buffer flush period can be specified on a per-channel basis. These options will be detailed in Section 5.3.2.

`DebugFS` is a virtual filesystem providing an interface to control kernel debugging and export data from kernel-space to user-space. The trace session and channel data structures are organised as `DebugFS` files to let `lttctl` and `lttd` interact with them.

The user-space program `lttctl` is a command-line interface interacting with the `DebugFS` file system to control kernel tracing. It configures the trace session before tracing starts and is responsible for starting and stopping trace sessions.

The user-space daemon `lttd` also interacts with `DebugFS` to extract the channels to disk or network storage. This daemon is only responsible for data extraction; this daemon has absolutely no direct interaction with trace sessions.

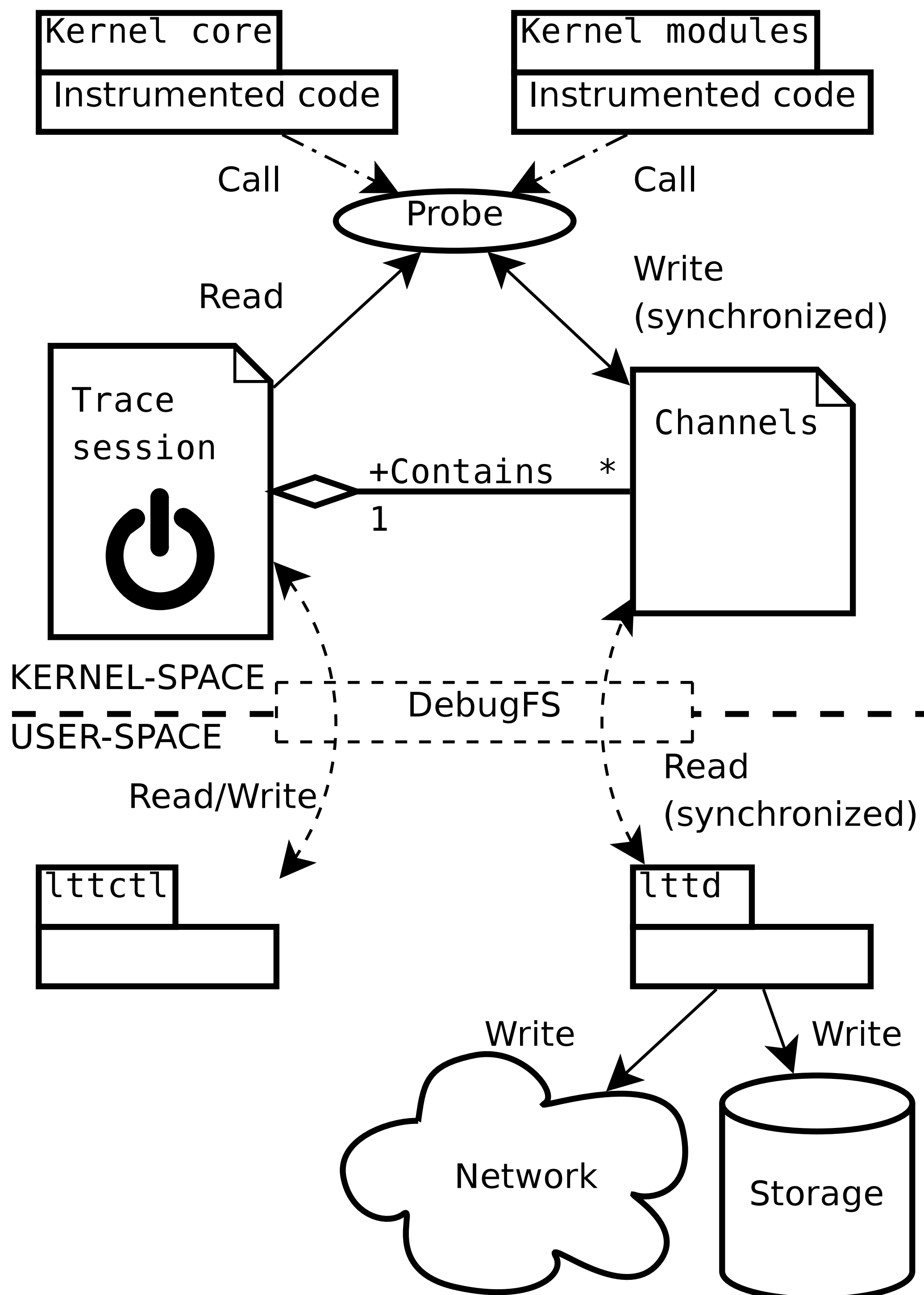

Figure 5.1 Tracer components overview

5.3.2 Channels

After the high-level tracer presentation, let's focus on the *Channel* components. They are presented in Figure 5.2.

A channel is a pipe between an information producer and consumer (producer and writer as well as consumer and reader will be respectively used as synonyms through this paper). It serves as a buffer to move data efficiently. It consists of one buffer per CPU to ensure cache locality and eliminate false-sharing. Each buffer is made of many sub-buffers where *slots* are reserved sequentially. Each sub-buffer is exported by the `lttd` daemon to disk or to the network separately.

A *slot* is a sub-buffer region reserved for exclusive write access by a probe. This space is reserved to write either a *sub-buffer header* or an *event header and payload*. Figure 5.2 shows space being reserved. On CPU 0, space is reserved in sub-buffer 0 following event 0. In this buffer, the *header* and *event 0* elements have been complelety written to the buffer. The grey area represents *slots* for which associated commit count increment has been done. Committing a reserved *slot* makes it available for reading. On CPU n, a slot is reserved in sub-buffer 0 but is still uncommitted. It is however followed by a committed event. This is possible due to the non serial nature of event write and commit operations. This situation happens when execution is interrupted between space reservation and commit count update and another event must be written by the interrupt handler. Sub-buffer 1, belonging to CPU 0, shows a fully committed sub-buffer ready for reading.

Events written in a reserved *slot* are made of a *header* and a variable-sized *payload*. The *header* contains information containing the time stamp associated with the event and the event type (an integer identifier). The event type information allows parsing the *payload* and determining its size. The maximum *slot* size is bounded by the sub-buffer size.

Channels can be configured in either of the two following tracing modes. *Flight recorder* tracing is a mode where the oldest buffer data is overwritten when a buffer is full. Conversely, *non-overwrite* tracing discards (and counts) events when a buffer is full. Those discarded events are counted to evaluate tracing accuracy. These counters are recorded in each sub-buffer header to allow identifying which trace region suffered from event loss. The former mode is made to capture a snapshot of the system preceding execution at a given point. The latter is made to collect the entire execution trace over a period of time.

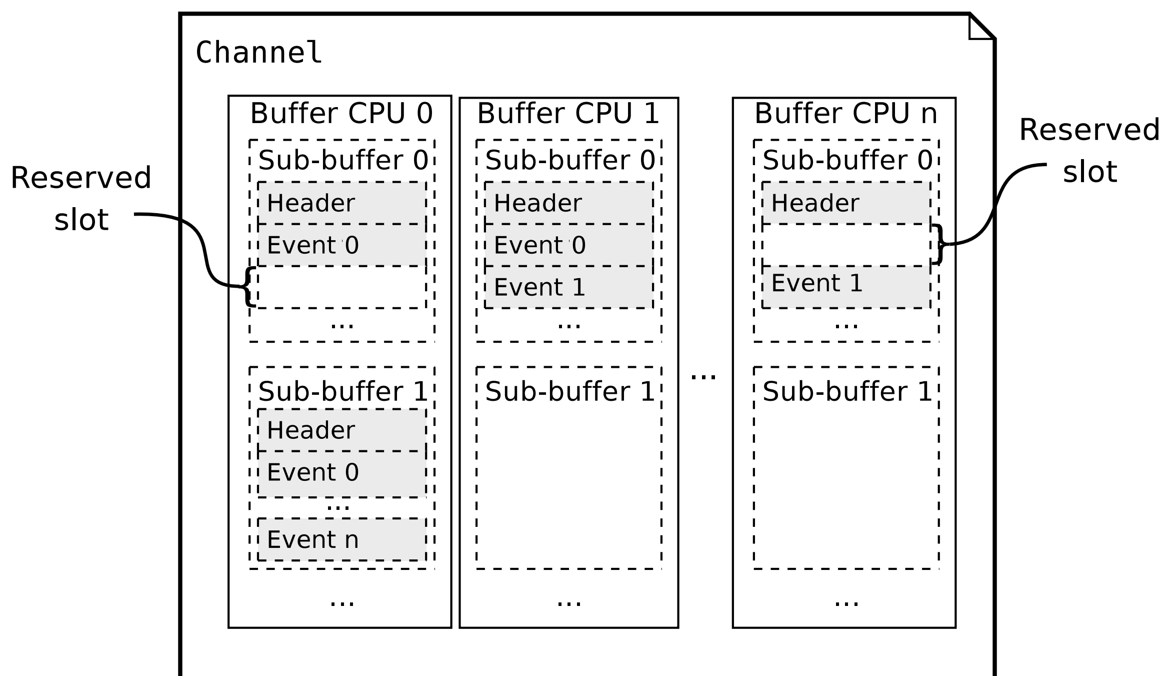

Figure 5.2 Channel components

5.3.3 Control

This section presents interactions with the *trace session* data structure depicted in Figure 5.1 along with the required synchronization.

Information controlling tracing includes, for instance, the channel location and a flag to specify if a specific set of buffers is active for tracing. This provides flexibility so users can tune the tracer following their system's workload. They can determine how much memory space must be reserved for buffering the tracing data. They can also configure each channel in *flight recorder* or *non-overwrite* mode. Selection of tracing behavior can be tuned on a per-channel basis. The channel identifier forms an intrinsic event categorization.

Tracing control operations include creating a new trace session, starting or stopping tracing, and freeing a trace session. Providing an external callback to be called for per-trace filtering is also possible. Upon new trace session creation, parameters must be set such as channel's buffer size, number of sub-buffers per buffer, tracing mode and if tracing is enabled for each information channel. This tracing control information is contained in a list of active trace sessions.

Tracing control is done by a kernel module, `ltt-tracer`, which updates the `RCU` list of active trace sessions. It protects the update operation from concurrent writes by holding a mutex. Two types of data structure modifications can be done: the data element can be updated atomically, in which case it is safe to perform the modification without copying the complete trace control data structure as long as the mutex is held. Non-atomic updates must be done on a copy of the trace control structure, followed by a replacement of the old copy in the list by two successive pointer changes in this precise order: first setting the pointer to next element within the new copy and then setting the pointer to the new copy in the previous element. Then it waits for quiescent state, which allows memory reclamation of the old data structure. This ensures no active data structure readers, the probes, still hold a reference to the old structure when it is freed.

Modification of buffer data structures by the *ltt-tracer* kernel module is only done upon new trace session creation and deletion. Once the trace is started, the module won't modify these structures until tracing is stopped. It makes sure only the data producers and consumers will touch the buffer management structures.

In order to provide the ability to export tracing information as a live stream, one must ensure a maximum latency between the moment the event is written to the memory buffers and the moment it is ready to be read by the consumer. However, because the information is only made available for reading after a sub-buffer has been filled, a low event rate channel might never be ready for reading until the final buffer flush is done when tracing is stopped.

To get around this problem, `LTTng` implements a per-CPU sub-buffer flush function which can be executed concurrently with tracing. It shares many similarities with tracing an event. However, it won't flush an empty sub-buffer because there is no information to send and it does not reserve space in the buffer. The only supplementary step required to stream the information is to call the buffer flush for each channel periodically in a per-CPU timer interrupt.

5.3.4 Probe Data Flow

The tracing data flow from the probe perspective is illustrated in Figure 5.3. This figure includes all data sources and sinks, including those which are not part of the tracer per se, such as kernel data structures and hardware time stamps.

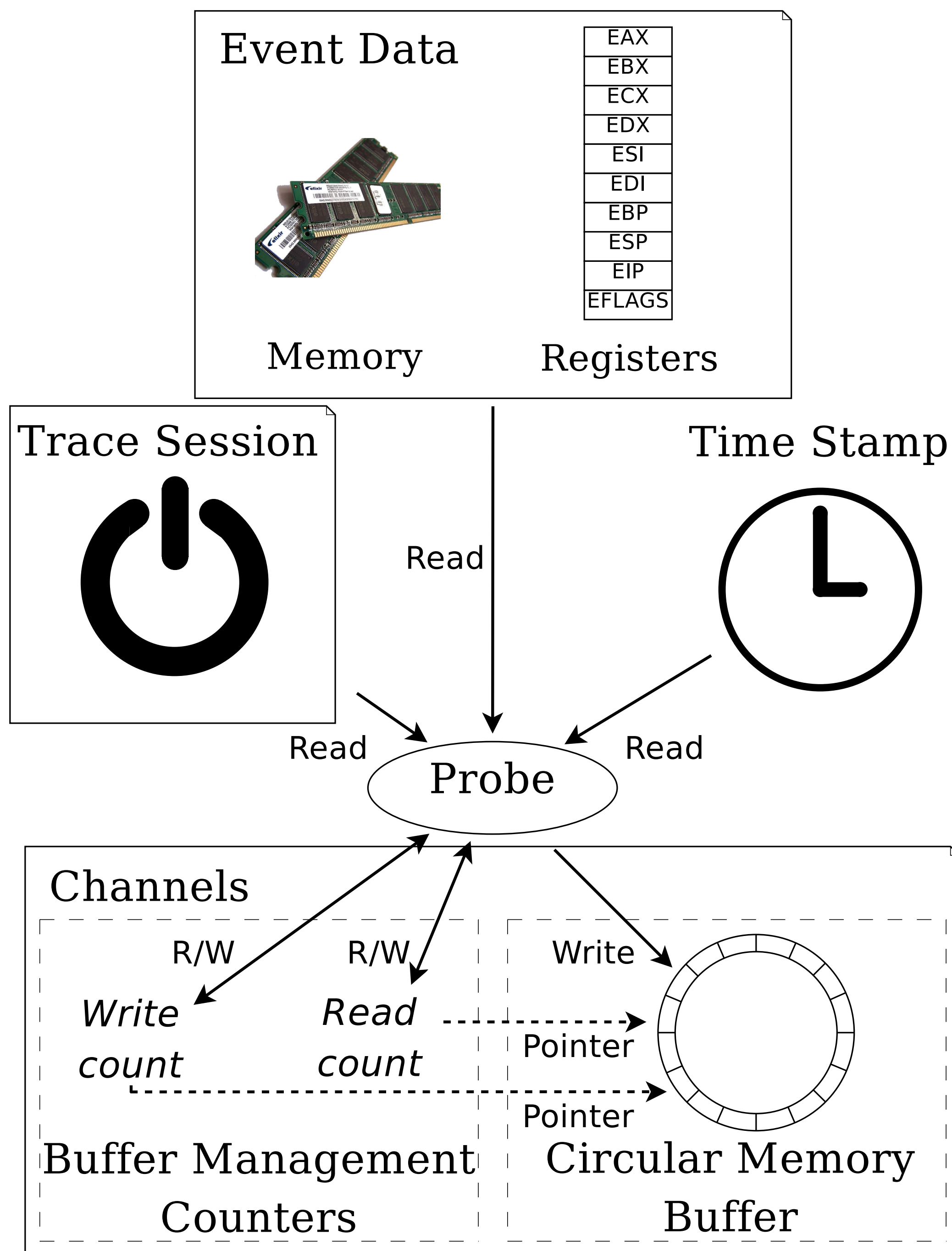

Figure 5.3 Probe data flow

A probe takes event data from registers, the stack, or from memory every time the instrumented kernel execution site is reached. A time stamp is then associated with this information to form an event, identified by an event ID. The tracing control information is read to know which channel is concerned by the information. Finally, the resulting event is serialized and written to a circular buffer to be later exported outside of kernel-space. The channels offer a producer-consumer semantic.

Instrumentation can be inserted either statically, at the source-code level, or dynamically, using a breakpoint. The former allows building instrumentation into the software and therefore identify key instrumentation sites, maintaining a stable API. It can also restrain the compiler from optimizing away variables needed at the instrumented site. However, in order to benefit from flexible live instrumentation insertion, without recompilation and reboot, it might be adequate to pay the performance cost associated with a breakpoint, but one must accept that the local variables might be optimized away and that the kernel debug information must be kept around.

Source-code level instrumentation, enabled at runtime, is currently provided by *Tracepoints* [51] and *Linux Kernel Markers* [52], developed as part of the `LTTng` project and merged into the mainline Linux kernel. Dynamic instrumentation, based on breakpoints, is provided in the Linux kernel by *Kprobes* [43] for many architectures. `LTTng`, `SystemTAP` and `DTrace` all use a combination of dynamic and static instrumentation. The details about the different instrumentation mechanisms are not, however, the focus of this paper. The following section presents channel ring-buffer synchronization.

5.4 Atomic Buffering Scheme

The atomic buffering scheme implemented in `LTTng` allows the probe to produce data in circular buffers with a buffer-space reservation mechanism which ensures correct reentrancy with respect to asynchronous event sources. These include maskable and non-maskable interrupts (`NMIs`). Preemption [1] is temporarily disabled around the tracing site to make sure no thread migration to a different CPU can occur in the middle of probe execution.

Section 5.4.1 first presents the data structures used to synchronize the buffering

1. With fully-preemptible Linux kernels (*CONFIG_PREEMPT=y*), the scheduler can preempted threads running in kernel context to run another thread.

scheme. Then, algorithms performing interactions between producer and consumer are discussed respectively in sections 5.4.2, 5.4.3, 5.4.3, 5.4.3 and 5.4.3.

5.4.1 Atomic data structures

On SMP (*Symmetric Multiprocessing*) systems, some instructions are designed to update data structures in one single indivisible step. Those are called atomic operations. To properly implement the semantic carried by these low-level primitives, memory barriers are required on some architecture (this is the case for PowerPC and ARMv7 for instance). For the x86 architecture family, these memory barriers are implicit, but a special lock prefix is required before these instructions to synchronize multiprocessor access. However, to diminish performance overhead of the tracer fast-path, we remove memory barriers and use atomic operations only synchronized with respect to the local processor due to their lower overhead than those synchronized across cores. They are the only instructions allowed to modify the per-CPU data, to ensure reentrancy with `NMI` context.

The main restriction that must be observed when using such operations is to disable preemption around all access to these variables, to ensure threads are not migrated from one core to another between the moment the reference is read and the atomic access. This ensures no remote core accesses the variable with SMP-unsafe operations.

The two atomic instructions required are the `CAS` (*Compare-And-Swap*) and a simple atomic increment. Figure 5.4 shows the data structures being modified by those *local atomic operations*. Each per-CPU buffer has a control structure which contains the *write count*, the *read count*, and an array of *commit counts* and *commit seq* counters [2]. The counters *commit count* keep track of the amount of data committed in a sub-buffer using a lightweight increment instruction. The *commit seq* counters are updated with a concurrency-aware synchronization primitive each time a sub-buffer is filled.

A local `CAS` is used on the *write count* to update the counter of reserved buffer space. This operation ensures space reservation is done atomically with respect to other execution contexts running on the same CPU. The atomic add instruction is used to increment the per sub-buffer *commit count*, which identifies how much

2. The size of this array is the number of sub-buffers.

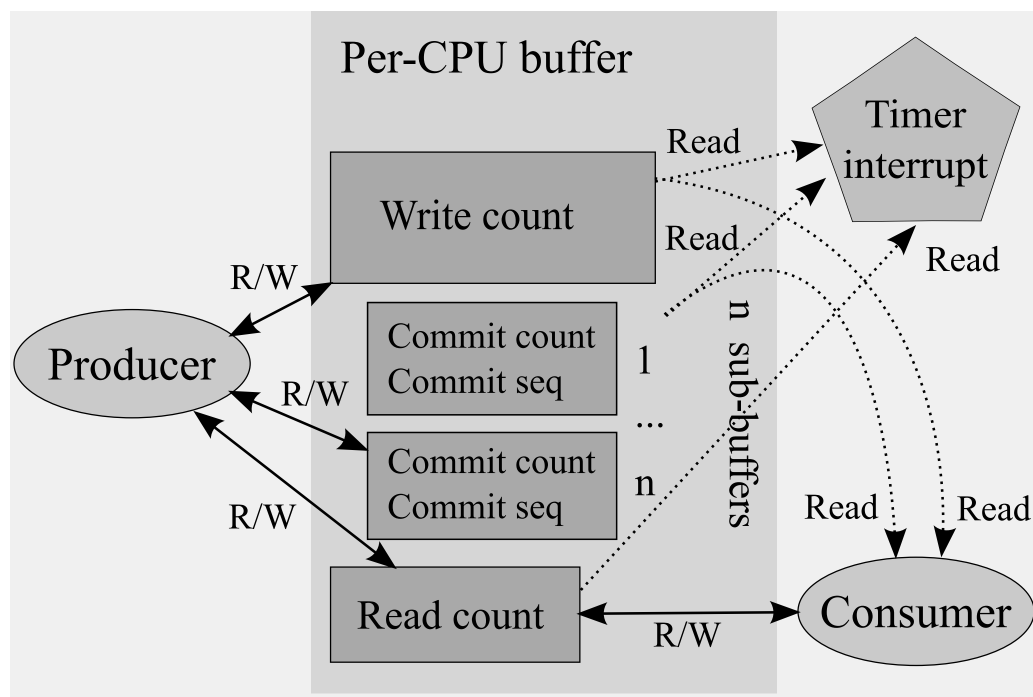

Figure 5.4 Producer-consumer synchronization

information has actually been written in each sub-buffer.

The sub-buffer size and the number of sub-buffers within a buffer are limited to powers of 2 for two reasons. First, using bitwise operations to access the sub-buffer offset and sub-buffer index is faster than the modulo and division. The second reason is more subtle: although the `CAS` operation could detect 32 or 64-bits overflows and deal with them correctly before they happen by resetting to 0, the *commit count* atomic add will eventually overflow the 32 or 64-bits counters, which adds an inherent power of 2 modulo that would be problematic if the sub-buffer size would not be power of 2.

On the reader side, the *read count* is updated using a standard SMP-aware `CAS` operation. This is required because the reader thread can read sub-buffers from buffers belonging to a remote CPU. It is designed to ensure that a traced workload executed on a very busy CPU can be extracted by other CPUs which have more idle time. Having the reader on a remote CPU requires SMP-aware `CAS`. This allows the writer to push the reader position when the buffer is configured in *flight recorder* mode. The performance cost of the SMP-aware operation is not critical because updating the *read*

count is only done once a whole sub-buffer has been read by the consumer, or when the writer needs to push the reader at sub-buffer switch, when a buffer is configured in *flight recorder* mode. Concurrency between many reader threads is managed by using a reference count on file open/release, which only lets a single process open the file, and by requiring that the user-space application reads the sub-buffers from only one execution thread at a time. Mutual exclusion of many reader threads is left to the user-space caller, because it must encompass a sequence of multiple system calls. Holding a kernel mutex is not allowed when returning to user-space.

5.4.2 Equations

This section presents equations determining buffer state. These are used by algorithms presented in Section 5.4.3.

These equations extensively use modulo arithmetic to consider physical counter overflows. On 64-bits architectures, equations are in modulo 2^{64}. On 32-bits architectures, they are modulo 2^{32}.

We first define the following basic operations. Let's define

- $|x|$ as length of x.
- $a \bmod b$ as modulo operation (remainer of $\frac{a}{b}$).
- $\underset{m}{\overset{n}{\mathcal{M}}}(x)$ as x bitwise AND $00\ldots0\underbrace{11\ldots1}_{n-m}\underbrace{00\ldots0}_{m}$,
  formally: $(x \bmod 2^n) - (x \bmod 2^m)$.

We define the following constants. Let

– $|\texttt{sbuf}|$ be the size of a sub-buffer.
 (power of 2)

– $|\texttt{buf}|$ be the size of a buffer.
 (power of 2)

– $sbfbits = \lg_2(|\texttt{sbuf}|)$.

– $bfbits = \lg_2(|\texttt{buf}|)$.

– $nsbbits = bfbits - sbfbits$.

– $wbits$ be the architecture word size in bits.
 (32 or 64 bits)

We have the following variables. Let

– $wcnt$ be write counter mod 2^{wbits}.

– $rcnt$ be read counter mod 2^{wbits}.

– $wcommit$ be the commit counter *commit seq* mod 2^{wbits} belonging to the sub-buffer where $wcnt$ is located.

– $rcommit$ be the commit counter *commit seq* mod 2^{wbits} belonging to the sub-buffer where $rcnt$ is located.

Less than one complete sub-buffer is available for writing when Equation 5.1 is satisfied. It verifies that the difference between the number of sub-buffers produced and the number of sub-buffers consumed in the ring buffer is greater or equal to the number of sub-buffers per buffer. If this equation is satisfied at buffer switch, it means the buffer is full.

$$\mathop{\mathcal{M}}_{sbfbits}^{wbits}(wcnt) - \mathop{\mathcal{M}}_{sbfbits}^{wbits}(rcnt) \geq |\texttt{buf}| \tag{5.1}$$

Write counter and read counter masks are illustrated by Figure 5.5. These masks are applied to $wcnt$ and $rcnt$.

A buffer contains at least one sub-buffer ready to read when Equation 5.2 is satisfied. The left side of this equation takes the number of buffers reserved so far, masks out the current buffer offset and divides the result by the number of sub-buffers per buffer. This division ensures the left side of the equation represents the number of sub-buffers reserved. The right side of this equation takes the commit count to which $rcnt$ points and substracts $|\texttt{sbuf}|$ from it. It is masked to clear the top bits, which ensures both sides of the equation overflow at the same value. This is required

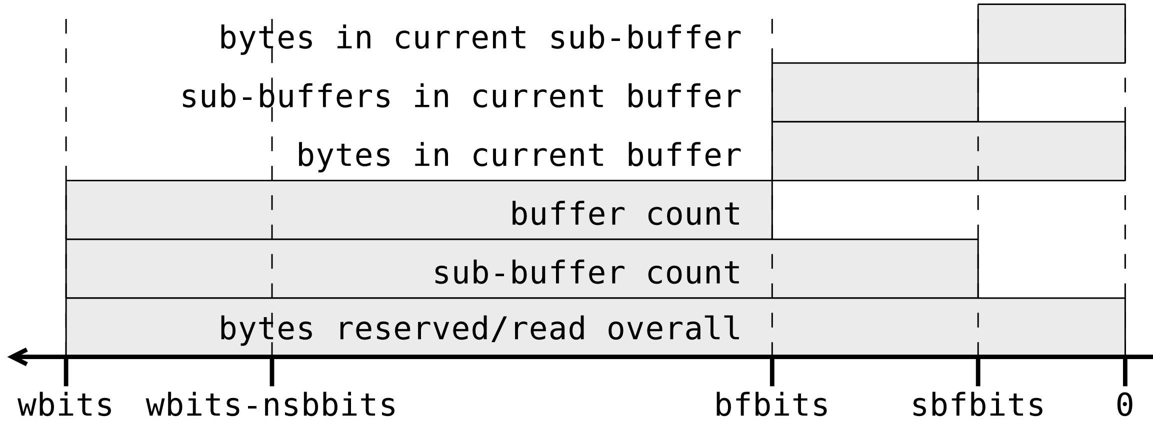

Figure 5.5 Write and read counter masks

because *rcnt* reaches a 2^{wbits} overflow *sbfnb* times more often than the per-subbuffer *rcommit* counters. $|\texttt{sbuf}|$ is substracted from *rcommit* because we need to know when the *commit seq* is one whole sub-buffer ahead of the read count.

$$\frac{\underset{bfbits}{\overset{wbits}{\mathcal{M}}}(rcnt)}{2^{nsbbits}} = \underset{0}{\overset{wbits-nsbbits}{\mathcal{M}}}(rcommit - |\texttt{sbuf}|) \tag{5.2}$$

The sub-buffer corresponding to *wcnt* is in a fully committed state when Equation 5.3 is satisfied. Its negation is used to detect a situation where an amount of data sufficient to overflow the buffer is written by concurrent execution contexts running between a reserve-commit pair.

$$\frac{\underset{bfbits}{\overset{wbits}{\mathcal{M}}}(wcnt)}{2^{nsbbits}} = \underset{0}{\overset{wbits-nsbbits}{\mathcal{M}}}(wcommit) \tag{5.3}$$

Commit counter masks are illustrated by Figure 5.6. These masks are applied to *rcommit* and *wcommit*.

The sub-buffer corresponding to *rcnt* is being written when Equation 5.4 is satisfied. It verifies that the number of sub-buffers produced and consumed are equal.

$$\underset{sbfbits}{\overset{wbits}{\mathcal{M}}}(wcnt) = \underset{sbfbits}{\overset{wbits}{\mathcal{M}}}(rcnt) \tag{5.4}$$

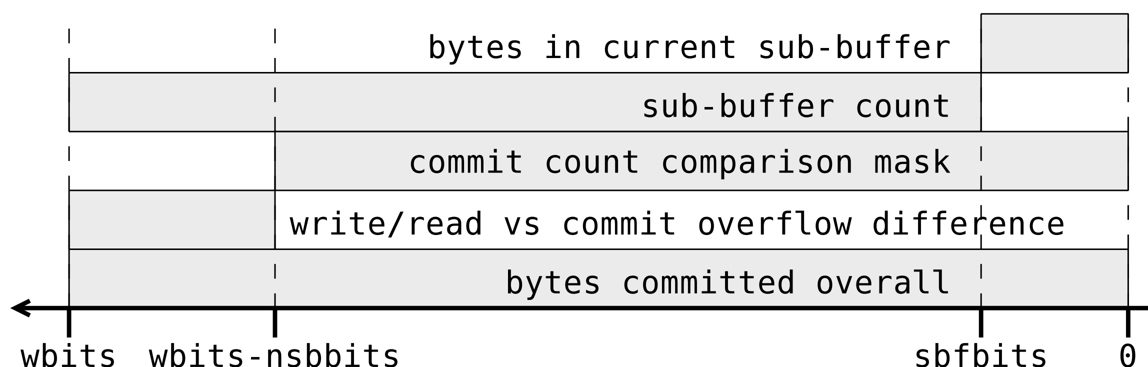

Figure 5.6 Commit counter masks

5.4.3 Algorithms

Algorithms used to synchronize the producer and consumer are presented in this section. It is followed by a presentation of the asynchronous buffer delivery algorithm.

Producer

This section presents the algorithms used by the information producer, the *probe*, to synchronize its *slot* reservation within the *channels*.

The overall call-graph presented in this section can be summarized as follow. When an event is to be written, space is reserved by calling ReserveSlot(), which calls TryReserveSlot() in a loop until it succeeds. Then, PushReader(), SwitchOldSubbuf(), SwitchNewSubbuf() and EndSwitchCurrent() (not expanded in this paper for brevity) are executed out-of-order to deal with sub-buffer change. After the event data is written to the slot, CommitSlot() is called to increment the commit counter.

The *write count* and *read count* variables have the largest size accessible atomically by the architecture, typically 32 or 64 bits. Since, by design, the sub-buffer size and the number of sub-buffers within a buffer are powers of two, a LSB (Least Significant Bit) mask can be used on those counters to extract the offset within the buffer. The MSBs (Most Significant Bits) are used detecting the improbable occurrence of a complete buffer wrap-around nested on top of the local `CAS` loop in *flight recorder* mode. Such overflow, if undetected, could cause a timestamp to go backward in a

buffer.

Such wrap-around could happen if many interrupts nest back-to-back on top of a `CAS` loop. A worse-case scenario would be to have back-to-back nested interrupts generating enough data to fill the buffer (typically 2 MiB in size) and bring the write count back to the same offset in the buffer. The `CAS` loop uses the most significant counter bits to detect this situation. On 32-bits architectures, it permits to detect counter overflow up to 4 GiB worth of buffer data. On 64-bits architectures, it detects up to 16.8 million TiB worth of data written while nested over a `CAS` loop execution. Given that this amount of trace data would have to be generated by interrupt handlers continuously interrupting the probe, we would consider an operating system facing such an interrupt rate to be unusable. As an example of existing code with similar assumptions, the Linux kernel sequence lock, used to synchronize the time-base, is made of a sequence counter also subject to overflow.

Slot reservation, presented in TryReserveSlot() and ReserveSlot() is performed as follow. From a high-level perspective, the producer depends on the *read count* and *write count* difference to know if space is still available in the buffers. If no space is available in *non-overwrite* mode, the event lost count is incremented and the event is discarded. In *flight recorder* mode, the next sub-buffer is overwritten by *pushing* the reader. Variables *write count*, *read count* and the *commit seq* array are used to keep track of the respective position of the writer and the reader gracefully with respect to counter overflow. Equations 5.1, 5.2, 5.3 and 5.4 are used to verify the state of the buffer.

The *write count* is updated atomically by the producer to reserve space in the sub-buffer. In order to apply monotonically increasing time stamps to events which are physically consecutive in the buffer, the time stamp is read within the `CAS` loop. This ensures that no space reservation succeeds between the time-stamp register read and the atomic space reservation, and therefore ensures that a successful buffer-space reservation and time-stamp read are indivisible from one another from a CPU's perspective. Such mechanisms to make many instructions appear to execute atomically is however limited to operations not having side-effects outside of the variables located on the stack or in registers which can be re-executed upon failure, except for the single `CAS` operation which has side-effects when it succeeds. It is therefore mostly limited to read operations and the computation of the required *slot* size for the event.

Once space is reserved, the remaining operations are done out-of-order. This

Algorithm: TRYRESERVESLOT(*payload_size*)
Require: An integer *payload_size* ≥ 0.

1: Read *write_count*
2: Read *time_stamp_counter*
3: Calculate required *slot_size*
4: Calculate *slot_offset*
5: **if** *slot_offset* is at beginnig of sub-buffer **then**
6:     **if** Negation of Eqn. 5.3 **then**
7:         Increment event lost count
8:         *slot_size* = *FAIL*
9:         **return** *slot_size*
10:     **end if**
11:     **if** Eqn. 5.1 (in non-overwrite mode) **then**
12:         Increment event lost count
13:         *slot_size* = *FAIL*
14:         **return** *slot_size*
15:     **end if**
16: **end if**
17: Update *buffer_switch_flags*
18: **return** $<$ *slot_size*, *slot_offset*, *buffer_switch_flags* $>$

Algorithm 5.1 TRYRESERVESLOT(*payload_size*)

means that if an interrupt nests over a probe, it will reserve a buffer slot next to the one being written to by the interrupted thread, will write its event data in its own reserved slot and will atomically increment the *commit count* before returning to the previous probe stack. When a slot has been completely written to, the COMMITSLOT() algorithm is used to update the *commit count*. It is also responsible for clearing the sub-buffer reference flag if the sub-buffer is filled and updating *commit seq*.

There is one *commit seq* per sub-buffer. It also increments forever in the same way the *write count* does, with the difference that it only counts the per-subbuffer bytes committed rather than the number of bytes reserved for the whole buffer. The difference between the `write count` MSBs divided by the number of sub-buffers and the *commit seq* MSBs (with the highest bits corresponding to the number of sub-buffers set to zero) indicates if the commit count LSBs represent an empty, partially

3. The compiler barrier will be promoted to a write memory barrier by an interprocessor interrupt sent by the read-side READGETSUBBUF(), as explained thoroughly in Section 5.4.4.

Algorithm: ReserveSlot(*payload_size*)

Require: An integer $payload_size \geq 0$

Ensure: *slot_offset* is the only reference to the slot during all the reserve and commit process, the slot is reserved atomically, time stamps of physically consecutive slots are always incrementing.

1: **repeat**
2:     $<$*slot_size*, *slot_offset*, *buffer_switch_flags*$>$ $= TryReserveSlot(payload_size)$
3:     **if** $slot_size = FAIL$ **then**
4:         **return** `FAIL`
5:     **end if**
6: **until** `CAS of` *write_count* `succeeds`
7: PushReader()
8: `Set reference flag in pointer to current sub-buffer. Indicates that the writer is using this sub-buffer.`
9: SwitchOldSubbuf()
10: SwitchNewSubbuf()
11: EndSwitchCurrent()
12: `return` $<$*slot_size*`,` *slot_offset*$>$

Algorithm 5.2 ReserveSlot(*payload_size*)

Algorithm: CommitSlot(*slot_size, slot_offset*)

Require: An integer $slot_size > 0$ and the *slot_offset*

1: Compiler barrier [3]
2: Issue *local_add()* to increment *commit_count* of *slot_size*
3: **if** Eqn. 5.3 **then**
4:     $commit_seq_old = commit_seq$
5:     **while** $commit_seq_old < commit_count$ **do**
6:         try `CAS` of *commit_seq*. Expect *commit_seq_old*, new value written is *commit_count*. Save value read to *commit_seq_old*.
7:     **end while**
8: **end if**

Algorithm 5.3 CommitSlot(*slot_size, slot_offset*)

or completely full sub-buffer.

As shown at the end of ReserveSlot(), switching between sub-buffers is done out-of-order. It consists of two phases: the first detects, within the CAS loop, if a buffer switch is needed. If this is the case, flags are set on the probe stack to make the out-of-order code, following the loop, increments the sub-buffer commit counts of the sub-buffer we are switching out from and the sub-buffer we are switching into. The sub-buffer switched out from will therefore have its *commit count* incremented by the missing amount of bytes between the number of bytes *reserved* (and thus monotonically incrementing) and the sub-buffer size. Switching to a new sub-buffer adds the new sub-buffer header's size to the new sub-buffer's commit count. Another case is also possible, namely when there is exactly enough event data to fit perfectly in the sub-buffer. In this case, an *end switch current* flag is raised so the header information is finalized. All these buffer switching cases also populate the sub-buffer headers with information regarding the current time stamp and padding size at the end of the sub-buffer, prior to incrementing the commit count. SwitchOldSubbuf(), SwitchNewSubbuf() and EndSwitchCurrent() are therefore responsible for incrementing the *commit count* of the amount of padding added at the end of a sub-buffer, clearing the reference flag when the sub-buffer is filled and updating *commit seq*.

Pushing a reader, represented by PushReader(), is done by a writer in *flight recorder* mode when it detects that the buffer is full. In that case, the writer sets the *read count* to the beginning of the following sub-suffer.

Flushing the buffers while tracing is active, as done by the pseudo-code presented in Algorithm ForceSwitch(), is required to permit streaming of information with a bounded latency, between the time events are written in the buffers and event delivery to user-space. It is a special-case of normal space reservation which does not reserve space in the sub-buffer, but forces a buffer switch if the current sub-buffer is non-empty. Buffer switch is called from a periodical timer, configurable by the user to select how often buffer data must be flushed.

Consumer

The consumer, `lttd`, uses two system calls, *poll()* and *ioctl()*, to control the interaction with the memory buffers, and *splice()* as a mean to extract the buffers to disk or to the network without extra copy. At kernel-level, we specialize those three sys-

Algorithm: FORCESWITCH()
Ensure: Buffer switch is done if sub-buffer contains data

```
1: repeat
2:     Calculate the commit_count needed to fill the current sub-buffer.
3: until CAS of write_count succeeds
4: PUSHREADER()
5: Set reference flag in pointer to current sub-buffer.  Indicates
   that the writer is using this sub-buffer.
6: SWITCHOLDSUBBUF()
7: SWITCHNEWSUBBUF()
```

Algorithm 5.4 FORCESWITCH()

tem calls for the virtual files presented by *DebugFS*. The daemon waits for incoming data using *poll()*. This system call waits to be woken up by the timer interrupt (see the ASYNCWAKEUPREADERSTIMER() pseudo-code in Algorithm 5.8). Once data is ready, it returns the poll priority to user-space. If the tracer is currently writing in the last available sub-buffer of the buffer, a high priority is returned. Pseudo-code READPOLL() summarizes the actions taken by the *poll()* system call.

Once control has returned to user-space from the *poll()* system call, the daemon takes a user-space mutex on the buffer and uses the *ioctl()* system call to perform buffer locking operations. Its implementation uses the READGETSUBBUF() and READPUTSUBBUF() algorithms. The former operation, detailed in Algorithm 5.6, reserves a sub-buffer for the reader and returns the *read count*. If the lower-level buffer writing scheme would allow concurrent accesses to the reserved sub-buffer between the reader and the writer, this value could be used to verify, in the READPUTSUBBUF() operation, detailed in Algorithm 5.7, that the reader has not been pushed by a writer dealing with buffers in *flight recorder* mode. However, as we present below, this precaution is unnecessary because the underlying buffer structure does not allow such concurrency.

The specialized *ioctl()* operation is responsible for synchronizing the reader with the writer's buffer-space reservation and commit. It is also responsible for making sure the sub-buffer is made private to the reader to eliminate any possible race in flight recorder mode. This is achieved by adding a supplementary sub-buffer, owned by the reader. A table with pointers to the sub-buffers being used by the writer

Algorithm: READPOLL()
Ensure: Returns buffer readability state and priority

1: Wait on *read_wait* wait queue.
2: **if** Eqn. 5.4 **then**
3:     **if** Sub-buffer is finalized (freed by the tracer) **then**
4:         Hang up.
5:         **return** `POLLHUP`
6:     **else**
7:         No information to read.
8:         **return** `OK`
9:     **end if**
10: **else**
11:     **if** Eqn. 5.1 **then**
12:         High-priority read.
13:         **return** `POLLPRI`
14:     **else**
15:         Normal read.
16:         **return** `POLLIN`
17:     **end if**
18: **end if**

Algorithm 5.5 READPOLL()

allows the reader to change the reference to each sub-buffer atomically. The READ-GETSUBBUF() algorithm is responsible for atomically exchanging the reference to the sub-buffer about to be read with the sub-buffer currently owned by the reader. If the `CAS` operation fails, the reader does not get access to the buffer for reading.

Given that sub-buffer management data structures are aligned on 4 or 8-bytes multiples, we can use the lowest bit of the sub-buffer pointer to encode whether it is actively referenced by the writer. This ensures that the pointer exchange performed by the reader can never succeed when the writer is actively using the reference to write to a sub-buffer about to be exchanged by the reader.

Asynchronous buffer delivery

Because the probe cannot interact directly with the rest of the kernel, it cannot call the scheduler to wake up the consumer. Instead, this ready to read sub-buffer delivery is done asynchronously by a timer interrupt. This interrupt checks if each buffer

Algorithm: ReadGetSubbuf()

Ensure: Take exclusive reader access to a sub-buffer.

1: Read *read_count*.
2: Read the *commit_seq* corresponding to the *read_count*.
3: Issue a *smp_mb()* (Memory Barrier on multiprocessor) to ensure *commit_seq* read is globally visible before sending the `IPI` (*Interprocessor Interrupt*).
4: Send `IPI` to target writer CPU (if differs from the local reader CPU) to issue a *smp_mb()*. This ensures that data written to the buffer and *write count* update are globally visible before the *commit seq* write. Wait for `IPI` completion.
5: Issue a *smp_mb()* to ensure the *reserve_count* and buffer data read are not re-ordered before `IPI` execution.
6: Read *reserve_count*.
7: **if** Negation of Eqn. 5.2 **then**
8:     **return** `EAGAIN`
9: **end if**
10: **if** Eqn. 5.4 (Only *flight recorder*) **then**
11:     **return** `EAGAIN`
12: **end if**
13: **if** Writer is holding a reference to the sub-buffer about to be exchanged $\vee$ Exchange of reader/writer sub-buffer reference fails **then**
14:     **return** `EAGAIN`
15: **end if**
16: **return** *read_count*

Algorithm 5.6 ReadGetSubbuf()

Algorithm: ReadPutSubbuf(*arg_read_count*)

Require: *read_count* returned by ReadGetSubbuf() (*arg_read_count*).

Ensure: Release exclusive reader access from a sub-buffer. Always succeeds even if the writer pushed the reader, because the reader had exclusive sub-buffer access.

1: $new_read_count = arg_read_count + subbuffer_size$.
2: `CAS` expects *arg_read_count*, replaces with *new_read_count*
3: **return** `OK`

Algorithm 5.7 ReadPutSubbuf(*arg_read_count*)

contains a filled sub-buffer and wakes up the readers waiting in the *read wait* queue associated with each buffer accordingly. This mechanism is detailed in Algorithm 5.8.

5.4.4 Memory Barriers

Although `LTTng` mostly keeps data local to each CPU, cross-CPU synchronization is still required at three sites:

– At initial time-stamp counters synchronization, done at boot-time by the operating system. This heavy synchronization, if not done by the `BIOS` (*Basic Input/Output System*), requires full control of the system.

– When the producer finishes writing to a sub-buffer, making it available for reading by a thread running on an arbitrary CPU. This involves using the proper memory barriers ensuring that all written data is committed to memory before another CPU starts reading the buffer.

– At consumed data counter update, involving the appropriate memory barriers ensuring the data has been fully read before making the buffer available for writing.

The two points at which a sub-buffer can pass from one CPU to another is when it is exchanged between the producer and the consumer and when it goes back from the consumer to the producer, because the consumer may run on a different CPU than the producer. Good care must therefore be taken to make sure correct memory ordering between buffer management variables and the buffer data writes. The condition which makes a sub-buffer ready for reading is represented by Eqn. 5.2, which depends on the *read count* and the *commit seq* counter corresponding to the *read count*. Therefore, before incrementing the sub-buffer *commit seq*, a write memory barrier must be issued on SMP systems allowing out-of-order memory writes to ensure the buffer data is written before the *commit seq* is updated. On the read-side, before reading the *commit seq*, a read memory barrier must be issued on SMP. It insures correct read ordering of counter and buffer data.

`LTTng` buffering uses an optimization over the classic memory barrier model. Instead of executing a write memory barrier before each *commit seq* update, a simple compiler optimization barrier is used to make sure data written to buffer and *commit seq* update happen in program order with respect to local interrupts. Given that the write order is only needed when the read-side code needs to check the buffer's *commit*

Algorithm: AsyncWakeupReadersTimer()
Ensure: Wake up readers for full sub-buffers

1: **for all** Buffers **do**
2: **if** Eqn. 5.2 **then**
3: Wake up consumers waiting on the buffer *read_wait* queue.
4: **end if**
5: **end for**

Algorithm 5.8 AsyncWakeupReadersTimer()

seq value, Algorithm 5.6 shows how the read-side sends an `IPI` to execute a memory barrier on the target CPU between two memory barriers on the local CPU to ensure that memory ordering is met when the sub-buffer is passed from the writer to the reader. This `IPI` scheme promotes the compiler barrier to a memory barrier each time the reader needs to issue a memory barrier. Given the reader needs to issue such a barrier only once per sub-buffer switch, compared to a write memory barrier once per event, this improves performance by removing a barrier from the fast path at the added cost of an extra `IPI` at each sub-buffer switch, which happen relatively rarely. With an average event size of 8 bytes and a typical sub-buffer size of 1 MiB, the ratio is one sub-buffer switch each 131072 events. Given an `IPI` executing a write memory barrier on an Intel Core2 Xeon 2.0 GHz takes about 2500 cycles and that a local write memory barrier takes 8 cycles, memory barrier synchronization speed is increased by a factor 419 to 1.

When the buffer is given back to the producer, a synchronized `CAS` is used to update the *read count*, which implies a full memory barrier before and after the instruction. The `CAS` ensures the buffer data is read before the *read count* is updated. Given that the writer does not have to read any data from the buffer and depends on reading the *read count* value to check if the buffer is full (in *non-overwrite* mode), only the *read count* is shared. The control dependency between the test performed on *read count* and write to the buffer ensures the writer never writes to the buffer before the reader has finished reading from it.

5.4.5 Buffer allocation

The lockless buffer management algorithm found in `LTTng` allows dealing with concurrent write accesses to segments of a circular buffer (slots) of variable length. This concurrency management algorithm does not impose any requirement on the nature of the memory backend which holds the buffers. The present section will expose the primary memory backends supported by `LTTng` as well as the backends planned for support in future versions.

The primary memory backend used by `LTTng` is a set of memory pages allocated by the operating system's page allocator. Those pages are not required to be physically contiguous. This ensures that page allocation is still possible even if memory is fragmented. There is no need to have any virtually contiguous address mapping, which is preferable given that there is a limited amount of kernel-addressable virtual address space (especially on 32-bits systems). These pages are accessed through a single-level page table which performs the translation from a linear address mapping (offset within the buffer) to a physical page address. Buffer *read()*, *write()* and *splice()* primitives abstract the non-contiguous nature of the underlying memory layout by providing an API which present the buffer as a virtually contiguous address space.

`LTTng` buffers are exported to user-space through the *DebugFS* file system. It presents the `LTTng` buffers as a set of virtual files to user applications and allows interacting with those files using *open()*, *close()*, *poll()*, *ioctl()* and *splice()* system calls.

`LTTng` includes a replacement of *RelayFS* aimed at efficient zero-copy data extraction from buffer to disk or to the network using the *splice()* system call. Earlier `LTTng` implementation, using *RelayFS*, were based on mapping the buffers into user-space memory to perform data extraction. However, this comes at the expense of wasting precious `TLB` entries usually available for other use. The current `LTTng` implementation uses the *splice()* system call. Its usage requires creating a pipe. A *splice()* system call, implemented specifically to read the buffer virtual files, is used to populate the pipe source with specific memory pages. In this case, the parts of the buffer to copy are selected. Then, a second *splice()* system call (the standard pipe implementation) is used to send the pages to the output file descriptor, which targets either a file on disk or a network socket.

Separating the buffer-space management algorithm from the memory backend support eases the implementation of specialized memory backends, depending on the

requirements:

– Discontiguous page allocation (presented above) requires adding a software single-level page table, but permits allocation of buffers at run-time when memory is fragmented.
– Early boot-time page allocation of large contiguous memory areas requires low memory fragmentation, but permits faster buffer page access because it does not need any software page-table indirection.
– Video memory backend can be used by reserving video memory for trace buffers. It allows trace data to survive hot reboots, which is useful to deal with kernel crash.

5.5 Experimental results

This section presents the experimental results from the design implementation under various workloads, and compares these with alternative existing technologies.

5.5.1 Methodology

To present the tracer performance characteristics, we first present the overhead of the `LTTng` tracer for various types of workloads on various types of systems. Then, we compare this overhead to existing state-of-the-art approaches.

The probe CPU-cycles benchmarks, presented in section 5.5.2, demonstrate the `LTTng` probe overhead in an ideal scenario, where the data and instructions are already in cache.

Then, benchmarks representing the real-life workloads tbench and dbench, simulate the load of a Samba server for network traffic and for disk traffic, respectively. A tbench test on loopback interface shows the worse-case scenario of 8 client and 8 server tbench threads heavily using a traced kernel. Scalability of the tracer when the number of cores increases is tested on the heavy loopback tbench workload.

Yet another set of benchmarks uses `lmbench` to individually test tracing overhead on various kernel primitives, mainly system calls and traps, to show the performance impact of active tracing on those important system components.

Finally, a set of benchmarks runs a compilation of the Linux kernel 2.6.30 with and without tracing to produce a CPU intensive workload.

Probe CPU-cycles overhead benchmarks are performed on a range of architectures. Unless specified, benchmarks are done on an Intel Core2 Xeon E5405 running at 2.0 GHz with 16 GiB of RAM. Tests are executed on a 2.6.30 Linux kernel with full kernel preemption enabled. The buffers configuration used for high event-rate buffers is typically two 1 MiB sub-buffers, except for block I/O events, where per-CPU buffers of eight 1 MiB sub-buffers are used.

5.5.2 Probe CPU-cycles overhead

This test measures the cycle overhead added by a `LTTng` probe. This provides us with a per-event overhead lower bound. This is considered a lower-bound because this test is performed in a tight loop, therefore favoring cache locality. In standard tracer execution, the kernel usually trashes part of the data and instruction caches between probe executions.

The number of cycles consumed by calling a probe from a static instrumentation site passing two arguments, a long and a pointer, on Intel Pentium 4, AMD Athlon, Intel Core2 Xeon and ARMv7 is presented in Table 5.1. These benchmarks are done in kernel-space, with interrupts disabled, sampling the CPU time-stamp counter before and after 20,000 loops of the tested case.

Given that one local `CAS` is needed to synchronize the tracing space reservation, based on the results published in [1], we can see that disabling interrupts instead of using the local `CAS` would add 34 cycles to these probes on Intel Core2, for an expected 14.3% slowdown. Therefore, not only is it interesting to use *local atomic operations* to protect against non-maskable interrupts, but it also improves the performance marginally. Changing the implementation to disable interrupts instead of using local `CAS` confirms this: probe execution overhead increases from 240 to 256 cycles, for a 6.6% slowdown.

Table 5.1 Cycles taken to execute a `LTTng` 0.140 probe, Linux 2.6.30

Architecture	Cycles	Core freq. (GHz)	Time (ns)
Intel Pentium 4	545	3.0	182
AMD Athlon64 X2	628	2.0	314
Intel Core2 Xeon	238	2.0	119
ARMv7 OMAP3	507	0.5	1014

5.5.3 tbench

The `tbench` benchmark tests the throughput achieved by the network traffic portion of a simulated Samba file server workload. Given it generates network traffic from data located in memory, it results in very low I/O and user-space CPU time consumption, and very heavy kernel network layer use. We therefore use this test to measure the overhead of tracing on network workloads. We compare network throughputs when running mainline Linux kernel, instrumented kernel and traced kernel.

This set of benchmarks, presented in Table 5.2, shows that tracing has very little impact on the overall performance under network load on a 100 Mb/s network card. 8 `tbench` client threads are executed for a 120s warm up and 600s test execution. Trace data generated in flight recorder mode reaches 0.9 GiB for a 1.33 MiB/s trace data throughput. Data gathered in normal tracing to disk reaches 1.1 GiB. The supplementary data generated when writing trace-data to disk is explained by the fact that we also trace disk activity, which generates additional events. This very little performance impact can be explained by the fact that the system was mostly idle.

Now, given that currently existing 1 Gb/s and 10 Gb/s network cards can generate higher throughput, and given the 100 Mb/s link was the bottleneck of the previous `tbench` test, Table 5.3 shows the added tracer overhead when tracing `tbench` running with both server and client on the loopback interface on the same machine, which is a worse-case scenario in terms of generated throughput kernel-wise. This workload consists in running 8 client threads and 8 server threads.

The kernel instrumentation, when compiled-in but not enabled, actually accelerates the kernel. This can be attributed to modification of instruction and data cache layout. Flight recorder tracing stores 92 GiB of trace data to memory, which repre-

Table 5.2 `tbench` client network throughput tracing overhead

Test	Tbench Throughput (MiB/s)	Overhead (%)	Trace Throughput ($*10^3$ events/s)
Mainline Linux kernel	12.45	0	–
Dormant instrumentation	12.56	0	–
Overwrite (flight recorder)	12.49	0	104
Normal tracing to disk	12.44	0	107

Table 5.3 `tbench` localhost client/server throughput tracing overhead

Test	Tbench Throughput (MiB/s)	Overhead (%)	Trace Throughput ($*10^3$ events/s)
Mainline Linux kernel	2036.4	0	–
Dormant instrumentation	2047.1	-1	–
Overwrite (flight recorder)	1474.0	28	9768
Normal tracing to disk	–	–	–

sents a trace throughput of 130.9 MiB/s for the overall 8 cores. Tracing adds a 28% overhead on this workload. Needless to say that trying to export such throughput to disk would cause a significant proportion of events to be dropped. This is why tracing to disk is excluded from this table.

5.5.4 Scalability

To characterize the tracer overhead when the number of CPUs increases, we need to study a scalable workload where tracing overhead is significant. The localhost `tbench` test exhibits these characteristics. Figure 5.7 presents the impact of *flight recorder* tracing on the `tbench` localhost workload on the same setup used for Table 5.3. The number of active processors varies from 1 to 8 together with the number of `tbench` threads. We notice that the `tbench` workload itself scales linearly in the absence of tracing. When tracing is added, linear scalability is invariant. It shows that the overhead progresses linearly as the number of processors increases. Therefore, tracing with `LTTng` adds a constant per-processor overhead independent from the number of processors in the system.

5.5.5 dbench

The `dbench` test simulates the disk I/O portion of a Samba file server. The goal of this benchmark is to show the tracer impact on such a workload, especially for *non-overwrite* tracing to disk.

This set of benchmarks, presented in Table 5.4, shows tracing overhead on a 8 thread `dbench` workload. Tracing in *flight recorder* mode causes a 3% slowdown on disk throughput while generating 30.2 GiB of trace data into memory buffers. Normal tracing to disk causes a 35% slowdown on heavy disk operations, but lack of

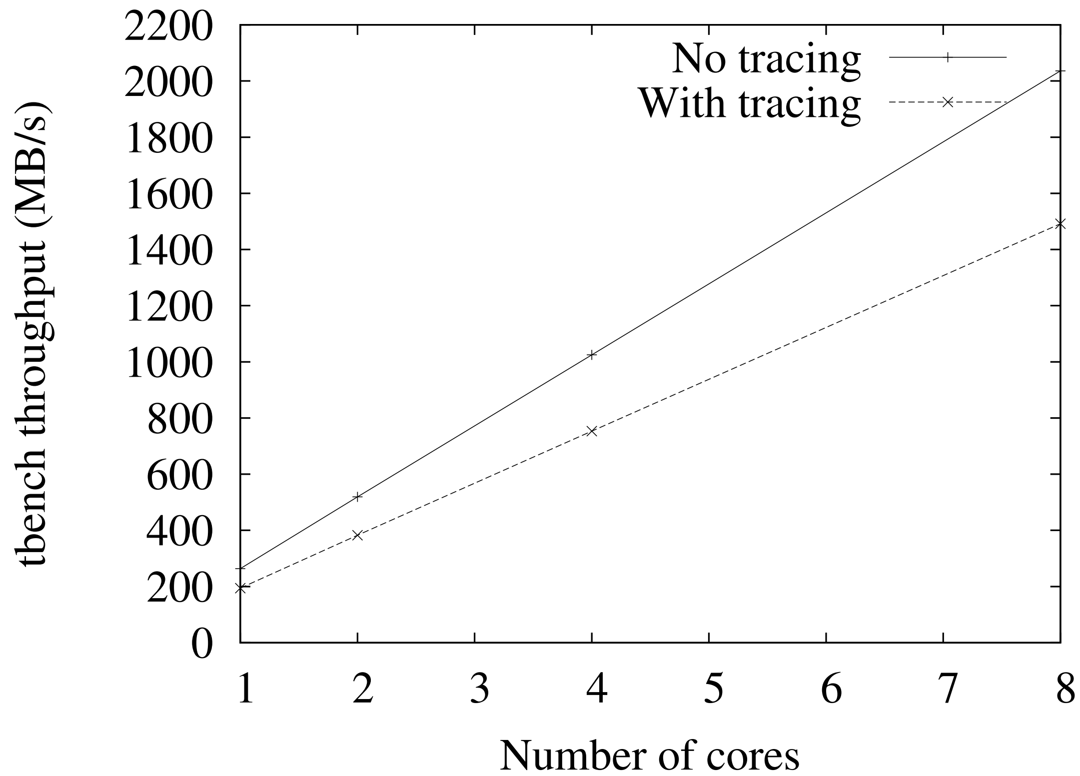

Figure 5.7 Impact of tracing overhead on localhost `tbench` workload scalability

disk bandwidth is causing a significant portion of trace events to be discarded.

Analysis of the buffer state in *flight recorder* mode shows that 30.2 GiB worth of data was generated in 720 seconds, for a sustained trace throughput of 43.0 MiB/s. In *non-overwrite* mode, the trace is written to the same disk `dbench` is using. The tracing throughput is therefore significant compared to the available disk bandwidth.

Table 5.4 `dbench` disk write throughput tracing overhead

Test	Dbench Throughput (MiB/s)	Overhead (%)	Trace Throughput ($*10^3$ events/s)
Mainline Linux kernel	1334.2	0	–
Dormant instrumentation	1373.2	-2	–
Overwrite (flight recorder)	1297.0	3	2840
Non-overwrite tracing to disk	872.0	35	2562

It comes without surprise that only 23 GiB of trace data has been collected to disk in the *non-overwrite* trace, with a total of 21.8 million events lost. This trace size difference is caused both by the events lost (only lost about 244 MiB of data given an average event size of 12 bytes) and, mostly, to the behavior change generated by the added disk I/O activity for tracing. While the system is busy writing large chunks of trace data, it is not available to process smaller and more frequent `dbench` requests. This nicely shows how the tracer, in *non-overwrite* mode, can affect disk throughput in I/O-heavy workloads.

5.5.6 lmbench

The `lmbench` test benchmarks various kernel primitives by executing them in loops. We use this test to appropriately test the tracer overhead on a per-primitive basis. Running `lmbench` on the mainline Linux kernel, *flight recorder* and *non-overwrite* tracing kernels, helps understanding the performance deterioration caused by tracing.

When running on a Intel Core2 Xeon E5405, the standard `lmbench` 3.0 OS test generates 5.41 GiB of trace data with the default `LTTng` instrumentation in 6 minutes for a throughput of 150 MiB/s. When writing to disk the total trace size reaches 5.5 GiB due to the added traced disk I/O overhead.

The "simple system call" test, which calls a system call with small execution time in a tight loop, takes 0.1752 μs on the mainline Linux kernel. Compared to this, it takes 0.6057 μs on the *flight recorder* mode traced kernel. In fact, the benchmarks for *flight recorder* tracing and disk tracing are very similar, because the only difference is the CPU time taken by the `lttd` daemon and the added disk I/O.

The "simple system call" slowdown is explained by the fact that two sites are instrumented: system call entry and system call exit. Based on measurements from Table 5.1, we would expect each event to add at least 0.119 μs to the system call. In reality, they add 0.215 μs each to the system call execution. The reasons for this additional slowdown is because supplementary registers must be saved in the system call entry and exit paths and cache effects. The register overhead is the same as the well-known *ptrace()* debugger interface, secure computing and process accounting because these and `LTTng` all share a common infrastructure to extract these registers.

Some system calls have more specific instrumentation in their execution path. For instance, the file name is extracted from the *open()* system call, the file descriptor

and size are extracted from the *read()* system call. The performance degradation is directly related to the number of probes executed. For the *read()* system call, the mainline Linux kernel takes 0.2138 μs, when the flight recorder tracing kernel takes 0.8043 μs. By removing the "Simple system call" tracing overhead, this leaves a 0.1600 μs, which corresponds to the added event in the *read()* system call.

The page fault handler, a frequently executed kernel code path, is instrumented with two tracepoints. It is very important due to the frequency at which it is called during standard operation. On workloads involving many short-lived processes, page faults, caused by copy-on-write, account for an important fraction of execution time (4% of a Linux kernel build). It runs in 1.3512 μs on the mainline Linux kernel and takes 1.6433 μs with flight recorder activated. This includes 0.146 μs for each instrumentation site, which is close to the expected 0.119 μs per event. Non-cached memory accesses and branch prediction buffer pollution are possible causes for such small execution time variation from expected results.

Instrumentation of such frequently executed kernel code path is the reason why minimizing probe execution time is critical to the tracer's usability on heavy workloads.

Other `lmbench` results show that some instrumented code paths suffer from greater overhead. This is mostly due to the use of a less efficient dynamic format-string parsing method to write the events into the trace buffers. For instance, the "Process fork+exit" test takes 211.5 μs to execute with tracing instead of 177.8 μs, for an added overhead of 33.7 μs for each entry/exit pair. Based on execution trace analysis of standard workloads, as of `LTTng` 0.140, events corresponding to process creation and destruction where not considered to be frequently used compared to page faults, system calls, interrupts and scheduler activity. If this becomes a concern, the optimized statically-compiled version of the event serializer could be used.

5.5.7 gcc

The `gcc` compilation test aims at showing the tracer impact on a workload where most of the CPU time is spent in user-space, but where many short-lived processes are created. Building the Linux kernel tree is such a scenario, where the `make` creates one short-lived `gcc` instance per file to compile. This therefore shows mostly tracer impact on process creation. This includes page fault handler instrumentation impact, due to

copy-on-write and lazy page population mechanisms when processes are created and when executables are loaded. This also includes instrumentation of scheduler activity and process state changes.

Table 5.5 presents the time taken to build the Linux kernel with `gcc`. This test is performed after a prior cache-priming compilation. Therefore, all the kernel sources are located in cache.

Tracing the kernel in *flight recorder* mode, with the default `LTTng` instrumentation, while compiling the Linux kernel, generates 1.1 GiB of trace data for a 3% slowdown. The results show, without surprise, that kernel tracing has a lower impact on user-space CPU-bound workloads than I/O-bound workloads. Tracing to disk generates 1.3 GiB of data output. This is higher than the trace data generated for flight recording due to the supplementary disk activity traced. Trace throughput, when tracing to disk, is lower than *flight recorder* mode, because the tracer disk activity generates fewer events per second than kernel compiling in the CPU time it consumes, hence reducing the number of events per second to record.

5.5.8 Comparison

Previous work on highly scalable operating systems has been done at IBM Research resulting in the `K42` operating system [38], which includes a built-in highly scalable kernel tracer based on a lockless buffering scheme. As presented in Section 5.2, `K42`'s buffering algorithm contains rare race conditions which could be problematic especially given `LTTng` buffer and event size flexibility. Being a research operating system, `K42` does not support CPU hotplug, nor distributing tracing overhead across idle cores, and is limited to a subset of existing widely used hardware, which provides a 64-bits cycle counter synchronized across cores.

Table 5.5 Linux kernel compilation tracing overhead

Test	Time (s)	Overhead (%)	Trace Throughput ($*10^3$ events/s)
Mainline Linux kernel	85	0	–
Dormant instrumentation	84	-1	–
Overwrite (flight recorder)	87	3	822
Normal tracing to disk	90	6	816

The instrumentation used in `LTTng` has been taken from the original LTT project [19]. It consists of about 150 instrumentation sites, some architecture-agnostic, others being architecture-specific. They have been ported to the "Linux Kernel Markers" [52] and then to "Tracepoints" [51] developed as part of the `LTTng` project and currently integrated in the mainline Linux kernel. The original LTT and earlier `LTTng` versions, used `RelayFS` [53] to provide memory buffer allocation and mapping to user-space. `LTTng` re-uses part of the *splice()* implementation found in `RelayFS`.

To justify the choice of using static code-level instrumentation instead of dynamic, breakpoint-based instrumentation, we must explain the performance impact of breakpoints. These are implemented with a software interrupt triggered by a breakpoint instruction temporarily replacing the original instructions to instrument. The specialized interrupt handler executes the debugger or the tracer when the breakpoint instruction is executed. An interesting result of the work presented in this paper is that the `LTTng` probe takes less time to run than a breakpoint alone. Tests running an empty Kprobe, which includes a breakpoint and single-stepping, in a loop shows it has a performance impact of 4200 cycles, or 1.413 μs, on a 3 GHz Pentium 4. Compared to this, the overall time taken to execute an `LTTng` probe is 0.182 μs, which represents a 7.8:1 acceleration compared to the breakpoint alone.

It is also important to compare the lockless scheme proposed to an equivalent solution based on interrupt disabling. We therefore created an alternative implementation of the `LTTng` buffering scheme based on interrupt disabling for this purpose. It uses non-atomic operations to access the buffer state variables and is therefore not `NMI`-safe. Table 5.6 shows that the lockless solution is either marginally faster (7–8%) on architectures where interrupt disabling cost is low, or much faster (34%) in cases where interrupt disabling is expensive in terms of cycles per instruction.

Table 5.6 Comparison of lockless and interrupt disabling `LTTng` probe execution time overhead, Linux 2.6.30

Architecture	IRQ-off (ns)	Lockless (ns)	Speedup (%)
Intel Pentium 4	212	182	14
AMD Athlon64 X2	381	314	34
Intel Core2 Xeon	128	119	7
ARMv7 OMAP3	1108	1014	8

Benchmarks performed on DTrace [33], the Solaris tracer, on a Intel Pentium 4 shows a performance impact of 1.18 μs per event when tracing all system calls to a buffer. LTTng takes 0.182 μs per event on the same architecture, for a speedup of 6.42:1. As shown in this paper, tracing a tbench workload with LTTng generates a trace throughput of 130.9 MiB/s, for approximately 8 million events/s with an average event size of 16 bytes. With this workload, LTTng has a performance impact of 28%, for a workload execution time of 1.28:1. DTrace being 6.42 times slower than LTTng, the same workload should be expected to be slowed down by 180% and therefore have an execution time of 2.8:1. Therefore, performance-wise, LTTng has nothing to envy [34]. This means LTTng can be used to trace workloads and diagnose problems outside of DTrace reach.

5.6 Conclusion

Overall, the LTTng kernel tracer presented in this paper presents a wide kernel code instrumentation coverage, which includes tricky non-maskable interrupts, traps and exception handlers, as well as the scheduler code. It has a per-event performance overhead 6.42 times lower than the existing DTrace tracer. The performance improvements are mostly derived from the following atomic primitive characteristics: *local atomic operations*, when used on local per-CPU variables, are cheaper than disabling interrupts on many architectures.

The atomic buffering mechanism presented in this paper is very useful for tracing. The good reentrancy and performance characteristics it demonstrates could be useful to other parts of the kernel, especially drivers. Using this scheme could accelerate buffer synchronization significantly and diminish interrupt latency.

A port of LTTng has already been done to the Xen hypervisor and as a user-space library as proofs of concept to permit studying merged traces taken from the hypervisor, the various kernels running in virtual machines, and user-space applications and libraries. Future work includes polishing these ports and integrating them to Xen. Work on modeling and formal verification by model-checking is currently ongoing.

Acknowledgements

We would like to thank the Linux Trace Toolkit, Linux and SystemTAP communities for their feedback, as well as NSERC, Google, IBM Research, Autodesk and Ericsson for funding parts of this work. We are indebted to Etienne Bergeron and Robert Wisniewski for reviewing this paper.

Chapter 6

Paper 3: User-Level Implementations of Read-Copy Update

Abstract

Read-copy update (RCU) is a synchronization primitive that is often used as a replacement for reader-writer locking, due to the fact that it provides extremely lightweight read-side primitives with sharply bounded execution times. RCU updates are typically much heavier weight than are RCU readers, especially when used in conjunction with locking.

Although RCU is heavily used in a number of kernel-level environments, these implementations make use of interrupt- and preemption-disabling facilities that are often unavailable to user-level applications. The few RCU implementations that are available to user applications either provide inefficient read-side primitives or restrict the application architecture.

This paper describes several classes of efficient RCU implementations that are based on primitives commonly available to user-level applications.

Finally, performance comparison of these RCU primitives with each other and to standard locking leads to a discussion on appropriate locking mechanisms for various workloads. This opens the door to use of RCU outside of kernels.

6.1 Introduction

Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002. RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast to conventional locking prim-

itives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete. RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring reclamation of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases, as will be presented in Section 6.4.2, RCU's read-side primitives have zero overhead.

Although mechanisms similar to RCU have been used in a number of operating-system kernels [54, 55, 56, 57, 58], and, as shown in Figure 6.1, is heavily used in the Linux kernel, we are not aware of significant application usage. This lack of application-level use is due in part to the fact that prior user-level RCU implementations imposed global constraints on the application's structure and operation [59], and in some cases heavy read-side overhead as well [60]. The popularity of RCU in operating-system kernels has been in part due to the fact that these can accommodate the required global constraints imposed by earlier RCU implementations. Kernels therefore permit use of the high-performance quiescent-state based reclamation (QSBR) class of RCU implementations. In fact, in server-class (CONFIG_PREEMPT=n) Linux-kernel builds, RCU incurs zero read-side overhead [61].

Whereas we cannot yet put forward a single user-level RCU implementation that is ideal for all user-level environments, the three classes of RCU implementations described in this paper should suffice for most applications.

First, Section 6.2 provides a brief overview of RCU, including RCU semantics. Then, Section 6.3 describes user-level scenarios that could benefit from RCU. This is followed by the presentation of three classes of RCU implementation in Section 6.4. Finally, Section 6.5 presents experimental results, comparing RCU solutions to each other and to standard locks. This leads to recommendations on locking use for various workloads presented in Section 6.6.

6.2 Brief Overview of RCU

This section introduces a conceptual view covering most RCU-based algorithms in Section 6.2.1 to familiarise the reader with RCU concepts and vocabulary. It then presents an informal RCU desiderata in Section 6.2.2, which details the goals pursued in this work. Then, Section 6.2.3 shows how RCU is used to delete an element from a linked list in the face of concurrent readers. Finally, Section 6.2.4 gives an overview of RCU semantics, presenting the synchronization guarantees provided by RCU.

6.2.1 Conceptual View of RCU Algorithms

A schematic for the high-level structure of an RCU-based algorithm is shown in Figure 6.2, which can be thought of as a pictorial view of Equation 6.1 presented in

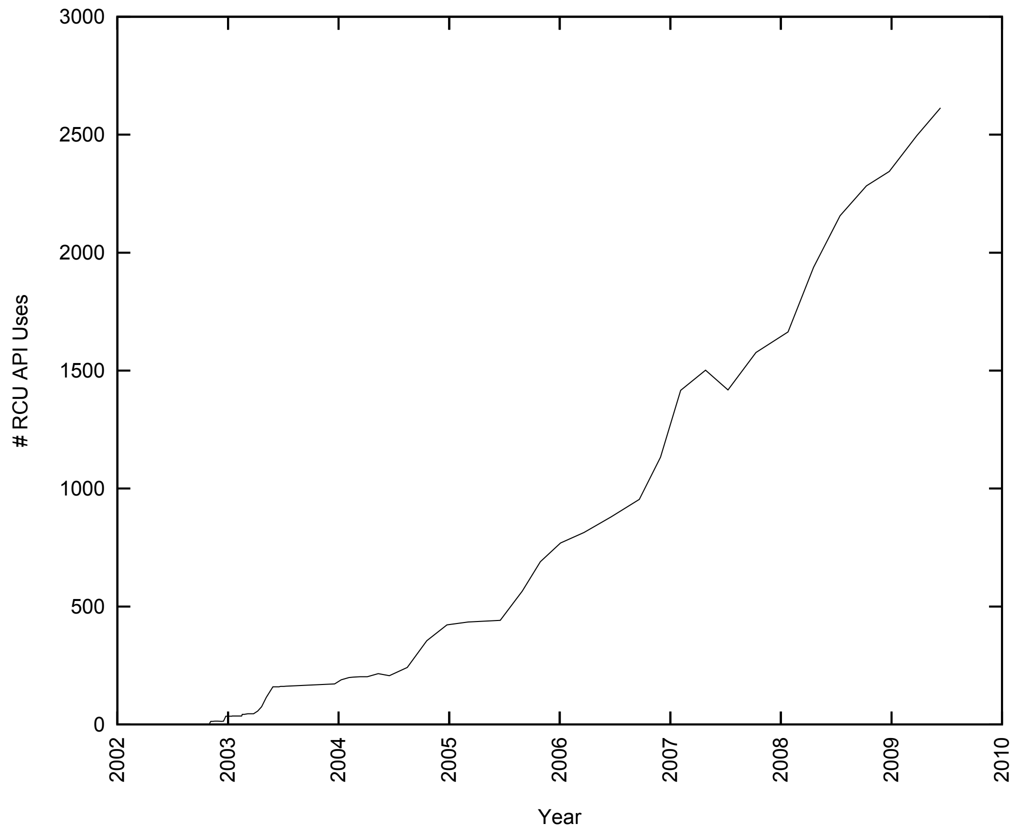

Figure 6.1 Linux-kernel usage of RCU

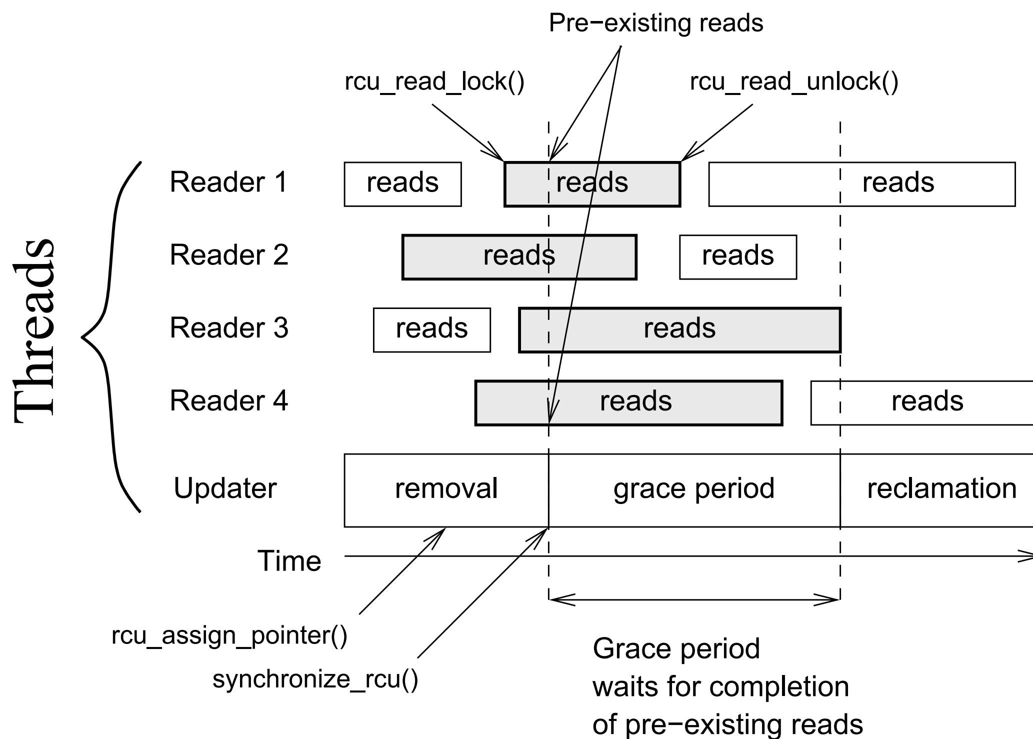

Figure 6.2 Schematic of RCU grace period and read-side critical sections

Section 6.2.4. The grace period concept, explained thoroughly in section 6.2.4, can be defined informally for the needs of this section as a period of time such that all `RCU` read-side critical sections in existence at the beginning of a given grace period have completed before its end.

Here, each box labeled "Reads" is an `RCU` read-side critical section that begins with `rcu_read_lock()` and ends with `rcu_read_unlock()`. Each row of `RCU` read-side critical sections denotes a separate thread, for a total of four read-side threads. The two boxes at the bottom left and right of the figure denote a fifth thread, this one performing an `RCU` update.

This `RCU` update is split into two phases, a removal phase denoted by the lower left-hand box and a reclamation phase denoted by the lower right-hand box. These two phases must be separated by a grace period, which is determined by the duration of the `synchronize_rcu()` execution. During the removal phase, the `RCU` update removes elements from the data structure (possibly inserting some as well) by issuing an `rcu_assign_pointer()` or equivalent pointer-replacement primitive. These removed

data elements will not be accessible to RCU read-side critical sections starting after the removal phase ends, but might still be accessed by RCU read-side critical sections initiated during the removal phase. However, by the end of the RCU grace period, all of the RCU read-side critical sections that might be accessing the newly removed data elements are guaranteed to have completed, courtesy of the definition of "grace period". Therefore, the reclamation phase beginning after the grace period ends can safely free the data elements removed previously.

6.2.2 User-Space RCU Desiderata

Extensive use of RCU applications has lead to the following user-space RCU desiderata:

1. Read-side primitives (such as `rcu_read_lock()` and `rcu_read_unlock()`) bounding RCU read-side critical sections and grace-period primitives (such as `synchronize_rcu()` and `call_rcu()`) must have the property that any RCU read-side critical section in existence at the start of a grace period completes by the end of the grace period.
2. RCU read-side primitives should avoid expensive operations such as cache misses, atomic instructions, memory barriers, and conditional branches.
3. RCU read-side primitives should have $O(1)$ computational complexity to enable real-time use. This property guarantees freedom from deadlock.
4. RCU read-side primitives should be usable in all contexts, including nested within other RCU read-side critical sections. Another important special context is library functions having incomplete knowledge of the user application.
5. RCU read-side primitives should be unconditional, thus eliminating the failure checking that would otherwise complicate testing and validation. This property has the nice side-effect of avoiding livelocks.
6. RCU read-side should not cause write-side starvation: grace periods should always complete, even given a steady flow of time-bounded read-side critical sections.
7. Any operation other than a quiescent state (and thus a grace period) should be permitted within an RCU read-side critical section. In particular, non-idempotent operations such as I/O and lock acquisition/release should be permitted.

8. It is permissible to mutate an `RCU`-protected data structure while executing within an `RCU` read-side critical section. Of course, any grace periods following this mutation must occur after the `RCU` read-side critical section completes.

9. `RCU` primitives should be independent of memory allocator design and implementation, so that `RCU` data structures may be protected regardless of how their data elements are allocated and freed.

10. `RCU` grace periods should not be blocked by threads that halt outside of `RCU` read-side critical sections. (But note that most quiescent-state-based implementations violate this desideratum.)

The `RCU` implementations described in Section 6.4 are designed to meet the above list of desiderata.

6.2.3 RCU Deletion From a Linked List

`RCU`-protected data structures in the Linux kernel include linked lists, hash tables, radix trees, and a number of custom-built data structures. Figure 6.3 shows how `RCU` may be used to delete an element from a linked list that is concurrently being traversed by `RCU` readers, as long as each reader conducts its traversal within the confines of a single `RCU` read-side critical section. The first column of the figure presents the data structure view of the updater thread. The second column presents the data structure view of a reader thread starting before the grace period begins. The third column presents a reader thread starting after the beginning of the grace period.

The first row of the figure shows a list with elements `A`, `B`, and `C`, to each of which every `RCU` readers initiated before the beginning of the grace period might both acquire and hold references.

The `list_del_rcu()` primitive unlinks element `B` from the list, but leaves the link from `B` to `C` intact, as shown on the second row of the figure. This permits any `RCU` readers currently referencing `B` to advance to `C`, as shown on the second and third rows of the figure. The transition between the second and third rows shows the reader thread data structure view gradually seeing element `B` disappear. During this transition, some readers will see element `B` and others will not. Although there might be `RCU` readers still referencing Element `B`, new `RCU` readers can no longer acquire a reference to it.

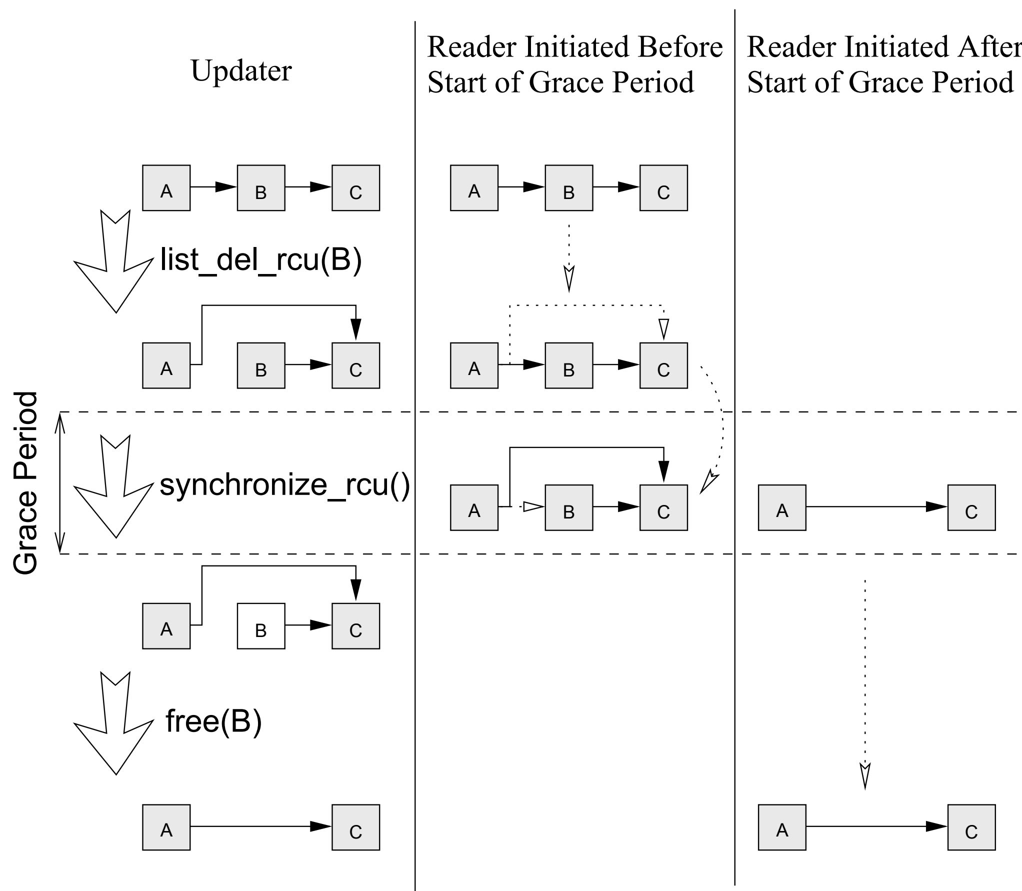

Figure 6.3 RCU linked-list deletion

The `synchronize_rcu()` primitive waits for one grace period, after which all pre-existing `RCU` read-side critical sections will have completed, resulting in the state shown in the fourth row of the figure. This state is the same as the second and third rows, except for the fact that there can no longer be any `RCU` readers holding references to Element `B`. This change of state of `B` from *globally visible* to *private* is depicted by using a white background for the `B` box. At this point, it is safe to invoke `free()`, reclaiming the memory consumed by element `B`, as shown on the last row of the figure.

Of course, the deletion process must be protected by some mutual-exclusion mechanism to ensure concurrent deletion of two contiguous list items do not corrupt the list. One common strategy to perform this mutual exclusion is to use locking.

Although `RCU` is used in a wide variety of ways, this list-deletion process is the

most common usage.

6.2.4 Overview of RCU Semantics

RCU semantics comprise the grace-period guarantee and the publication guarantee. Synchronization guarantees among concurrent modifications of the RCU-protected data structure must be provided by some other mechanism. In the Linux kernel, this other mechanism is typically locking, but any other suitable mechanism may be used, including atomic operations, non-blocking synchronization, transactional memory, or a single designated updater thread.

Grace-Period Guarantee

RCU operates by defining *RCU read-side critical sections*, delimited by `rcu_read_lock()` and `rcu_read_unlock()`, and by defining *grace periods*, which are periods of time such that all RCU read-side critical sections in existence at the beginning of a given grace period have completed before its end. The RCU primitive `synchronize_rcu()` starts a grace period and then waits for it to complete. Most RCU implementations allow RCU read-side critical sections to be nested.

Somewhat more formally, suppose we have a group of C-language statements S_i within an RCU read-side critical section as follows:

`rcu_read_lock()`; S_0; S_1; S_2; ...; `rcu_read_unlock()`;

Suppose further that we have a group of C-language mutation statements M_i and a group of C-language destruction statements D_i separated by an RCU grace period:

M_0; M_1; M_2; ...; `synchronize_rcu()`; D_0; D_1; D_2; ...;

Then the following holds, where "$\rightarrow$" indicates that the statement on the left executes prior to that on the right, and where "$\Longrightarrow$" denotes logical implication:

$$\exists S_a, M_b (S_a \rightarrow M_b) \Longrightarrow \forall S_i, D_j (S_i \rightarrow D_j) \tag{6.1}$$

In other words, if any statement in a given RCU read-side critical section executes prior to any statement preceding a given grace period, then all statements in that RCU read-side critical section must execute prior to any statement following that same grace period.

This guarantee permits `RCU`-based algorithms to trivially avoid a number of difficult race conditions that can otherwise result in poor performance, limited scalability, and great complexity. However this guarantee is insufficient, as it does not show that readers can operate consistently while an update is in progress. This case is covered by the guarantee presented in the next section.

Publication Guarantee

It is important to note that the statements S_a and M_b may execute concurrently, even in the case where S_a is referencing the same data element that M_b is concurrently modifying. The publication guarantee associated with the `rcu_assign_pointer()` and `rcu_dereference()` primitives allow this concurrency to be handled both correctly and easily: any dereference of a pointer returned by `rcu_dereference()` is guaranteed to see any changes prior to the corresponding `rcu_assign_pointer()`, including any changes prior to any earlier `rcu_assign_pointer()` involving that same pointer.

Somewhat more formally, suppose that the `rcu_assign_pointer()` is used as follows:

I_0; I_1; I_2; ...; `rcu_assign_pointer(g, p)`;

where each I_i is a C-language statement that initializes a field in the structure referenced by the local pointer `p`, and where the global pointer `g` is visible to reading threads.

Then the body of a canonical `RCU` read-side critical section would appear as follows:

`q` = `rcu_dereference(g)`; R_0; R_1; R_2; ...;

where this `RCU` read-side critical section is enclosed in `rcu_read_lock()` and `rcu_read_unlock()`, `q` is a local pointer, `g` is the same global pointer updated by the earlier `rcu_assign_pointer()` (and possibly updated again by some later invocations of `rcu_assign_pointer()`), and each R_i dereferences `q` to access one of the fields initialized by one of the statements I_i.

Then we have the following, where A is the `rcu_assign_pointer()` and D is the `rcu_dereference()`:

$$A \rightarrow D \Longrightarrow \forall I_i, R_j (I_i \rightarrow R_j) \tag{6.2}$$

In other words, if a given `rcu_dereference()` statement accesses the value stored by a given `rcu_assign_pointer()`, then all statements dereferencing the pointer returned by that `rcu_dereference()` must see the effects of any initialization statements preceding the `rcu_assign_pointer()`.

This guarantee allows new data to be initialized and added to an RCU-protected data structure in face of concurrent RCU readers.

Given both the grace-period and publication guarantees, these five primitives enable a wide variety of algorithms and data structures providing extremely low read-side overheads for read-mostly data structures [61, 59, 62, 63]. Again, note that concurrent updates must be handled by some synchronization mechanism, be it locking, atomic operations, non-blocking synchronization, transactional memory, or a single updater thread.

With this background on RCU, we are ready to consider how it might be used in user-level applications.

6.3 User-Space RCU Usage Scenarios

The past year has seen increased interest in applying RCU to user-space applications.

User-level RCU was needed for a user-level infrastructure that provides low-overhead tracing for user-mode applications. RCU is used for tracer control data synchronization in the LTTng tracer implementation [1], which is being ported to a user-space library. This usage scenario poses important constraints on the RCU requirements. This tracing library cannot be too intrusive in terms of program modification, which makes the QSBR approach presented in Section 6.4.2 inappropriate for such usage scenario. It also needs to support extensible instrumentation of user-selected execution sites, including signal handlers, which therefore requires supporting nested RCU critical sections and RCU reader critical sections in signal handlers. This usage scenario is also very performance demanding on workloads involving instrumentation of frequent execution sites. Therefore, having a low-overhead and scalable read-side is very important. Therefore, an ideal locking primitive for a tracing library would require no knowledge of the application and could be used to protect data structures used in a library.

User-level RCU has also been proposed for an elliptics-network distributed cloud-

based storage project [64]. BIND, a major domain name server at the root of Internet domain name resolution, is facing multi-threading scalability issues that are currently addressed with reader-writer locks [65]. Given the fact that domain names are read often but rarely updated, these could benefit from major performance improvement by using user-level RCU. Others have mentioned possibilities in financial applications. One can also argue that RCU has seen long use at user level in the guise of user-mode Linux.

In general, the area of applicability of RCU to user-mode applications appears similar to that in the Linux kernel: to read-mostly data structures, especially in cases where stale data can be accommodated.

6.4 Classes of RCU Implementations

This section describes several classes of RCU implementations, with Sections 6.4.2, 6.4.3, and 6.4.4 presenting user-space RCU implementations that are optimized for different usage by user-space applications, but first Section 6.4.1 describes some primitives that might be unfamiliar to the reader. The implementation presented in Section 6.4.2 offers the best possible read-side performance, but requires that each of the application's threads periodically pass through a quiescent state, thus strongly constraining the application's design. The implementation presented in Section 6.4.3 places almost no constraints on the application's design, thus being appropriate for use within a general-purpose library, but having higher read-side overhead. Section 6.4.4 presents an implementation having low read-side overhead, and requiring only that the application give up one signal to RCU processing. Finally, Section 6.4.5 demonstrates how to create wait-free RCU update primitives.

6.4.1 Notation

The examples in this section use a number of primitives that may be unfamiliar, and are thus listed in this section.

Per-thread variables are defined via `DEFINE_PER_THREAD()`. A thread may access its own instance of a per-thread variable using `__get_thread_var()`, or some other thread's instance via `per_thread()`. The `for_each_thread()` primitive sequences through all threads, one at a time.

The `pthread_mutex` is a type defined by the pthread library for mutual exclusion variables. The `mutex_lock()` primitive acquires a `pthread_mutex` instance, and `mutex_unlock()` releases it. The `mb` keyword stands for "memory barrier". The `smp_mb()` primitive emits a full memory barrier, for example, the `sync` instruction on the PowerPC architecture. The `smp_wmb()` and `smp_rmb()` primitives are, respectively, store and load memory barriers, corresponding, for example, to the `sfence` and `lfence` instructions on the x86 architecture. The `ACCESS_ONCE()` primitive prohibits any compiler optimization that might otherwise turn a single fetch or store into multiple fetches, as might happen under heavy register pressure. The `barrier()` primitive prohibits any compiler code-motion optimization that might otherwise move fetches or stores across the `barrier()` primitive.

```
 1 long rcu_gp_ctr = 0;
 2 DEFINE_PER_THREAD(long, rcu_reader_qs_gp);
 3
 4 static inline void rcu_read_lock(void)
 5 {
 6 }
 7
 8 static inline void rcu_read_unlock(void)
 9 {
10 }
11
12 static inline void rcu_quiescent_state(void)
13 {
14   smp_mb();
15   __get_thread_var(rcu_reader_qs_gp) =
16     ACCESS_ONCE(rcu_gp_ctr) + 1;
17   smp_mb();
18 }
19
20 static inline void rcu_thread_offline(void)
21 {
22   smp_mb();
23   __get_thread_var(rcu_reader_qs_gp) =
24     ACCESS_ONCE(rcu_gp_ctr);
25 }
26
27 static inline void rcu_thread_online(void)
28 {
29   __get_thread_var(rcu_reader_qs_gp) =
30     ACCESS_ONCE(rcu_gp_ctr) + 1;
31   smp_mb();
32 }
```

Figure 6.4 RCU read side using quiescent states

6.4.2 Quiescent-State-Based Reclamation RCU

The QSBR RCU implementation provides near zero-overhead read-side, but requires modifying the application, as this section explains.

Figure 6.4 shows the read-side primitives used to construct a user-level quiescent-state-based reclamation (QSBR) implementation of RCU based on quiescent states. As can be seen from lines 4–10 in the figure, the `rcu_read_lock()` and `rcu_read_unlock()` primitives do nothing, and can in fact be expected to be inlined and optimized away, as they are in server builds of the Linux kernel. This is due to the fact that quiescent-state-based RCU implementations *approximate* the extents of RCU read-side critical sections using the aforementioned quiescent states, which contain calls to `rcu_quiescent_state()`, shown from lines 12–18 in the figure. Threads entering extended quiescent states (for example, when blocking) may instead use the `thread_offline()` and `thread_online()` APIs to mark the beginning and the end, respectively, of such an extended quiescent state. As such, `thread_online()` is analogous to `rcu_read_lock()` and `thread_offline()` is analogous to `rcu_read_unlock()`. These two functions are shown on lines 20–32 in the figure. In either case, it is invalid for a quiescent state to appear within an RCU read-side critical section.

In `rcu_quiescent_state()`, line 14 executes a memory barrier to prevent any code prior to the quiescent state from being reordered into the quiescent state. Lines 15–16 pick up a copy of the global `rcu_gp_ctr` (RCU grace-period counter), using `ACCESS_ONCE()` to ensure that the compiler does not employ any optimizations that would result in `rcu_gp_ctr` being fetched more than once, and then adds one to the value fetched and stores it into the per-thread `rcu_reader_qs_gp` variable, so that any concurrent instance of `synchronize_rcu()` will see an odd-numbered value, thus becoming aware that a new RCU read-side critical section has started. Instances of `synchronize_rcu()` that are waiting on older RCU read-side critical sections will know to ignore this new one. Finally, line 17 executes a memory barrier to ensure that the update to `rcu_reader_qs_gp` is seen by all threads to happen before any subsequent RCU read-side critical sections.

Some applications might use RCU only occasionally, but use it very heavily when they do use it. Such applications might choose to use `rcu_thread_online()` when starting to use RCU and `rcu_thread_offline()` when no longer using RCU. The time between a call to `rcu_thread_offline()` and a subsequent call to `rcu_thread_online()` is an extended quiescent state, so that RCU will not expect explicit quiescent

states to be registered during this time.

The `rcu_thread_offline()` function simply sets the per-thread `rcu_reader_qs_gp` variable to the current value of `rcu_gp_ctr`, which has an even-numbered value. Any instance of `synchronize_rcu()` will thus know to ignore this thread. A memory barrier is needed at the beginning of the function to ensure all RCU read side-effects are globally visible before making the thread appear offline. No memory barrier is needed in the innermost part of `rcu_thread_offline()` because it is invalid to perform RCU accesses on this side of the function. There is therefore no need to prevent reordering.

The `rcu_thread_online()` function is the counterpart of `rcu_thread_offline()`. It marks the end of the extended quiescent state. It is similar to `rcu_quiescent_state()`, except that the only memory barrier required is at the end of the function.

Figure 6.5 shows the implementation of `synchronize_rcu()`. It implicitly refers to the variables declared in Lines 1–2 of Figure 6.4. Lines 1–4 show the `rcu_gp_ongoing()` helper function, which returns true if the specified thread's `rcu_reader_qs_gp` variable has an odd-numbered value. Lines 6–22 show the implementation of `synchronize_rcu()` itself. Line 10 is a memory barrier that ensures that the caller's mutation of the RCU-protected data structure is seen by all CPUs to happen before the grace period identified by this invocation of `synchronize_rcu()`. Line 11 ac-

```
 1 static inline int rcu_gp_ongoing(int thread)
 2 {
 3   return per_thread(rcu_reader_qs_gp, thread) & 1;
 4 }
 5
 6 void synchronize_rcu(void)
 7 {
 8   int t;
 9
10   smp_mb();
11   mutex_lock(&rcu_gp_lock);
12   rcu_gp_ctr += 2;
13   for_each_thread(t) {
14     while (rcu_gp_ongoing(t) &&
15            ((per_thread(rcu_reader_qs_gp, t) -
16              rcu_gp_ctr) < 0)) {
17       poll(NULL, 0, 10);
18       barrier();
19     }
20   }
21   mutex_unlock(&rcu_gp_lock);
22   smp_mb();
23 }
```

Figure 6.5 RCU update side using quiescent states

quires a pthread_mutex named rcu_gp_lock in order to serialize concurrent calls to synchronize_rcu(), and line 21 releases it. Line 12 adds the value "2" to the global variable rcu_gp_ctr to indicate the beginning of a new grace period. Line 13 sequences through all threads, and lines 14–16 check to see if the current thread is still in an RCU read-side critical section that began before the counter was incremented back on line 12: if so, we must wait for it on line 17. Line 18 ensures that the compiler refetches the rcu_reader_qs_gp variable. Line 22 executes one last memory barrier to ensure that all other CPUs have fully completed their RCU read-side critical sections before the caller of synchronize_rcu() performs any destructive actions (such as freeing up memory).

This implementation has low-cost read-side primitives, as can be seen in Figure 6.4. Read-side overhead depends on how often rcu_quiescent_state() is called. These read-side primitives qualify as wait-free under the most severe conceivable definition [66]. The synchronize_rcu() overhead ranges from about 600 nanoseconds on a single-CPU Power5 system up to more than 100 microseconds on a 64-CPU system with one thread per CPU.

Because it waits for readers to complete, synchronize_rcu() does not qualify as non-blocking. Section 6.4.5 describes how RCU updates can support wait-free algorithms in the same sense as wait-free algorithms are supported by garbage collectors.

However, this implementation requires that each thread either invoke the primitive rcu_quiescent_state() periodically or invoke rcu_thread_offline() for extended quiescent states. The need to invoke these functions periodically can make this implementation difficult to use in some situations, such as for certain types of library functions.

In addition, this implementation does not permit concurrent calls to synchronize_rcu() to share overlapping grace periods. That said, one could easily imagine a production-quality RCU implementation based on this version of RCU.

Finally, on systems where the rcu_gp_ctr is implemented using 32-bit counters, this algorithm can fail if a reader is preempted in line 3 of rcu_read_lock() in Figure 6.4 for enough time to allow the rcu_gp_ctr to advance through more than half (but not all) of its possible values. Although one solution is to avoid 32-bit systems, 32-bit systems can be handled by adapting rcu_read_lock() and rcu_read_unlock() from Figure 6.6 for use in rcu_quiescent_state() and rcu_offline_thread(), respectively. This would of course also require adopting the synchronize_rcu() im-

plementation from Figure 6.7.

Another point worth discussing is that if read-side critical sections are expected to execute in a signal handler, the `rcu_quiescent_state()` primitive must run with signals disabled, and signals must be kept disabled while threads are kept offline. Effectively, if a signal handler nests over `rcu_quiescent_state()` between the memory barriers, the read-side could be interleaved with the `rcu_reader_qs_gp` update and therefore spawn across two grace periods, which could cause `synchronize_rcu()` to return before the quiescent state is reached and lead to data corruption.

The next section discusses an `RCU` implementation that is safe for use in libraries, where the library code cannot guarantee that all threads of a yet-as-unwritten application will traverse quiescent states in a timely fashion.

6.4.3 General-Purpose RCU

The general-purpose `RCU` implementation can in theory be used in any software environment, including even in library functions that are not aware of the design of the enclosing application. However, the price paid for this generality is relatively high read-side overhead, though this overhead is still significantly less than a single compare-and-swap operation on most hardware.

A global variable `rcu_gp_ctr` is initialized to 1 and a per-thread variable `rcu_reader_gp` is initialized to zero. The low-order bits of `rcu_reader_gp` is a count of the `rcu_read_lock()` nesting depth, while the upper bit indicates the grace-period phase at the time of the invocation of the outermost `rcu_read_lock()` [67]. The upper bit of global variable `rcu_gp_ctr` is the current grace-period phase, while the low-order field is set to the value 1 for reasons that will become apparent shortly.

The read-side primitives are shown in Figure 6.6. Lines 1–4 are declarations, lines 6–19 are `rcu_read_lock()`, and lines 21–27 are `rcu_read_unlock()`.

In `rcu_read_lock()`, line 11 obtains a reference to the current thread's instance of `rcu_reader_gp`, and line 12 fetches the contents into the local variable `tmp`. Line 13 then checks to see if this is the outermost `rcu_read_lock()`, and, if so, line 14 copies the current value of the global `rcu_gp_ctr` to this thread's `rcu_reader_gp` variable, thereby snapshotting the current grace-period phase and setting the nesting count to 1 in a single operation. Otherwise, line 17 increments the nesting count in this thread's `rcu_reader_gp` variable.

```
 1 #define RCU_GP_CTR_BOTTOM_BIT 0x80000000
 2 #define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BOTTOM_BIT - 1)
 3 long rcu_gp_ctr = 1;
 4 DEFINE_PER_THREAD(long, rcu_reader_gp);
 5
 6 static inline void rcu_read_lock(void)
 7 {
 8   long tmp;
 9   long *rrgp;
10
11   rrgp = &__get_thread_var(rcu_reader_gp);
12   tmp = *rrgp;
13   if ((tmp & RCU_GP_CTR_NEST_MASK) == 0) {
14     *rrgp = ACCESS_ONCE(rcu_gp_ctr);
15     smp_mb();
16   } else {
17     *rrgp = tmp + 1;
18   }
19 }
20
21 static inline void rcu_read_unlock(void)
22 {
23   long tmp;
24
25   smp_mb();
26   __get_thread_var(rcu_reader_gp)--;
27 }
```

Figure 6.6 RCU read side using memory barriers

Line 26 decrements the thread's `rcu_reader_gp`, which has the effect of decrementing the nesting count.

For outermost read-side `rcu_read_lock()`, the memory barrier on line 15 ensures that the `rcu_reader_gp` value is globally observable before any of the outermost read-side critical section memory accesses. It ensures that neither the compiler nor the CPU will reorder memory accesses across this barrier by adding a compiler barrier and issuing a memory barrier instruction. Only the outermost `rcu_read_lock()` needs to have such memory barrier because only this outermost lock can change the reader's current grace period.

In `rcu_read_unlock()`, line 25 executes a memory barrier to ensure that all globally observable effects of the `RCU` read-side critical section reach memory before `rcu_reader_gp` is decremented. The memory barrier on line 25 is needed only for the outermost `rcu_read_unlock()`, but given the outermost and innermost nesting level behave in the exact same way, a branch in the `rcu_read_unlock()` code is unneeded, and given the common case is to perform single-level nesting, the memory barrier is executed unconditionally for innermost and outermost nesting levels.

Section 6.4.4 shows one way of getting rid of both memory barriers; however, even with the memory barriers, both `rcu_read_lock()` and `rcu_read_unlock()` are wait-free.

The effect of this implementation of `rcu_read_lock()` and `rcu_read_unlock()` is that a given thread may be ignored by the current grace-period phase in either of the following cases:

1. The lower-order bits of the thread's `rcu_reader_gp` variable are all zero, in which case the thread is not currently in an `RCU` read-side critical section.
2. The upper bit of the thread's `rcu_reader_gp` variable matches that of the global `rcu_gp_ctr`, in which case this thread's `RCU` read-side critical section started after the beginning of the current grace-period phase.

These checks are implemented by the function `rcu_old_gp_ongoing()`, which is shown on lines 1–7 of Figure 6.7. This figure implicitly refers to the declarations and variables in Lines 1–4 of Figure 6.6. Given a thread `t`, line 3 fetches `t`'s `rcu_-`

```
 1 static inline int rcu_old_gp_ongoing(int t)
 2 {
 3   int v = ACCESS_ONCE(per_thread(rcu_reader_gp, t));
 4
 5   return (v & RCU_GP_CTR_NEST_MASK) &&
 6          ((v ^ rcu_gp_ctr) & ~RCU_GP_CTR_NEST_MASK);
 7 }
 8
 9 static void flip_counter_and_wait(void)
10 {
11   int t;
12
13   rcu_gp_ctr ^= RCU_GP_CTR_BOTTOM_BIT;
14   for_each_thread(t) {
15     while (rcu_old_gp_ongoing(t)) {
16       poll(NULL, 0, 10);
17       barrier();
18     }
19   }
20 }
21
22 void synchronize_rcu(void)
23 {
24   smp_mb();
25   mutex_lock(&rcu_gp_lock);
26   flip_counter_and_wait();
27   flip_counter_and_wait();
28   mutex_unlock(&rcu_gp_lock);
29   smp_mb();
30 }
```

Figure 6.7 RCU update side using memory barriers

reader_gp variable, with the ACCESS_ONCE() primitive ensuring the variable is read with a single memory access. This prevents the compiler from refetching the variable or fetching it in pieces. Line 5 then checks to see if the low-order field is non-zero, and line 6 checks to see if the upper bit differs from that of the rcu_gp_ctr global variable. Only if both these conditions hold does rcu_old_gp_ongoing() report that the current grace-period phase must wait on this thread.

Lines 9–20 of Figure 6.7 show flip_counter_and_wait(), which initiates a grace-period phase and waits for it to elapse. Line 13 complements the upper bit of global variable rcu_gp_ctr, which initiates a new grace-period phase. Line 14 cycles through all threads. The "while" loop at line 15 repeatedly executes lines 16–17 until rcu_old_gp_ongoing() reports that the thread no longer resides in an RCU read-side critical section that affects the current grace-period phase. Line 16, which is optional, blocks for a short period of time, and line 17 ensures that the compiler refetches variables when executing rcu_old_gp_ongoing().

Lines 22–30 of Figure 6.7 shows synchronize_rcu(), which waits for a full two-phase grace period to elapse. Line 24 executes a memory barrier to ensure that any prior data-structure modification is seen by all threads to precede the grace period. Line 25 acquires rcu_gp_lock to serialize any concurrent invocations of synchronize_rcu(). Lines 26–27 wait for two grace-period phases, line 28 releases the lock, and line 29 executes a memory barrier to ensure that all threads see the grace period happening before any subsequent destructive operations (such as free()).

Memory ordering between the rcu_gp_ctr complement and testing the reader's current grace period with rcu_old_gp_ongoing() is not strictly needed. The only requirement is that each and every reader thread that was executing in a read-side critical section before memory barrier on line 24 has finished its critical section after the memory barrier on line 29. This two-phase grace period scheme is used to ensure updater progress through a grace period even if a steady flow of readers comes. The only requirement is that, when the updater busy-loops waiting for readers, it eventually reaches a point where all new readers are in the new grace period parity.

Grace period identification, by either a bit (in the two-phase scheme) or by a counter, ensures that readers starting during the grace period will not prevent the grace period from completing. In fact, if a simplistic scheme where the updater waits for *all* readers to complete would be used, the grace period would be considered as complete when the updater reaches a point where no reader is active in the system.

However, this would allow new readers starting after the beginning of the grace period to impede reaching quiescent state. This would prevent grace-period progress in the presence of reader threads releasing the read-side critical section for very short periods. Faster cached local data access would therefore provide an unfair advantage to the reader over the updater.

Now that the grace period identification question is settled, this raises the question "why isn't a single grace-period phase sufficient?" To see why, consider the following sequence of events which involves one read-side critical section and two consecutive grace periods:

1. Thread A invokes `rcu_read_lock()`, executing lines 11–13 of Figure 6.6, and finding that this instance of `rcu_read_lock()` is not nested, fetching the value of `rcu_gp_ctr` on line 14, but not yet storing it.
2. Thread B invokes `synchronize_rcu()`, executing lines 24 and 25 of Figure 6.7, then invoking `flip_counter_and_wait()` on line 26, where it complements the grace-period phase bit on line 13, so that the new value of this bit is now 1.
3. Because no thread is in an `RCU` read-side critical section (recall that thread A has not yet executed the store operation on line 14), Thread B proceeds through lines 14–19 of Figure 6.7, returns to `synchronize_rcu()`, executing lines 28–30 (recall that line 27 is omitted in this scenario), and returning to the caller.
4. Thread A now performs the store in line 14 of Figure 6.6. Recall that it is using the old value of `rcu_gp_ctr` where the value of the grace-period phase bit is 0.
5. Thread A then executes the memory barrier on line 15, and returns to the caller, which proceeds in to the `RCU` read-side critical section.
6. Thread B invokes `synchronize_rcu()` once more, again complementing the grace-period phase bit on line 13 of Figure 6.7, so that the value is again zero.
7. When Thread B examines Thread A's `rcu_reader_gp` variable on line 6 of Figure 6.7, it finds that the grace-period phase bit matches that of the global variable `rcu_gp_ctr`. Thread A is therefore ignored, and Thread B therefore exits from `synchronize_rcu()`.
8. But Thread A is still in its `RCU` read-side critical section in violation of `RCU` semantics.

Invoking `flip_counter_and_wait()` twice avoids this problem by making sure the grace period waits for reader critical sections for each of the possible two phases.

A single-phase approach is possible if the current grace period is identified by a free-running counter, as shown in Section 6.4.2. However, the counter size is important because this counter is subject overflow. The single-flip problem shown above, which involves two consecutive grace periods, is actually a case where a single-bit overflow occurs. A similar scenario is therefore possible given a number of grace periods sufficient to overflow the grace period counter passing during a read lock section. This could realistically happen on 32-bit architectures if read-side critical sections are preempted.

The following section shows one way to eliminate the read-side memory barriers.

6.4.4 Low-Overhead RCU Via Signal Handling

The largest sources of overhead for the `QSBR` and general-purpose `RCU` read-side primitives shown in Figures 6.4 and 6.6 are the memory barriers. One way to eliminate this overhead is to use POSIX signals. The readers' signal handlers contain memory-barrier instructions, which allows an updater to force readers to execute a memory-barrier instruction only when needed, rather than suffering the extra overhead during every call to a read-side primitive.

One unexpected but quite pleasant surprise is that this approach results in relatively simple read-side primitives. In contrast, those of preemptable `RCU` are notoriously complex.

The read-side primitives are shown in Figure 6.8, along with the data definitions and state variables. The `urcu_` prefix used for variables stands for "user-space `RCU`" Lines 1–3 show the definitions controlling both the `urcu_gp_ctr` global variable (line 5) and the `urcu_active_readers` per-thread variable (line 6). The low-order bits (those corresponding to 1-bit in `RCU_GP_CTR_NEST_MASK`) are used to count the `rcu_read_lock()` nesting level, while the bit selected by `RCU_GP_CTR_BIT` is used to detect grace periods. All other bits are unused. The global `urcu_gp_ctr` may be accessed at any time by any thread, but may be updated only by the thread holding the lock that guards grace-period detection. The per-thread `urcu_active_readers` variable may be modified only by the corresponding thread, and is otherwise read only by the thread holding the lock that guards grace-period detection.

The `rcu_read_lock()` implementation is shown on lines 9–18. Line 12 picks up the current value of this thread's `urcu_active_readers` variable and places it in the

```
 1 #define RCU_GP_COUNT         (1UL << 0)
 2 #define RCU_GP_CTR_BIT       (1UL << (sizeof(long) * 4))
 3 #define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BIT - 1)
 4
 5 long urcu_gp_ctr = RCU_GP_COUNT;
 6 long __thread urcu_active_readers = 0L;
 7
 8 static inline void rcu_read_lock(void)
 9 {
10   long tmp;
11
12   tmp = urcu_active_readers;
13   if (!(tmp & RCU_GP_CTR_NEST_MASK))
14     urcu_active_readers = ACCESS_ONCE(urcu_gp_ctr);
15   else
16     urcu_active_readers = tmp + RCU_GP_COUNT;
17   barrier();
18 }
19
20 static inline void rcu_read_unlock(void)
21 {
22   barrier();
23   urcu_active_readers = urcu_active_readers - RCU_GP_COUNT;
24 }
```

Figure 6.8 RCU read side using signals

local variable `tmp`. Line 13 checks to see if the nesting-level portion of `urcu_active_readers` is zero (indicating that this is the outermost `rcu_read_lock()`), and, if so, line 14 copies the global variable `urcu_gp_ctr` to this thread's `urcu_active_readers` variable. Note that `urcu_gp_ctr` has been initialized with its low-order bit set, so that the nesting level is automatically set correctly. Otherwise, line 16 increments the nesting level in this thread's `urcu_active_readers` variable. In either case, line 17 executes a barrier directive in order to prevent the compiler from undertaking any code-motion optimization that might otherwise cause the contents of the subsequent `RCU` read-side critical section to be reordered to precede the `rcu_read_lock()`.

The implementation of `rcu_read_unlock()` is shown on lines 20–24. Line 22 executes a barrier directive, again, in order to prevent the compiler from undertaking any code-motion optimization that might otherwise cause the contents of the prior `RCU` read-side critical section to be reordered to follow the `rcu_read_unlock()`. Line 23 decrements the value of this thread's `urcu_active_readers` variable, so that if this is the outermost `rcu_read_unlock()`, the low-order bits indicating the nesting level will now be zero.

Both `rcu_read_lock()` and `rcu_read_unlock()` execute a sharply bounded number of instructions, hence both are wait-free.

```
 1 struct reader_registry {
 2   pthread_t tid;
 3   long *urcu_active_readers;
 4   char *need_mb;
 5 } *registry;
 6 static char __thread need_mb;
 7 static int num_readers;
 8
 9 static void force_mb_all_threads(void)
10 {
11   struct reader_registry *index;
12
13   if (!registry)
14     return;
15   index = registry;
16   for (; index < registry + num_readers; index++) {
17     *index->need_mb = 1;
18     pthread_kill(index->tid, SIGURCU);
19   }
20   index = registry;
21   for (; index < registry + num_readers; index++) {
22     while (*index->need_mb) {
23       pthread_kill(index->tid, SIGURCU);
24       poll(NULL, 0, 1);
25     }
26   }
27   smp_mb();
28 }
29
30 static void sigurcu_handler(int signo, siginfo_t *siginfo,
31                             void *context)
32 {
33   smp_mb();
34   need_mb = 0;
35   smp_mb();
36 }
```

Figure 6.9 RCU signal handling

The signal-handling primitives are shown in Figure 6.9, including variable declarations on lines 1–7, `force_mb_all_threads()` on lines 9–28 and `sigurcu_handler()` on lines 30–36.

The structures on lines 1–5 represents a thread, with its thread ID in `tid`, a pointer to its `urcu_active_readers` per-thread variable, and a pointer to its `need_mb` per-thread variable. Line 6 declares the per-thread `need_mb` variable, and line 7 defines the global variable `num_readers`, which contains the number of threads that are represented in the `registry` array defined on line 5.

The `force_mb_all_threads()` function ensures a memory barrier is executed on each running threads by sending a POSIX signal to all threads, waiting for each to respond. As we will see, this has the effect of promoting compiler-ordering directives

such as `barrier()` to full memory barriers, while avoiding the need to incur the cost of expensive barriers in read-side primitives in the common case. Lines 13–14 return if there are no readers, and lines 16-19 set each thread's `need_mb` per-thread variable to the value one, then send that thread a POSIX signal. Note that the system call executed for `pthread_kill()` implies a full memory barrier before the system call execution at the operating system level. This memory barrier ensures that all memory accesses done prior to the call to `pthread_kill()` are not reordered after the start of the system call. Lines 20–26 then rescan the threads, waiting until one each has responded by setting its `need_mb` per-thread variable to zero. Because some versions of some operating systems can lose signals, line 23 will resend the signal if a response is not received in a timely fashion. Finally, line 27 executes a memory barrier to ensure that the signals have been received and acknowledged before later operations that might otherwise destructively interfere with readers.

Lines 30–36 show the signal handler that runs in response to a given thread receiving the POSIX signal sent by `force_mb_all_threads()`. This `sigurcu_handler()` function executes a pair of memory barriers separated by setting its `need_mb` per-thread variable to zero. This has the effect of placing a full memory barrier at whatever point in the thread's code that was executing at the time that the signal was received, preventing the CPU from reordering across that point.

The sender thread has two memory barriers around whole sequence consisting of sending the signal and waiting for the remote thread to acknowledge its reception. The remote thread executes a memory barrier before acknowledging the signal reception. These two conditions ensure that the remote thread's program order and memory accesses passed by a point where they were executing in order between the two memory barriers on the sender thread. Therefore, execution in program order and with ordered memory accesses is ensured on the remote processor at that point. This promotes all compiler barriers on the receiver side to memory barriers, but only when the matching memory barrier is executed on the sender side.

The update-side grace-period primitives are shown in Figure 6.10. These include the `switch_next_urcu_qparity()` on lines 1–4, `rcu_old_gp_ongoing()` on lines 6–15, `wait_for_quiescent_state()` on lines 17–29, and `synchronize_rcu()` on lines 31–41.

The `switch_next_urcu_qparity()` function starts a new grace-period phase, where a pair of such phases make up a grace period. A single phase is insufficient for the same

```
 1 static void switch_next_urcu_qparity(void)
 2 {
 3   urcu_gp_ctr = urcu_gp_ctr ^ RCU_GP_CTR_BIT;
 4 }
 5
 6 static inline int rcu_old_gp_ongoing(long *value)
 7 {
 8   long v;
 9
10   if (value == NULL)
11     return 0;
12   v = ACCESS_ONCE(*value);
13   return (v & RCU_GP_CTR_NEST_MASK) &&
14      ((v ^ urcu_gp_ctr) & RCU_GP_CTR_BIT);
15 }
16
17 static void wait_for_quiescent_state(void)
18 {
19   struct reader_registry *i;
20
21   if (!registry)
22     return;
23   i = registry;
24   for (; i < registry + num_readers; i++) {
25     while (rcu_old_gp_ongoing(i->urcu_active_readers))
26       cpu_relax();
27     }
28   }
29 }
30
31 void synchronize_rcu(void)
32 {
33   internal_urcu_lock();
34   force_mb_all_threads();
35   switch_next_urcu_qparity();
36   wait_for_quiescent_state();
37   switch_next_urcu_qparity();
38   wait_for_quiescent_state();
39   force_mb_all_threads();
40   internal_urcu_unlock();
41 }
```

Figure 6.10 RCU update side using signals

reasons discussed in Section 6.4.3. This function simply complements the designated bit in the urcu_gp_ctr global variable.

The rcu_old_gp_ongoing() determines whether or not the thread with the referenced per-thread urcu_active_readers variable is still executing within an RCU read-side critical section that started before this grace-period phase. Lines 10–11 check to see if there is no thread, and returns zero if there is not, given that a non-existent thread cannot be executing at all, let alone within an RCU read-side critical section. This will hold for the whole grace-period because thread registration needs to hold the internal_rcu_lock. Otherwise, line 12 fetches the value, using the ACCESS_ONCE() primitive to defeat compiler optimizations that might otherwise cause the value to be fetched more than once. Line 13 then checks to see if the corresponding thread is in an RCU read-side critical section, and, if so, line 14 checks to see if that RCU read-side critical section predates the beginning of the current grace-period phase.

The wait_for_quiescent_state() waits for each thread to pass through a quiescent state, thereby completing one phase of the grace period. Lines 21–22 return immediately if there are no threads. Otherwise, the loop spanning lines 23–28 waits for each thread to exit any pre-existing RCU read-side critical section.

The synchronize_rcu() primitive waits for a full grace period to elapse. Line 33 acquires a pthread_mutex that prevents concurrent synchronize_rcu() invocations from interfering with each other and reader thread registration. Line 40 releases this same pthread_mutex. Line 34 ensures that any thread that sees the start of the new grace period (line 35) will also see any changes made by the caller prior to the synchronize_rcu() invocation. Line 35 starts a new grace-period phase, and line 36 waits for it to complete. Lines 37 and 38 similarly start and end a second grace-period phase. Line 39 forces each thread to execute a memory barrier, ensuring that each thread will see any destructive actions subsequent to the call to synchronize_rcu() as happening after any RCU read-side critical section that started before the grace period began.

Of course, as with the other two RCU implementations, this implementation's synchronize_rcu() primitive is blocking. The next section shows a way to provide wait-freedom to RCU updates as well as to RCU readers.

6.4.5 Wait-Free RCU Updates

Although some algorithms use RCU as a first-class technique, in most situations RCU is instead simply used as an approximation to a garbage collector. In these situations, given sufficient memory, the delays built into synchronize_rcu() need not block the algorithm itself, just as delays built into an automatic garbage collector need not block a wait-free algorithm.

One way of accomplishing this is shown in Figure 6.11, which implements the asynchronous call_rcu() primitive found in the Linux kernel. Lines 4 and 5 initialize an RCU callback, and line 6 uses a wait-free enqueue algorithm [68] to enqueue the callback on the rcu_data list. This call_rcu() function is then clearly wait-free.

A separate thread would remove and invoke these callbacks after a grace period has elapsed, using synchronize_rcu() for this purpose, as shown on lines 9–24 of Figure 6.11, with each pass of the loop spanning lines 14–23 waiting for one grace period. Line 15 uses a (possibly blocking) dequeue algorithm to remove all elements from the rcu_data list en masse, and line 16 waits for a grace period to elapse. Lines 17–21 invoke all the RCU callbacks from the list dequeued by line 15. Finally, line 22 blocks for a short period to allow additional RCU callbacks to be enqueued.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(struct rcu_head *head))
 3 {
 4   head->func = func;
 5   head->next = NULL;
 6   enqueue(head, &rcu_data);
 7 }
 8
 9 void call_rcu_cleanup(void)
10 {
11   struct rcu_head *next;
12   struct rcu_head *wait;
13
14   for (;;) {
15     wait = dequeue_all(head);
16     synchronize_rcu();
17     while (wait) {
18       next = wait->next;
19       wait->func(wait);
20       wait = next;
21     }
22     poll(NULL, 0, 1);
23   }
24 }
```

Figure 6.11 Avoiding update-side blocking by RCU

Note that the longer line 22 waits, the more `RCU` callbacks will accumulate on the `rcu_data` list. This is a classic memory/CPU trade-off, with longer waits allowing more memory to be occupied by `RCU` callbacks, but decreasing the per-callback CPU overhead.

Of course, the use of `synchronize_rcu()` causes `call_rcu_cleanup()` to be blocking. However, as long as the callback function `func` that was passed to `call_rcu()` does nothing other than free memory, as long as the synchronization mechanism used to coordinate `RCU` updates is wait-free, and as long as there is sufficient memory for allocations to succeed without blocking, `RCU`-based algorithms that use `call_rcu()` will themselves be wait-free.

6.5 Experimental Results

This section presents benchmarks of each `RCU` mechanism presented in this paper with respect to each other, compared to mutexes, to reader-writer locks and to per-thread locks [1]. It first demonstrates read-side scalability, discusses the impact of read-side critical section length on the respective locking primitive behavior and finally presents update operation rate impact on read-side performance. The goal of this section is to clearly demonstrate in which situation `RCU` outperforms classic locking solutions to help identifying for which workloads `RCU` can bring performance improvements compared to classic locks in existing applications.

The machines used to run the benchmarks are an 8-core Intel Core2 Xeon E5405 clocked at 2.0 GHz and a 64-core PowerPC POWER5+ clocked at 1.9 GHz. Each core of the PowerPC machine has 2 hardware threads. To eliminate thread-level contention for processor resources, benchmarks are performed with affinity to the 64 even-numbered CPUs of the 128 logical CPUs presented by the system.

The mutex and reader-writer lock implementations used for comparison are the standard pthreads implementations from the GNU C Library 2.7 for 64-bit Intel and GNU C Library 2.5 for 64-bit PowerPC.

`STM` (Software Transactional Memory) is not included in these comparisons because it is already known to incur high overhead and to scale poorly [69]. `HTM` (Hard-

1. The per-thread lock approach consists in using one mutex per reader thread. The updater threads must take all the mutexes, always in the same order, to exclude all readers. This approach ensures reader cache locality at the expense of a slower write-side locking.

ware Transactional Memory) [70, 71, 72] is likely to be more scalable than STM. However, HTM hardware is not available to us due to the fact that it is expensive and not very common, preventing us from including it in our performance results.

6.5.1 Scalability

Figure 6.12 presents the read-side scalability comparison of each RCU mechanism with standard locking primitives for the PowerPC. The goal of this test is to determine how each synchronization primitive performs in heavy read-side scenarios when the number of CPU increases. This is done by executing from 1 to 64 reader threads for 10 seconds, each taking a read-lock, reading a data unit and releasing the lock in a tight loop. No updater thread is present in this test. As a result, we observe that linear scalability is achieved for RCU and per-thread mutex approaches. This is expected, given readers do not need to exchange cache-lines. The QSBR approach is the fastest, followed by the signal-based RCU, general-purpose RCU and per-thread mutex, each adding a constant per-CPU overhead. The Intel Xeon behaves similarly and is not shown here.

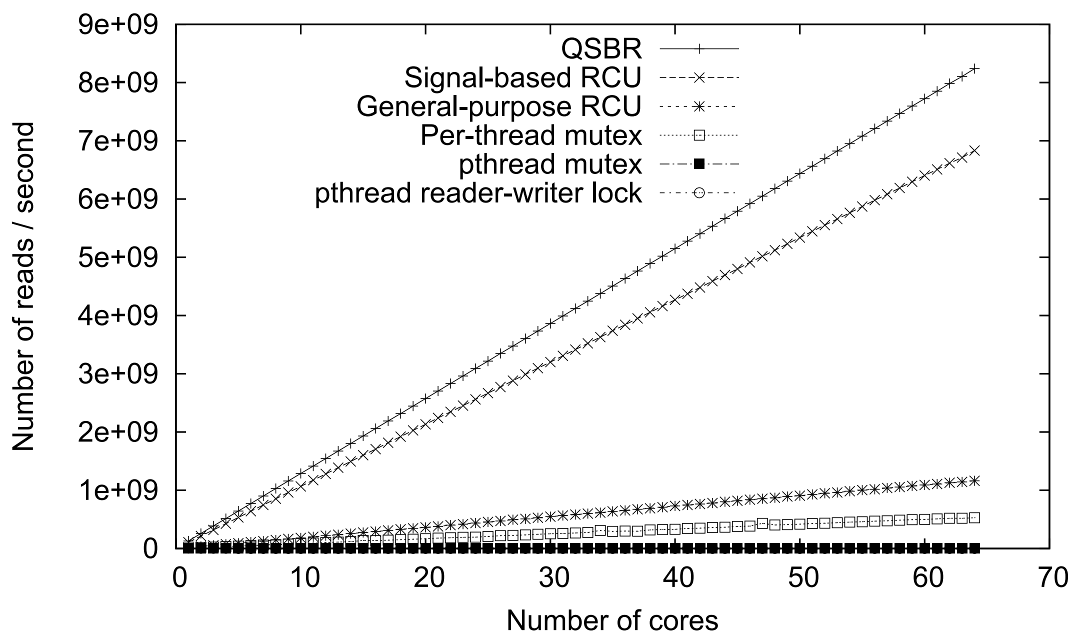

Figure 6.12 Read-side scalability of various synchronization primitives, 64-core POWER5+

However, Figure 6.12 does not show the scalability trend of the pthread mutex and pthread reader-writer lock primitives. This is the purpose of Figure 6.13, which presents scalability of those two primitives. As we can see, with more than 8 cores, overall performance actually decreases when the number of core increases.

6.5.2 Read-Side Critical Section Length

Due to the large performance difference between `RCU` and other approaches, we notice that linear-scaled graphs are not appropriate for the following comparisons.

Therefore, Figure 6.14 presents the read-side critical section length impact using logarithmic x and y axis. This benchmark is performed with 8 reader threads taking the read lock, reading the data structure, waiting for a variable delay and releasing the lock, without any active updater. Interestingly, on this 8-core machine, we notice that starting at about 1000 cycles per critical section, the difference between `RCU` and per-thread locks becomes insignificant. At 20,000 cycles per critical section, the reader-writer locks are almost as fast as the other solutions. Only pthread mutex performance always has significantly worse performance for all critical section lengths.

To appropriately present the 64-core read-side critical section length impact on the read-side speed, we must first introduce the effects that alter the reader-writer lock and mutex behavior, which explain why these two primitives saturate with read-side

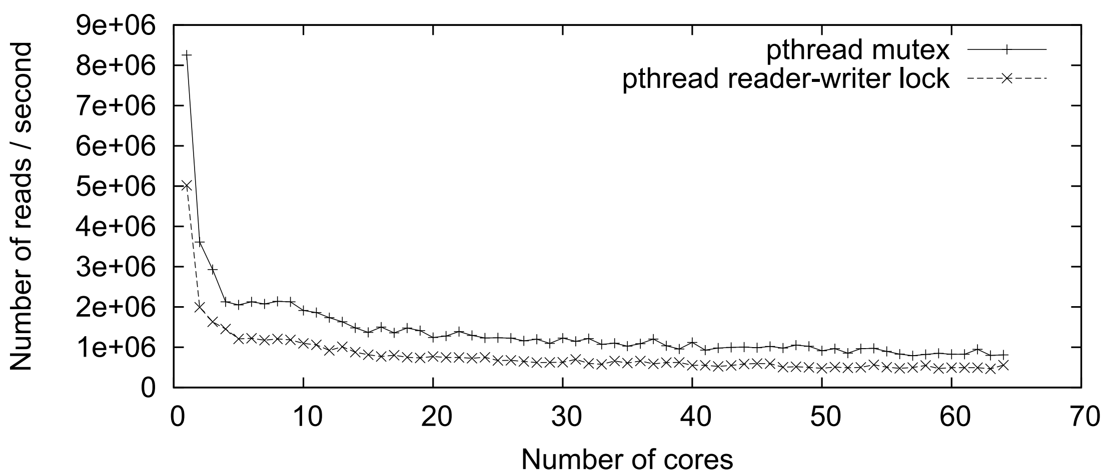

Figure 6.13 Read-side scalability of mutex and reader-writer lock, 64-core POWER5+

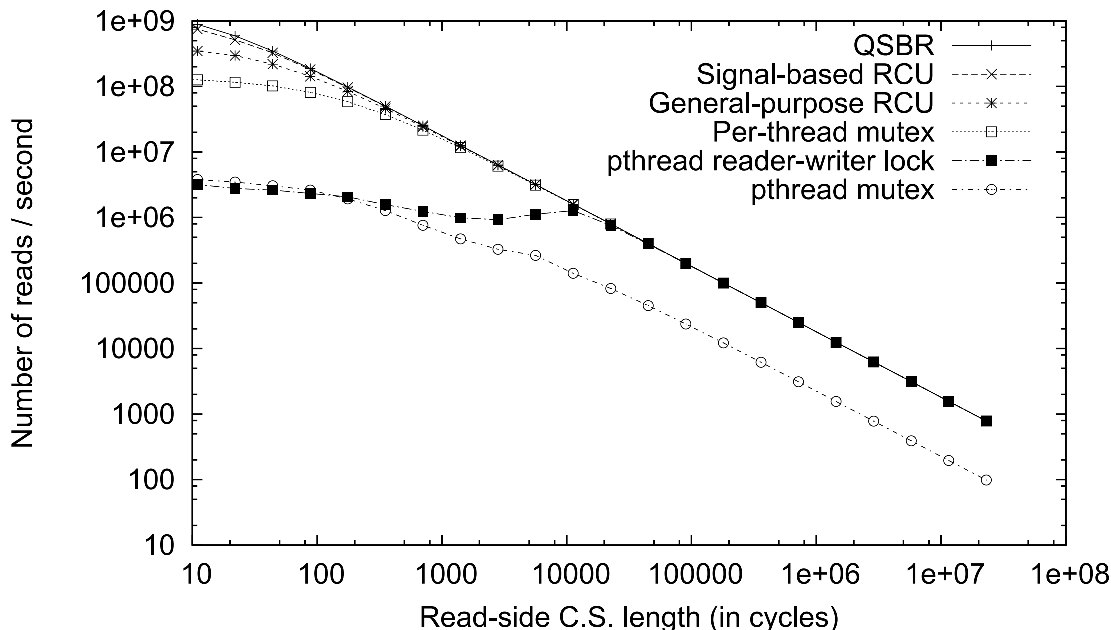

Figure 6.14 Impact of read-side critical section length, 8-core Intel Xeon, logarithmic scale

critical sections still significantly larger than the primitives using per-cpu data for read-side synchronization. First, the interprocessor cache-line exchange time affects the lock access time. Second, the number of cores needing to access the lock also affects the number of lock-access per second.

Therefore, we first present, in Figure 6.15, the equivalent POWER5+ graph with only 8 cores used to specifically show the effect of architecture and cache-line access time change. The cores are spaced by a striding of 8. Changing stride to 1, 2 or 4 (not presented here for brevity) only very slightly affects read speed for reader-writer lock and mutex. Cores close to each other share a common L2 and L3 cache on the POWER5+, which causes reader-writer lock and mutex to be slightly faster at lower striding values. Given it has no significant effect on the read-side critical section length at which the various locking primitives are equivalent, this factor can be left out of the rest of this study.

The same workload executed with 64 reader threads is presented in Figure 6.16. These threads are concurrently reading the data structure with an added variable delay. We notice, when comparing to the 8-core graph in Figure 6.15, that we need a critical section about 10 times larger (20,000 instead of 2000 cycles) before the reader-

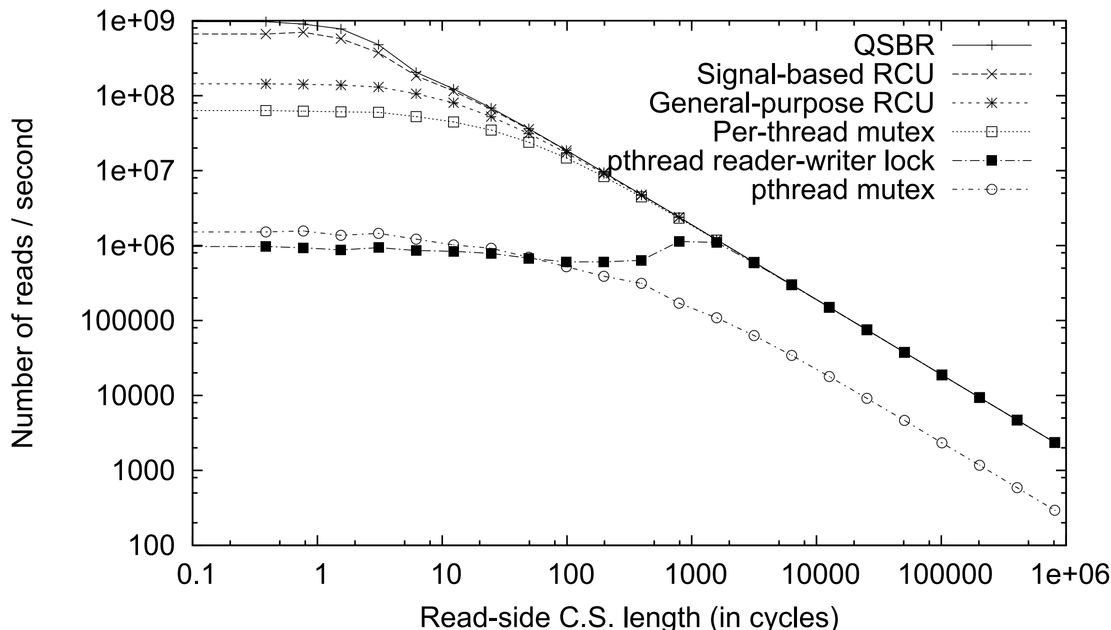

Figure 6.15 Impact of read-side critical section length, 8 reader threads on POWER5+, logarithmic scale

writer lock performance reaches the RCU or per-thread lock performances. Therefore, as we increase the number of cores, reader-writer lock protected critical sections must be larger to behave similarly to RCU and per-thread locks.

6.5.3 RCU Grace-Period Batch Calibration

After looking at read-side only performance, it is appropriate to see how concurrent updates influence the read-side behavior. To appropriately represent the RCU update-side performance impact, we must first calibrate the reclamation batch size to ensure we amortize the grace-period overhead over multiple updates. Such calibration is presented for Intel and POWER5+ in Figures 6.17 and 6.18, respectively for 8 cores and 64 cores. For update operation benchmark, we use half the number of cores for readers and the other half for updaters.

We calibrate with the signal-based RCU approach, likely to provide the highest grace-period overhead due to signal-handler execution. The ideal batch size for both architectures with 8 cores used is determined to be 32768 per updater thread. Given the test duration is 10 sec, we have to eliminate batch sizes large enough to be

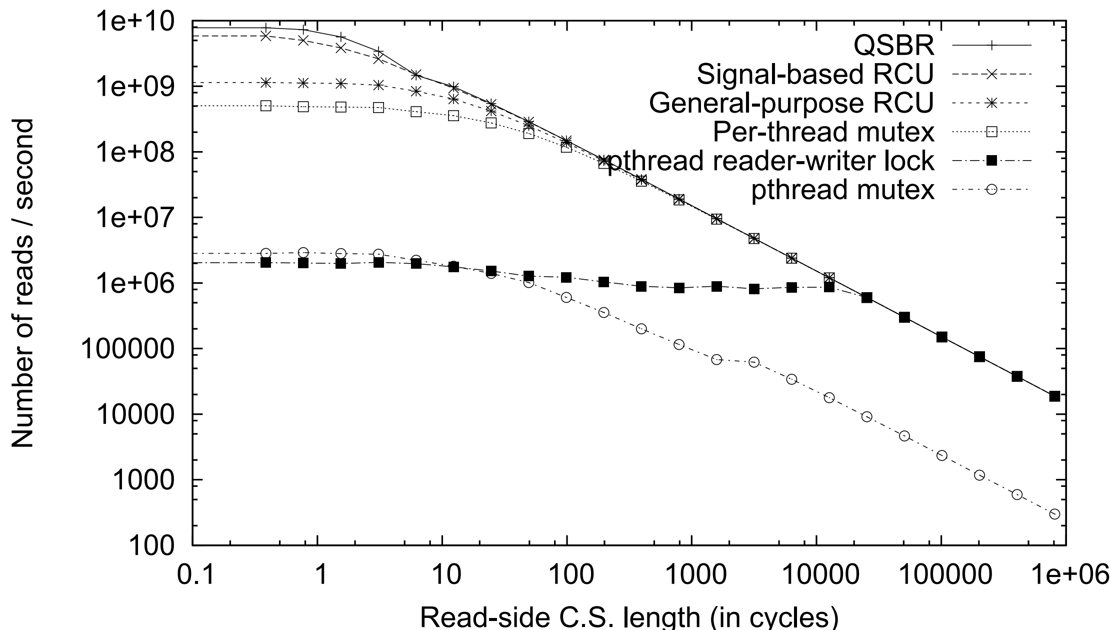

Figure 6.16 Impact of read-side critical section length, 64 reader threads on POWER5+, logarithmic scale

a significant portion of updates performed during the test because non-reclaimed batches are not accounted for. This is why the largest batch sizes are ignored even if they seem slightly better. Figure 6.18 shows that with 64 cores used, the ideal batch size is slightly lower (4096) due to an increased per-update pointer exchange overhead caused by linked-load/store-conditional contention. Therefore, smaller batch sizes are required to amortize the grace-period overhead and perform slightly better due to increased cache locality. However, given the performance difference is not very large, we use a 32768 batch size for both 8-core and 64-core tests.

6.5.4 Update Overhead

Once batch-size calibration is performed, we can proceed to update rate impact comparison. Figure 6.19 presents the impact of update frequency on read-side performance for the various locking primitives. It is performed by running 4 reader and 4 updater threads and varying the delay between updates. We notice that `RCU` approches outperforms the per-thread lock approach especially in terms of maximum updates per second. The former can reach 2 million updates per second while per-

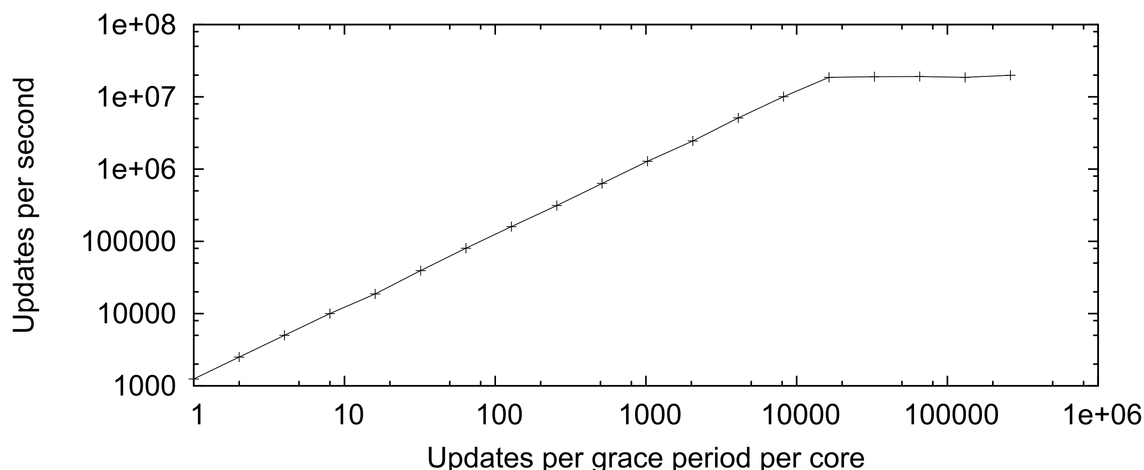

Figure 6.17 Impact of grace-period batch-size on number of update operations, 8-core Intel Xeon, logarithmic scale

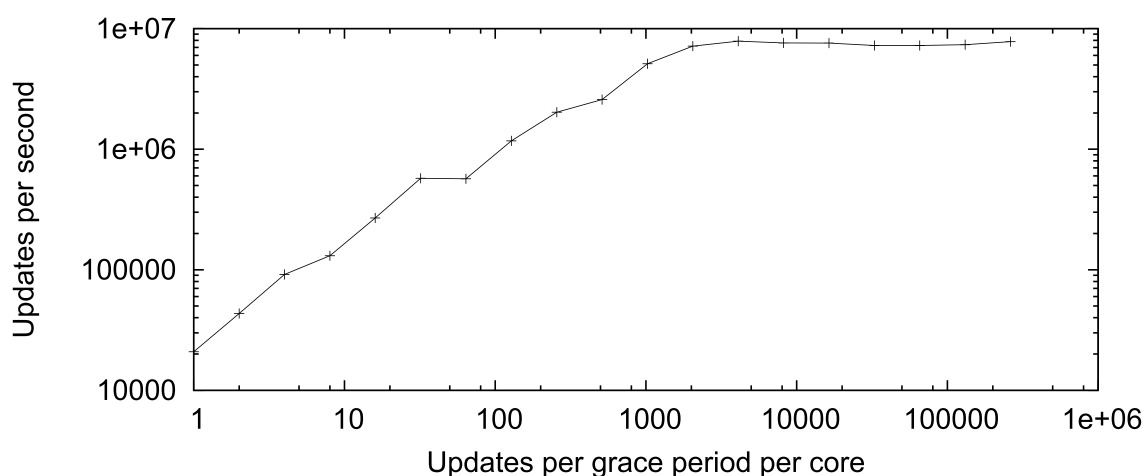

Figure 6.18 Impact of grace-period batch-size on number of update operations, 64-core POWER5+, logarithmic scale

thread locks can only perform 0.1 million updates per second. Interestingly, on such workload with 4 tight loop readers, mutexes outperforms the reader-writer lock primitive in all aspects. Furthermore, reader-writer locks seems to show a case of reader starvation with high updates per second rates.

But while Figure 6.19 presents a fair comparison between the locking primitives, it presents a non-ideal scenario for `RCU`. In our attempt to present a comparison

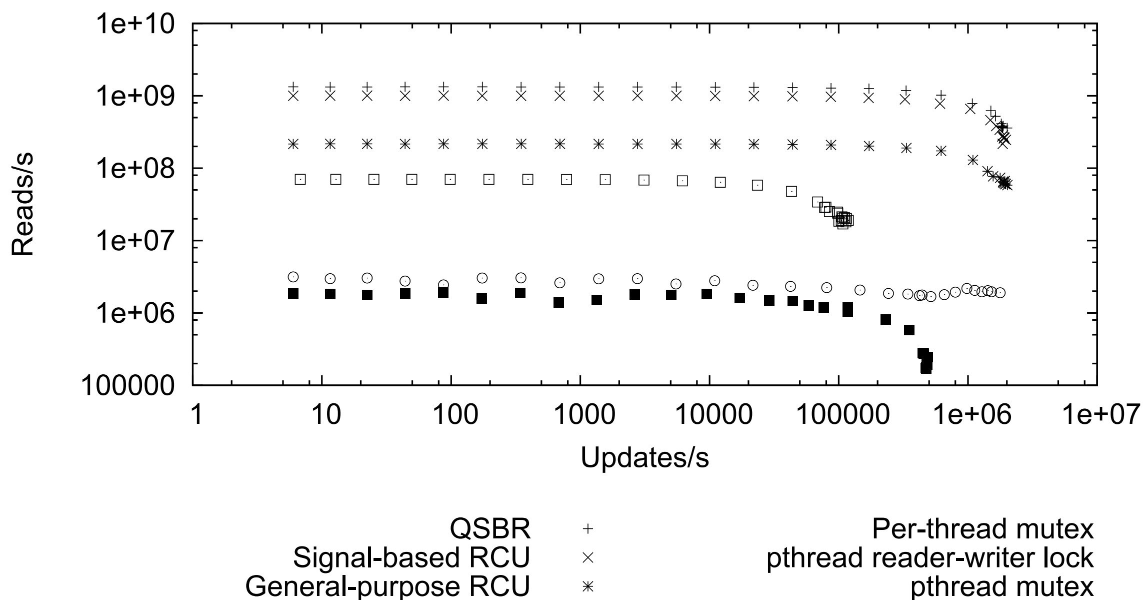

Figure 6.19 Update overhead, 8-core Intel Xeon, logarithmic scale

where all locking primitives perform equivalent work, this figure includes the `RCU` pointer exchange overhead. To factor out this overhead, Figure 6.20 shows both ideal `RCU` grace-period performance and the equivalent with added pointer exchange. This shows that it is the pointer exchange that becomes the update-rate bottleneck, not the grace period. Therefore, in an ideal scenario where pointers updates would be local to each thread, `RCU` could be expected to preserve its read-side scalability characteristics even under frequent updates. Such local updates could be ensured by appropriately designed list or hash table data structures.

Figure 6.21 shows the update overhead on a 64-core POWER5+, with 32 reader and 32 updater threads. We can conclude that `RCU QSBR` and general purpose approaches reach the highest update rates, even compared to mutexes. This is attributed to the lower performance overhead for exchanging a pointer compared to the multiple atomic operations and memory barriers implied by acquiring and releasing a mutex. Mutex-based benchmark performance seems to drop starting at 30,000 updates per second with 32 updater threads. A similar effect is present with only 4 updater threads (graph not presented for brevity). Figure 6.19 seemed to show that update overhead stayed constant even at higher update frequency for 4 updater threads on

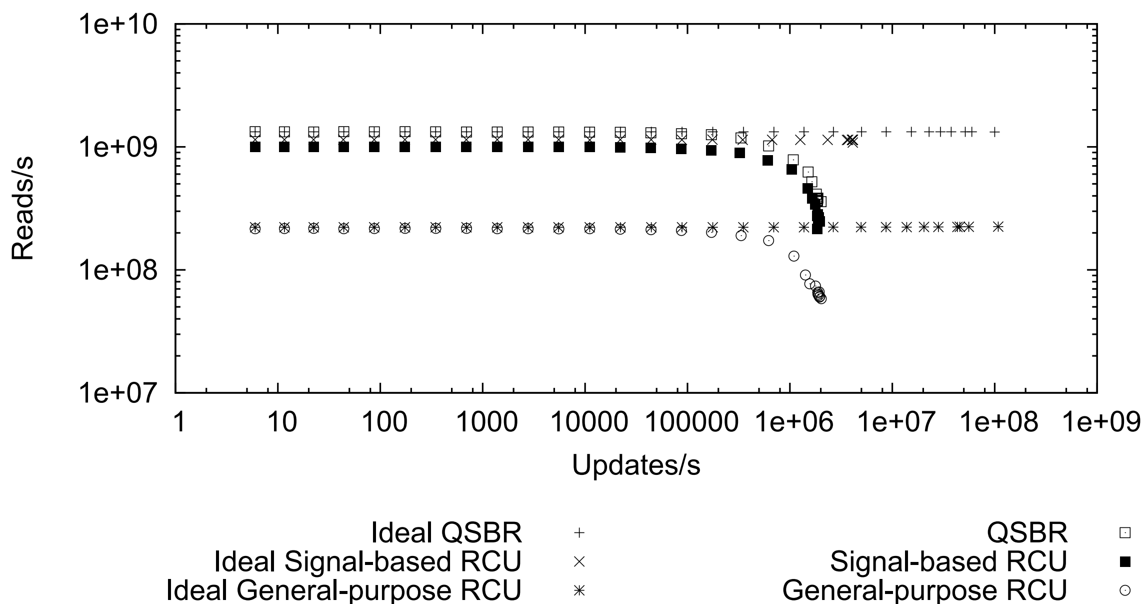

Figure 6.20 Impact of pointer exchange on update overhead, 8-core Intel Xeon, logarithmic scale

the Xeon. Therefore, as the number of concurrent updaters increases, mutex behavior seems to depend on the architecture and on the specific GNU C Library version. Two approaches seems to be significantly affected by increasing the number of updaters. The reader-writer lock, where updaters clearly seem to be starved by readers, has a maximum update rate of 175 updates per second. Per-thread locks are limited to a maximum update rate of 10,000 updates per second with 32 reader threads.

Finally, Figure 6.22 presents, as previously done for Xeon, how grace-period detection (ideal `RCU`) compares to `RCU` grace period with pointer exchange. Therefore, with appropriately designed data structures, better update locality would ideally lead to constant updater overhead as the update frequency increases.

6.6 Conclusions

We have presented a set of `RCU` implementations covering a wide spectrum of application architectures. `QSBR` shows the best performance characteristics, but severely constrains the application architecture by requiring each reader thread to periodically pass through a quiescent state. Signal-based `RCU` performs almost as well as `QSBR`,

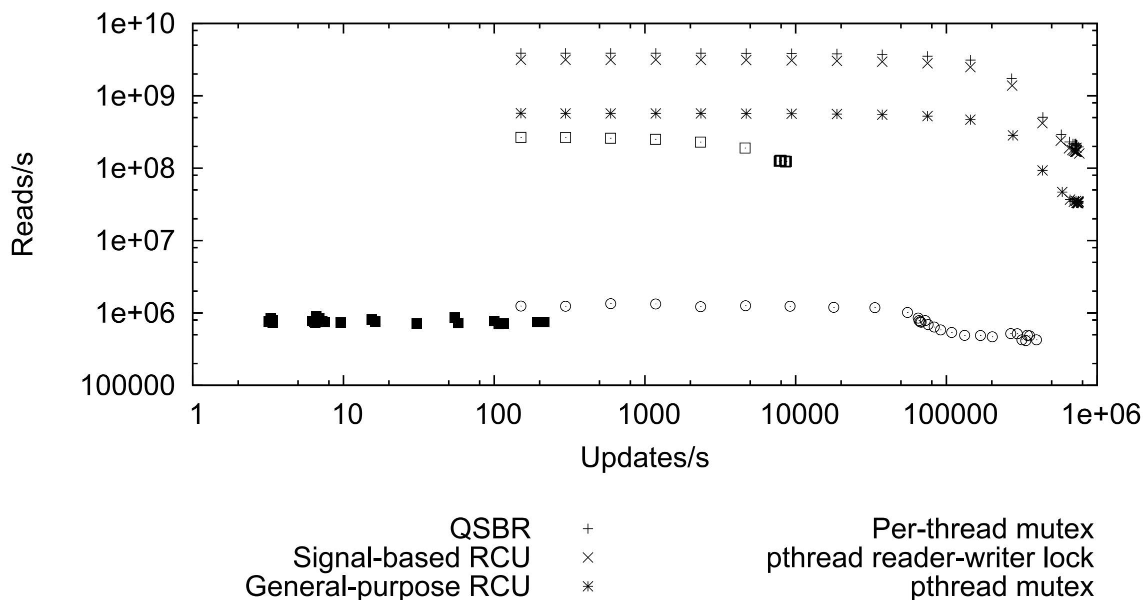

Figure 6.21 Update overhead, 64-core POWER5+, logarithmic scale

but requires reserving a signal. Unlike the other two, general-purpose `RCU` incurs significant read-side overhead. However, it minimizes constraints on application architecture, requiring only that each thread invoke an initialization function before entering its first `RCU` read-side critical section.

Benchmarks demonstrate read-side linear scalability of the `RCU` and per-thread lock approaches. It also shows that the smallest read-side critical section duration for which reader-writer locks, `RCU` and per-thread lock approaches are nearly equivalent in terms of read-side performance impact grows larger as the number of cores increases. These benchmarks also show that, by performing memory reclamation in batch, `RCU` approaches reach update rates much higher than reader-writer locks, per-thread locks and mutexes on similar workloads where updates are performed on a shared data structure. Furthermore, given ideal data structures preserving update cache locality, `RCU` approaches are shown to have a constant update overhead as update frequency increases. Therefore, the upper-bound for `RCU` update overhead is demonstrated to be far below lock-based overhead. Furthermore, it is still possible to decrease `RCU` update-side overhead even more by designing data structures providing good update cache-locality.

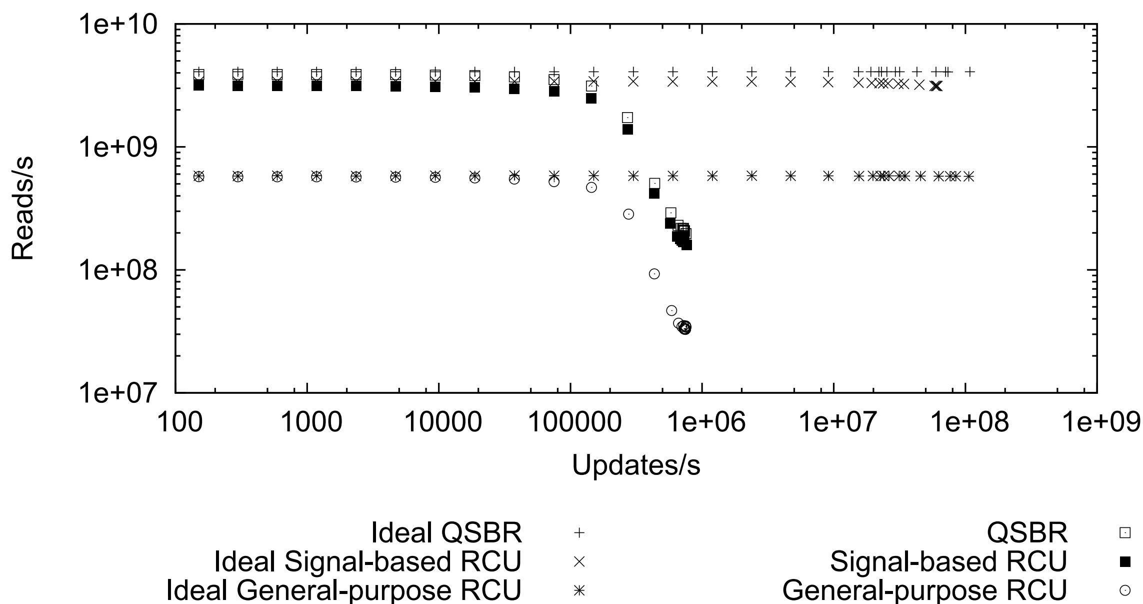

Figure 6.22 Impact of pointer exchange on update overhead, 64-core POWER5+, logarithmic scale

Acknowledgements

We owe thanks to Maged Michael for many illuminating discussions, to Kathy Bennett for her support of this effort, to Etienne Bergeron and Alexandre Desnoyers for reviewing this paper.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851. This work is funded by Google, Natural Sciences and Engineering Research Council of Canada, Ericsson and Defence Research and Development Canada.

Legal Statement

This work represents the views of the authors and does not necessarily represent the view of Ecole Polytechnique de Montreal, Harvard, IBM, or Portland State University.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

Chapter 7

Paper 4: Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms

Abstract

Modeling parallel algorithms at the architecture level permits to explore side-effects of weak ordering performed by modern processors. Formal verification of such models with model-checking can ensure that algorithm guarantees will hold even in the presence of the most aggressive compiler and processor optimizations.

This paper proposes a virtual architecture to model the effects of such optimizations. It first presents the *OoOmem* framework to model out-of-order memory accesses. It then presents the *OoOisched* framework to model the effects of out-of-order instruction scheduling.

These two frameworks are explained and tested using weakly-ordered memory interaction scenarios known to be affected by weak ordering. Then, modeling of user-level `RCU` (Read-Copy Update) synchronization algorithms is presented. It uses the virtual architecture proposed to verify that the `RCU` guarantees are indeed respected.

7.1 Introduction

Formal verification of synchronization primitives for shared memory multiprocessor architectures is undoubtedly useful due to the architecture and context dependency of bug occurrence. An algorithm can appear bug-free when used in a large set of test cases. However, testing is unable to certify that compiler optimizations done for a different invocation of a synchronization primitive will work as expected. Portability is also hard to certify with testing, because it would involve testing the

primitives on all architecture variants.

Our principal motivation is to create detailed architecture-level models taking into account weak instruction scheduling and memory access ordering able to verify the user-space `RCU` (Read-Copy Update) implementations presented in [3]. The complexity level of these algorithms makes formal verification well worthwhile.

This paper first depicts the modeling challenges, then summarizes the `LTL` model-checking principles and introduces modeling of parallel algorithms. This is followed by a presentation of the frameworks created to accurately model real architectures. Finally, the `RCU` library modeling and its verification are presented.

7.2 Modeling Challenges

Modeling multi-core systems for formal verification of parallel algorithm implementations brings interesting challenges. These are caused by the architecture and compiler ordering semantics. These challenges come from the presence of:

– compiler-level optimizations,
– execution of nested signal handlers,

on architectures with the following characteristics:

– shared-memory multiprocessor,
– weak memory-ordering,
– pipelined and superscalar,
– out-of-order instruction scheduling.

All these challenges are a direct result of our desire to model algorithms for production use. If these models were instead meant to be only applied to prototypes, we might be strongly tempted to assume a non-optimizing compiler and almost inexistent sequentially consistent machines to avoid dealing with optimization, out-of-order execution and out-of-order memory access effects. We might also be tempted to assume signal handlers out of existence as a simplification.

Concurrent algorithms have been modeled on weakly ordered systems in the past [73], and interrupts (similar to signals) have been modeled as well [74]. However, the methods used in this past work to model weakly-ordered systems can result in combinatorial explosion of the model. Models specifically covering the x86 [75], as well as PowerPC and ARM [76] architectures for parallel algorithm verification have also been proposed in the past. In this paper, we present a more general approach

that allows the model to more closely follow the architecture behavior than the previous RCU models and to take into account the weakest ordering amongst multiple architectures, therefore modeling weak-ordering effects more accurately.

To further reduce the computational requirements of the validation process, we approximate the properties of the actual hardware, but in all cases modeling weaker ordering than the actual hardware provides. This weaker-ordering approximation ensures that any algorithm passing our validation will run correctly on conforming hardware.

We propose a model representing the interprocessor interactions, which we call our *virtual architecture*. In this architecture, each Promela [77] process represents a processor. To model CISC architectures, complex instructions are divided into micro-operations, which are then represented as atomic Promela statements. It results in a RISC architecture, which allows to easily detail micro-operations dependencies. Through this article, micro-operation and instruction will be used as synonyms, given those apply to our RISC virtual architecture.

The models proposed here specifically use a data flow representation of each processor, where each node repesents a micro-operation and where each arc represents a data or control dependency between micro-operations. This permits to model accurately all possible micro-operation interleavings as seen from the point of view of their visible effect outside of the processor. We model the interactions between processors by creating a cache-memory interaction model.

7.3 Modeling and Model-Checking

This section summarizes the principles of LTL model-checking and presents an introduction to modeling of parallel algorithms. It constitutes the background on which the rest of this article is constructed.

7.3.1 LTL Model-Checking

The verification is carried out using Promela [77], a special-purpose modeling language that performs a full state-space search. This in turn allows all possible execution histories to be examined for specified classes of errors, including race conditions and livelock conditions.

The equivalence between data flow analysis and model-checking of abstract interpretations has been used [78, 79] to model simple sequential programs. Data flow analysis has also been used to verify properties of concurrent programs [80].

We represent our model in Promela and perform verification of properties expressed as `LTL` (Linear Temporal Logic) formulas [1]. The model description expresses atomic statements executed by one or more processes. The Spin verifier [81, 82] transforms the model into a Büchi [83] automaton. The `LTL` formulas are transformed in the negation of never-claims suited for verification of the model. The Spin verifier visits all atomic statements required to validate the `LTL` claims in all execution orders allowed by the model. One sequence used to visit atomic statements in a particular order forms a path.

`LTL` formalism allows basic logical operators within predicates:

$\Rightarrow$: logical implication,

$\Longleftrightarrow$: equivalence,

$\wedge$: conjunction,

$\vee$: disjunction,

$\neg$: negation.

To reason over the future of paths, temporal operators can be applied to predicates:

G: Temporal operator *always* (a predicate will always hold),

F: Temporal operator *eventually* (a predicate must eventually become true),

U: Strong until (will hold until another predicate is true).

Model checking permits to explore all execution scenarios required to verify a specific `LTL` property. Unlike simulation-based approaches, the model-checker only needs to generate the states required to verify the property, rather than performing an exhaustive state-space exploration. Each Promela process being represented as an automaton, we can represent the complete state-space generated by parallel processes by performing the product of these automata.

One major limitation is that `LTL` model-checking is PSPACE-complete. Reasoning about specific predicates to verify, allows limiting the state-space to the subset

1. See the Promela reference for equivalent `LTL` symbolism `http://spinroot.com/spin/Man/ltl.html`

required to verify the predicates. This is why Spin enhances state-space exploration with lossless compression techniques based on the characteristics of the claims validated. For instance, *Partial Order Reduction* [84] permits to merge states for which the partial order does not affect the property to verify.

7.3.2 Introduction to Parallel Algorithm Modeling

As an introductory example, let us consider the verification of the busy-waiting lock primitives, usually known as *spinlock*, present in the PowerPC and Intel architectures of Linux kernel 2.6.30.

The spinlock implementation found in the PowerPC architecture is relatively straightforward: it consists of two states, either 0 or non-zero. It uses the "`lwarx`" (Load Word and Reserve Indexed) and "`stwcx.`" (Store Word Conditional Indexed) instructions to atomically compare and update the lock value. The store only succeeds if the memory location has not been updated since the load.

As an example, a Promela model of this locking primitive is presented in Figure 7.1. Line 1 contains the lock variable definition, followed by a data access reference count defined in Line 2. Lines 4–17 contain the spin lock primitive. The `inline` function in Promela is close to that of the C language, except that such functions in Promela have type-free arguments, are not permitted to contain declarations, and do not return any value. Lines 6–16 contain the busy-waiting loop `do ... od`, stopped only by the `break` statement on Line 13 if the variable `lock` is 0. The `skip` statement on Line 10 is an empty statement. It has no effect other than permitting to follow the Promela grammar. Line 7 begins with "`:: 1 ->`", which indicates a condition always fulfilled. This lets the statements following the "`->`" execute unconditionally. Line 7 ends with a very important keyword: `atomic`. It precedes a sequence of statements, contained within brackets, which is considered as indivisible. They are therefore executed in a single execution step, similarly to an atomic instruction executed by a processor. Lines 19–22 contain the unlock function. Lines 24–33 contain the body of the processes, which takes a spinlock, increments and decrements the reference count, and releases the lock in an infinite loop. Two instances of the process are run upon initialization by `init` at Lines 35–39.

This Promela code is represented by the diagram found in Figure 7.2. Each node represents a Promela statement. A name is added to most nodes to make

```
 1 byte lock = 0;
 2 byte refcount = 0;
 3
 4 inline spin_lock(lock)
 5 {
 6   do
 7   :: 1 -> atomic {
 8       if
 9       :: lock ->
10           skip;
11       :: else ->
12           lock = 1;
13           break;
14       fi;
15     }
16   od;
17 }
18
19 inline spin_unlock(lock)
20 {
21   lock = 0;
22 }
23
24 proctype proc_X()
25 {
26   do
27   :: 1 ->
28        spin_lock(lock);
29        refcount = refcount + 1;
30        refcount = refcount - 1;
31        spin_unlock(lock);
32   od;
33 }
34
35 init
36 {
37   run proc_X();
38   run proc_X();
39 }
```

Figure 7.1 Promela model for PowerPC spinlock

interpretation easier. Each node contains a Promela statement. Arrows connecting the nodes represent how the model-checker can move between nodes. Some require conditions to be active, e.g. `(lock == 1)`, to allow moving to the target node. The `STEP++` statements on the arrows represent that the execution counter is incremented. A concurrent process may run between different steps, but not while `STEP` stays invariant. The latter scenario happens in the `ATOMIC` box, which represents the atomic sequence of statements.

Safety of this locking primitive is successfully verified by the Spin model-checker by verifying that the reference count value is never higher than 1. This is performed by prepending `#define refcount_gt_one (refcount > 1)` to the model and by using

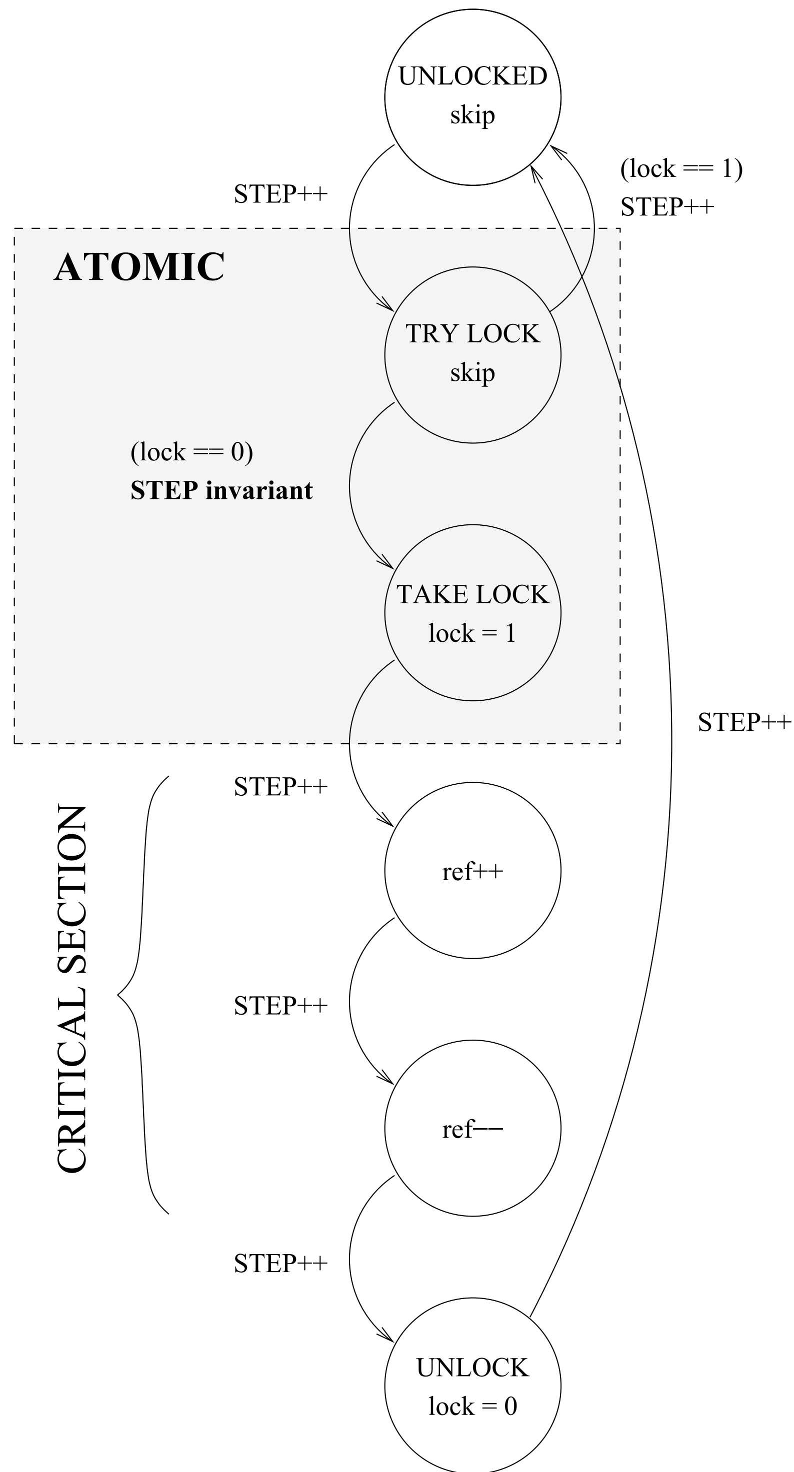

Figure 7.2 Diagram representation of PowerPC spinlock model

the following LTL formula. PowerPC spinlock safety is verified by the LTL claim:

$$\mathbf{G}\ \big(\neg\ \text{refcount_gt_one}\ \big)$$

However, one major downside of this spinlock implementation is its lacks of fairness. A CPU always acquiring the same spinlock in a loop could effectively starve other CPUs. This can be verified using the following LTL formula:

$$\mathbf{G}\ \big(\ \mathbf{F}\ (\neg\ \text{np_}\ )\big)$$

The keyword `np_` has a special meaning: it is true if all system states do not correspond to progress states. We modify the code from Figure 7.1 to insert such progress states: this involves separating the process body in two different definitions to add a `progress:` keyword within the infinite loop in one of them. Using the Spin verifier *weak fairness* option lets it detect non-progress cycles involving more than one process. This corresponds to starvation of a process by one or more other processes.

Ticket spinlocks used for the Intel spinlock implementation found in Linux correct the fairness problem identified in the PowerPC architecture. The Promela implementation is omitted due to space considerations, but the state diagram is presented at Figure 7.3. The new elements added to this graphs are the `LOW_HALF()` and `HIGH_HALF()` primitives, which select half lower and upper bits of the lock, respectively.

The Spin model-checker verifies that this model is safe and fair, under certain conditions. Changing the number of bits available for the low and high halves of the ticket lock as well as the number of processes shows that fairness is only ensured when the number of processes fits in the number of bits available for each half.

7.4 Weakly-Ordered Memory Framework

7.4.1 Architecture

Modeling out-of-order memory accesses performed by processors at a level consistent with their hardware implementation is important to enable accurate modeling of side-effects that can be caused by missing memory barriers in synchronization algorithms. The bugs within this category are hard to reproduce, mainly because they are dependent on the architecture, execution context and timing between the processors.

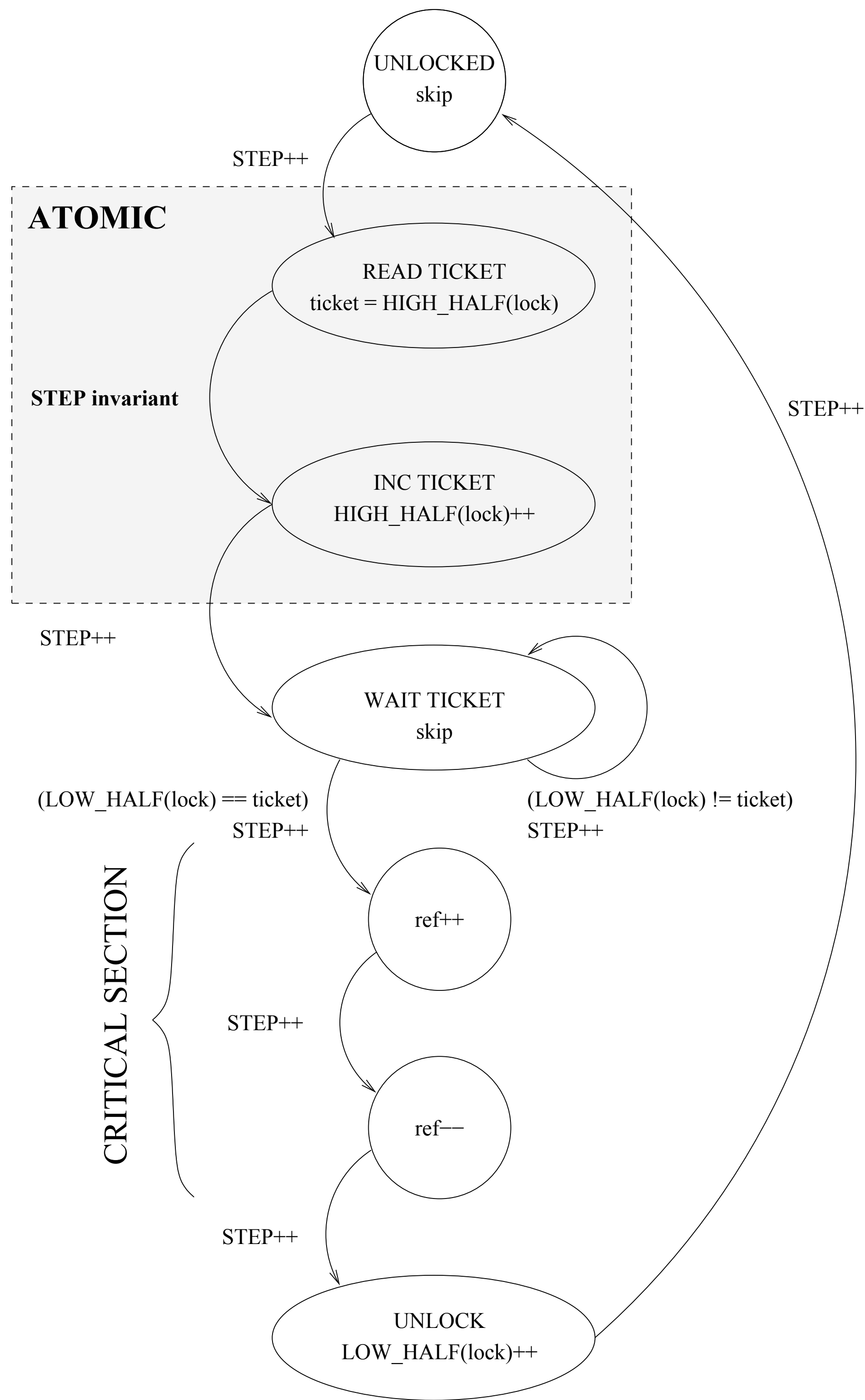

Figure 7.3 Diagram representation of Intel ticket spinlock model

Therefore, testing the implementation might not be sufficient to certify the absence of bugs.

To model an algorithm including the effects of the memory barriers required on various architectures, or more importantly lack thereof, we choose to create a virtual architecture performing the most aggressive memory reordering. The Alpha 21264 seems to be an especially interesting architecture with respect to reordering, given its ability to reorder dependent loads [85, 86]. It also reorders loads after stores, stores after loads, loads after loads and stores after stores. Even atomic operations can be reordered with respect to loads and stores.

The Alpha architecture can reorder dependent loads in addition to other memory accesses due to the design of its caches. These are divided into banks, which can each communicate with main memory. If the channel between a bank and memory is saturated, the updates of the less busy channels could reach their destination before loads or stores initiated earlier. Such extremely weak ordering can therefore cause any sequence of loads and stores to different addresses, including dependent loads, to be perceived in a different order from the point of view of another processor. Indeed, when memory content changes, in-cache view updates are not guaranteed to reach the cache-lines in the same order the main memory stores appear.

We name the weak memory ordering part of our virtual architecture *OoOmem*, where *OoO* stands for *Out-of-Order*. It models the exchanges between CPU cache-lines and main memory. To model the worst possible case, each variable belongs to a different cache-line and there is only one cache-line per bank. These cache-lines are therefore free to be updated (or not) with main memory between each instruction execution.

Each CPU is modeled as a Promela process. Each variable is represented with a main memory entry, a per-CPU entry and a per-CPU dirty flag. Operations and accesses to these memory entries are performed through Promela macros created as part of the *OoOmem* framework to facilitate manipulation of the variables. We call these the "primitives" of the *OoOmem* framework. For each cache-local variable, the `ooo_mem()` primitive models the case where it is updated as well as the case where it is not. This primitive must be called between each cache-line access. This causes all possible interleaving of exchanges between caches and memory to appear in the set of generated execution traces.

Explicit memory ordering of loads and stores can be respectively forced by using

the primitives `smp_rmb()` and `smp_wmb()`. These memory barriers respectively send the cache-local stores to memory and ensure that all in-memory data is loaded into the local cache. The per-variable, per-CPU dirty flag makes sure that a given CPU fetches data that it has recently written from its emulated cache rather than fetching stale data from main memory.

The primitive `write_cached_var()` updates the cache-local version of a variable with a new value and sets the dirty flag for this variable on the local CPU. The dirty flag will let `ooo_mem()` and `smp_wmb()` subsequently commit this change to main memory. In addition, this ensures that neither `ooo_mem()` nor `smp_rmb()` overwrite the cache-local version before it has been committed to memory.

The read-side equivalent is `read_cached_var()`, which loads a cache-local variable into the local process variables.

Modeling other architectures such as the Intel, PowerPC and Sparc processor families, which do not permit to reorder dependent loads, only requires replacing the conditional load part of the `ooo_mem()` primitive by a call to `smp_rmb()`. As a result, the local cache is unconditionally updated by loading all main memory variables into the non-dirty local cache-lines. The effects of independent loads reordering done by these architectures is modeled by the *Out-of-order Instruction Scheduling Model* presented in Section 7.5.

Modeling of nested execution contexts, such as interrupt handlers, requires that the Promela process used to represent the interrupt handler execution shares the per-CPU data with the Promela process modeling the interrupted processor.

7.4.2 Testing

In order to validate the accuracy of the framework, we use architectural litmus tests for which results are known. One such litmus test involves processor A performing consecutive updates to a pair of variables, while processor B concurrently reads these same variables in reverse order. We expect that if the ordering is correct, whenever processor B sees the updated second variable, it must eventually read the updated first variable. This is expressed in the following pseudo-code and `LTL` formula:

Pseudo-code modeled:

```
alpha = 0;
beta = 0;

Processor A                Processor B
alpha = 1;                 x = beta;
smp_wmb();                 smp_rmb();
beta = 1;                  y = alpha;
```

`LTL` claim to satisfy:

$$\mathbf{G}\ \big(x = 1 \Rightarrow \mathbf{F}\ (y = 1)\big)$$

This model is verified successfully by the Spin verifier. Error-injection is performed to ensure that the verifier would appropriately generate the erroneous execution traces if the required memory barriers were missing. This is performed by removing the `smp_wmb()` from Processor A or `smp_rmb()` from Processor B. Verifying these two altered models shows the expected errors and execution traces: variables being either stored to or loaded from memory in the wrong order fails to verify the `LTL` claim.

7.5 Out-of-Order Instruction Scheduling Framework

Although the *OoOmem* framework presented earlier represents exchanges between cache and memory accurately, it does not reproduce all reordering performed at the processor level regarding out-of-order micro-operation (`RISC` instruction) scheduling. This section explains how we model these effects.

7.5.1 Architecture

Superscalar pipelined architectures leverage instruction-level parallelism by allowing multiple instructions to start concurrently and by reordering instruction completion. It can, more aggressively, reorder the sequence in which independent instructions are issued. Speculative execution can also cause execution of instructions before their result is proven to be needed. Such out-of-order instruction execution can be seen on the Alpha 21264 [86].

Our virtual architecture framework for out-of-order instruction scheduling therefore encompasses all possible instruction scheduling which can be done by either compiler optimizations (lifting, combining reads, re-loading a variable to diminish register pressure, etc.) or the processor. The weakest scheduling possible is bounded by the dependencies between instructions. In order to let the verifier explore all possible execution orders, our virtual processor framework, *OoOisched*, provides:

- an infinite number of registers,
- a pipeline with an infinite number of stages,
- a superscalar architecture able to fetch and execute an infinite number of instructions concurrently,
- and the ability to perform speculative instruction execution when they have no side-effect on cache.

As in the *OoOmem* framework, one Promela process represents one processor. A key element of this framework is to have a compact instruction execution state and dependency representation. We choose to use a per-processor set of tokens to represent the dependencies with a single token per instruction. Tokens are produced by executing instructions and typically cleared only at the end of the instruction sequence. The conditions required to activate an instruction are represented by a set of tokens. Each token can be represented by a single bit.

As an example, the dependencies of the test-case presented in Section 7.5.2 are modeled in the Promela listing in Figure 7.4 and illustrated in Figure 7.5.

An instruction scheduling loop tests for every instructions dependency constraints to execute them. Execution of instruction is non-deterministic: when the dependencies of multiple instructions are met, any one of them may fire, but does not have to. It therefore explores all the possible execution orderings fitting within the dependency constraints. It proceeds until the end of the loop, which consists in executing the last instruction of the sequence. This last instruction clears all the tokens and breaks the instruction scheduling loop. When the bit allocated for an instruction is enabled, it inhibits its execution and enables its dependent instructions. The state of each CPU's execution is kept in a per-process data structure containing the current execution state tokens.

The macro `CONSUME_TOKENS(tokens, enable, inhibit)` is presented in the Promela model in Figure 7.4. It is used as trigger to execute an instruction. The parameter *tokens* is the token container of the current processor. The scheduler is allowed

to execute an instruction only if all the *enable* tokens are active and all the *inhibit* tokens are inactive. Its role is to check for pre-conditions for instruction execution, but does *not* clear any token.

The macro `PRODUCE_TOKENS(tokens, prod)` adds tokens identified by *prod* to *tokens*. It is typically used at the end an instruction execution by producing its own token. Finally, `CLEAR_TOKENS(tokens, clear)` clears all the specified CPU tokens. It is typically used after the last instruction of the scheduling loop, but can also be used to partially clear the token set to produce loops.

The diagram representing the Promela model in Figure 7.5 represents each instruction by a node. White arrows represent unmet dependencies and black arrows correspond to dependencies met. Colored nodes are those currently candidate for execution: all their dependencies are met, which means that all tokens they consume are enabled, and all the tokens that inhibit their execution are cleared. The column on the left represent the tokens associated with each instruction.

We choose this representation of the data and control flow rather than more classic token-based models like Petri networks [87] or coloured Petri networks [88] to allow easy injection of faults in the model. This would be cumbersome to do with a classic representation where one instruction would produce a token that would be later consumed by a following instruction. For instance, removing a read barrier or write barrier from the model would require to completely modify the dependency graph of the following instructions to make sure they now depend on prior instructions. Failure to do so would create an artificial synchronization barrier which would not model the error injection correctly. Since the token model provides the complete list of instruction dependencies, errors can be injected by enabling the token corresponding to the instructions to disable before entering the instruction scheduling loop. The effect is the inhibition of the instruction and satisfaction of all dependencies normally met when this instruction is executed.

Given our virtual architecture models all possible sequences of code execution allowed by the data dependencies, only a few specific issues must be addressed to make sure compiler optimizations are taken into account. In our framework, a temporary per-process variable, corresponding to a processor register, should never be updated concurrently by multiple instructions. `SSA` (Static Single Assignment) [89] is an intermediate representation typically used in compilers where each variable is assigned exactly once. Using such representation for registers would ensure to have no more

than a single instruction using a temporary register, but this would cost additional state-space. Given this resource is limited, we re-use registers outside of their liveness region. There are only two cases where we expect the compiler to re-use the result of loads. The first case is when the compiler is instructed to perform a single `volatile` access to load the variable to a register. The second case is when an explicit compiler barrier is added between the register assignment (load from cache) and register use. In all other cases, re-use of loaded variables will be taken into account by performing the two loads next to each other due to speculative execution (prefetching) support in the scheduler.

One limitation of this framework is that it adds an artificial compiler barrier and core synchronization between consecutive instruction scheduler executions. This would not take into account side-effects caused by scheduler execution within a loop. This is caused by the instruction's inability to cross the artificial synchronization generated by the last instruction executed at the end of the scheduler loop. This last instruction is required to clear all tokens before the next execution. Such effect can be modeled by unrolling the loop.

Because the token container is already occupied by the outermost execution of the scheduler, recursion is also not handled by the framework. A supplementary container could be used to model the nested execution. However, using a different instruction scheduler for the nested context would fail to appropriately model interleaving of instructions between different nesting levels. Therefore, nested calls must be expanded into the caller site.

7.5.2 Testing

Before introducing the more complex `RCU` model, we present a test model for the *OoOisched* framework. This model is based on both the *OoOisched* and *OoOmem* frameworks. It models an execution involving two processors and two memory locations. In this model, Processor A successively writes to `alpha` and reads `beta`. Processor B successively writes to `beta` and then reads `alpha`. This verifies that at least one processor reads the updated variable. This is shown in the following pseudo-code. Dependency constraints applied on instructions executed by Processor A are illustrated by Figure 7.5.

Pseudo-code modeled:

```
alpha = 0;
beta = 0;
x = 1;
y = 1;

Processor A              Processor B
alpha = 1;               beta = 1;
smp_mb();                smp_mb();
x = beta;                y = alpha;
```

LTL claim to satisfy:

$$\mathbf{G}\ (x = 1 \lor y = 1))$$

This model is successfully verified by the Spin verifier. Error-injection is performed to ensure that the verifier would appropriately generate the erroneous execution traces if the required memory barriers were missing. This is performed by either:

– completely removing the `smp_mb()`,
– removing only the `smp_rmb()` part of the barrier,
– removing only the `smp_wmb()` part of the barrier,
– removing the implicit core synchronization provided by the `smp_mb()` semantics, which leaves the reads and the writes free to be reordered.

Verifying these altered models shows the expected errors and execution traces, where the read or write instructions being reordered fails to verify the LTL claims.

Removing core synchronization from the model presented in Section 7.5.2 permits verifying its behavior when injecting errors. The diagram presented in Figure 7.6 shows a snapshot of instruction execution with core synchronization removed. It shows that two instructions are candidate for execution: either `alpha = 1` or `x = beta`. It this case, the store and load can be performed in any order by the instruction scheduler.

As an example of the result of an error-injection, we present an execution trace generated by the Spin model-checker. We use the test model presented in Figure 7.4 with core synchronization removed. This partial execution trace excludes the empty execution and `else`-statements for conciseness. Some statements are also folded.

Lines 6–8 show the instruction scheduler from processor B scheduling the two first instructions of this processor: production of the initial token and error-injection by

```
 1 #define PA_PROD_NONE    (1 << 0)
 2 #define PA_WRITE        (1 << 1)
 3 #define PA_WMB          (1 << 2)
 4 #define PA_SYNC_CORE    (1 << 3)
 5 #define PA_RMB          (1 << 4)
 6 #define PA_READ         (1 << 5)
 7
 8 byte pa_tokens;
 9
10 active proctype processor_A()
11 {
12   PRODUCE_TOKENS(pa_tokens, PA_PROD_NONE);
13
14   do
15   :: CONSUME_TOKENS(pa_tokens,
16                     PA_PROD_NONE, PA_WRITE) ->
17       ooo_mem();
18       WRITE_CACHED_VAR(alpha, 1);
19       ooo_mem();
20       PRODUCE_TOKENS(pa_tokens, PA_WRITE);
21   :: CONSUME_TOKENS(pa_tokens,
22                     PA_WRITE, PA_WMB) ->
23       smp_wmb();
24       PRODUCE_TOKENS(pa_tokens, PA_WMB);
25   :: CONSUME_TOKENS(pa_tokens,
26                     PA_WRITE | PA_WMB,
27                     PA_SYNC_CORE) ->
28       PRODUCE_TOKENS(pa_tokens, PA_SYNC_CORE);
29   :: CONSUME_TOKENS(pa_tokens,
30                     PA_SYNC_CORE, PA_RMB) ->
31       smp_rmb();
32       PRODUCE_TOKENS(pa_tokens, PA_RMB);
33   :: CONSUME_TOKENS(pa_tokens,
34                     PA_SYNC_CORE | PA_RMB,
35                     PA_READ) ->
36       ooo_mem();
37       pa_read = READ_CACHED_VAR(beta);
38       ooo_mem();
39       PRODUCE_TOKENS(pa_tokens, PA_READ);
40   :: CONSUME_TOKENS(pa_tokens,
41                     PA_PROD_NONE | PA_WRITE |
42                     PA_WMB | PA_SYNC_CORE |
43                     PA_RMB | PA_READ, 0) ->
44       CLEAR_TOKENS(pa_tokens,
45                    PA_PROD_NONE | PA_WRITE |
46                    PA_WMB | PA_SYNC_CORE |
47                    PA_RMB | PA_READ);
48       break;
49   od;
50 }
```

Figure 7.4 Out-of-order instruction scheduling and memory frameworks promela test code, Processor A

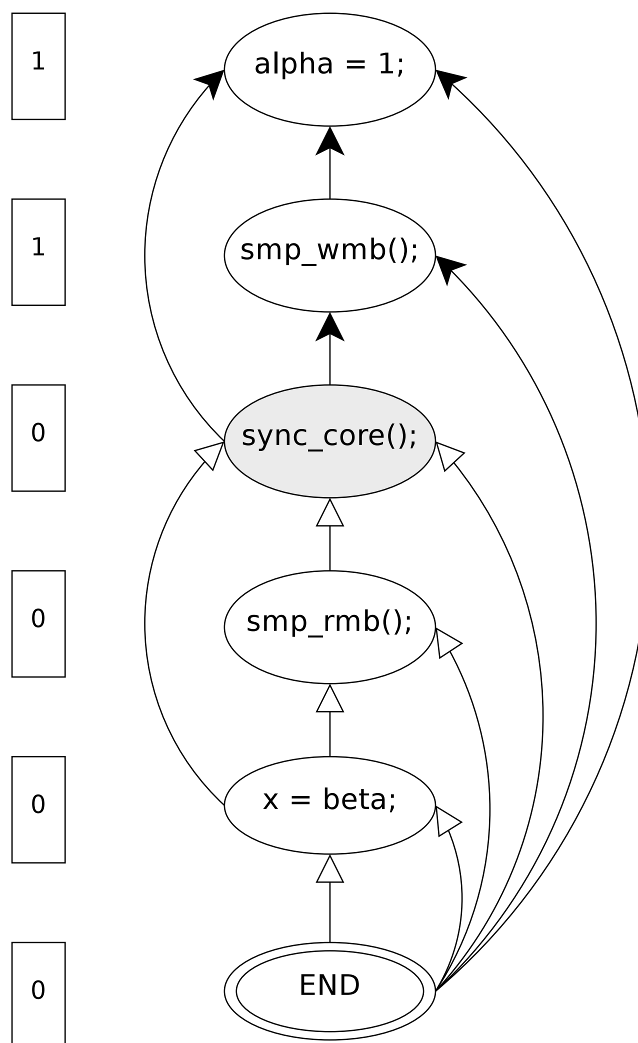

Figure 7.5 Instruction dependencies of out-of-order instruction scheduling and memory framework test

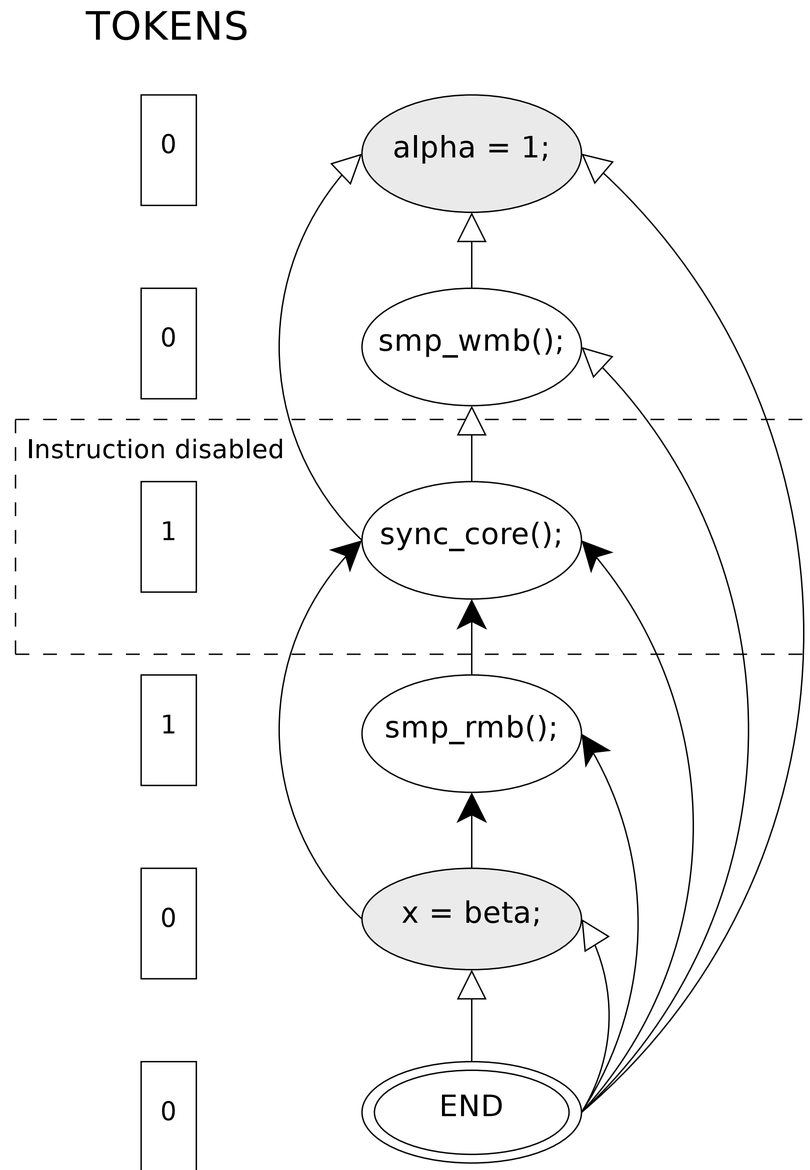

Figure 7.6 Instruction dependencies of out-of-order instruction scheduling and memory framework test (error injection)

producing the core synchronization token ahead of the instruction scheduler execution.

```
 6:    proc  1 (CPU_B) line 252 "mem.spin" (state 2)
              [pb_tokens = (pb_tokens|(1<<PA_PROD_NONE))]
 8:    proc  1 (CPU_B) line 261 "mem.spin" (state 3)
              [pb_tokens = (pb_tokens|(1<<PA_SYNC_CORE))]
```

Line 10 presents the trigger permitting activation of the write instruction.

```
10:    proc  1 (CPU_B) line 265 "mem.spin" (state 4)
              [((!((pb_tokens&(1<<PA_WRITE)))
               && ((pb_tokens&(1<<PA_PROD_NONE))
                   ==(1<<PA_PROD_NONE))))]
```

Lines 19–21 show processor B updating its alpha and beta cache-view from memory. Processor B performs a random cache update. In this execution trail, it loads alpha and beta into its local cache.

```
19:    proc  1 (CPU_B) line 125 "mem.spin" (state 30)
              [(!(cache_dirty_alpha[_pid]))]
19:    proc  1 (CPU_B) line 125 "mem.spin" (state 31)
               [cached_alpha[_pid] = mem_alpha]
21:    proc  1 (CPU_B) line 126 "mem.spin" (state 41)
               [(!(cache_dirty_beta[_pid]))]
21:    proc  1 (CPU_B) line 126 "mem.spin" (state 42)
               [cached_beta[_pid] = mem_beta]
```

At Line 23, processor B writes to its cache view of beta and sets the matching dirty flag.

```
23:    proc  1 (CPU_B) line 267 "mem.spin" (state 53)
               [cached_beta[_pid] = 1]
23:    proc  1 (CPU_B) line 267 "mem.spin" (state 54)
               [cache_dirty_beta[_pid] = 1]
```

A succinct high-level summary of the execution trail follows:

```
CPU B: writes to in-cache beta
CPU A: writes to in-cache alpha
CPU A: smp_rmb()
CPU B: smp_wmb()
CPU B: smp_rmb()
CPU B: reads alpha from its cache
CPU A: smp_wmb()
CPU A: reads beta from its cache
```

At the bottom of the execution trail, the state for which the LTL condition did not hold is shown:

```
spin: trail ends after 161 steps
#processes: 1
                mem_alpha = 1
                cached_alpha[0] = 1
                cached_alpha[1] = 0
                cache_dirty_alpha[0] = 0
                cache_dirty_alpha[1] = 0
                mem_beta = 1
                cached_beta[0] = 1
                cached_beta[1] = 1
                cache_dirty_beta[0] = 0
                cache_dirty_beta[1] = 0
                x = 0
                y = 0
                pa_tokens = 31
                pb_tokens = 63
161:    proc  0 (CPU_A) line 218 "mem.spin" (state 239)
2 processes created
```

At that point, both `pa_read` and `pb_read` contain 0. By examining the execution trace, we understand that this behavior is made possible by letting processor A execute its read memory barrier before the write memory barrier. This is allowed because the removed core synchronization permits reordering these unrelated types of barriers.

Therefore, even given the known model limitations regarding loops and nesting, the instruction scheduling and weakly-ordered memory architecture models are sufficient to model RCU algorithms, as is shown in Section 7.6.

7.6 Read-Copy Update Algorithm Modeling

Read-Copy Update (RCU) is a synchronization primitive allowing multiple readers of a data structure to execute concurrently with extremely low-overhead [3]. Its main characteristic is to provide linear read-side scalability as the number of processor increases. It performs this by allowing multiple copies of a data structure to exist at the same time. In a period of time called *grace period*, each processor is allowed to see a different copy of the data structure. The RCU synchronization guarantees specify a lower bound to the duration of the grace period, after which no further references to old copies exist, so that the underlying memory becomes reclaimable.

The main motivation for validating the RCU algorithms is their complexity level. These algorithms are parallel and imply inconsistent views between processors at a specific point in time. Also, because RCU's read-side primitives contain no standard mutual exclusion primitives, memory ordering must be performed by these same RCU primitives.

This section presents the model we created to verify if the grace-period and publication guarantees are satisfied by two RCU synchronization algorithms proposed in paper [3]: *General-Purpose RCU* and *Low-Overhead RCU Via Signal Handling*. It also verifies that both updater and readers always progress. We first describe the general RCU model, which is subsequently derived into a signal-based memory barrier model. We choose the Promela language to express the model based on prior successful modeling of low-level synchronization primitives with this language [73, 74]. Our motivation for using the Spin model-checker comes mainly from its level of maturity (it is available freely since 1991) and stability. Using the Promela syntax, which is close to C, makes it straightforward for C programmers translate C code into models. The model code uses precompiler directives to select architecture behavior and to perform error injection. The model consists of 1300 lines of Promela code.

A schematic for the high-level structure of an RCU-based algorithm is shown in Figure 7.7. An RCU grace period is informally defined as any time period such that all RCU read-side critical sections in existence at the beginning of that period have completed before its end.

Here, each box labeled "Reads" is an RCU read-side critical section that begins with `rcu_read_lock()` and ends with `rcu_read_unlock()`. Each row of RCU read-side critical sections denotes a separate thread, for a total of four read-side threads. The two boxes at the bottom left and right of the figure denote a fifth thread, this one performing an RCU update.

This RCU update is split into two phases, a removal phase denoted by the lower left-hand box and a reclamation phase denoted by the lower right-hand box. These two phases must be separated by a grace period, which is determined by the duration of the `synchronize_rcu()` execution. During the removal phase, the RCU update removes elements from the data structure (possibly inserting some as well) by issuing an `rcu_assign_pointer()` or equivalent pointer-replacement primitive. These removed data elements will not be accessible to RCU read-side critical sections starting after the removal phase ends, but might still be accessed by RCU read-side critical sections

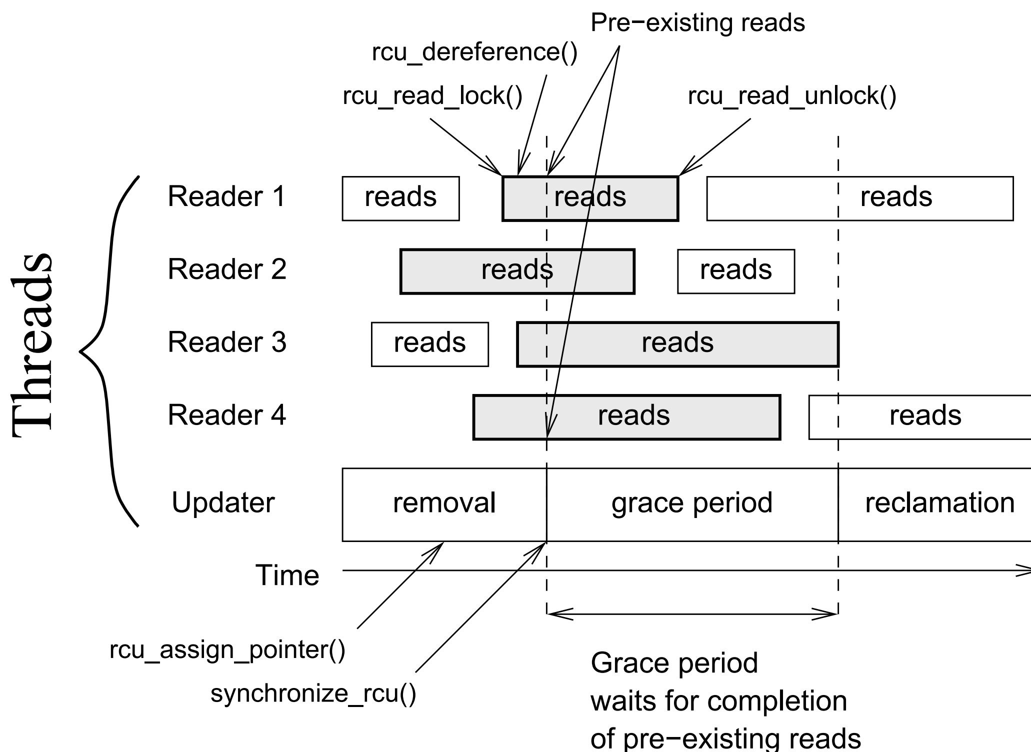

Figure 7.7 Schematic of RCU grace period and read side critical sections

initiated during the removal phase. However, by the end of the `RCU` grace period, all of the `RCU` read-side critical sections that might be accessing the newly removed data elements are guaranteed to have completed, courtesy of the definition of *grace period*. Therefore, the `grace-period guarantee` ensures that the reclamation phase, beginning after the grace period ends, can safely free the data elements removed previously.

The *publication guarantee* ensures that data accessed by the read-side through the `rcu_dereference()` primitive (always executed between `rcu_read_lock()` and `rcu_read_unlock()`) will see changes made by the write-side before publication of the `RCU` pointer by `rcu_assign_pointer`.

The model which abstracts the `RCU` algorithm has one updater and one reader process. It is based on the *OoOmem* and *OoOisched* frameworks to verify that it satisfies the above-mentioned grace-period and publication guarantees when executed on a weakly-ordered processor and memory architecture. In addition to global and per-thread `RCU` synchronization variables, the data structures required are a pointer

to the current `RCU` data and an *arena*: a pool of free memory used by the memory allocator. All these data structures are modeled with the *OoOmem* framework. The *arena* data is initially poisoned.

The updater process performs two loops which first update data in a newly allocated arena entry from the *OoOmem* model and then publishes a pointer to the new entry into another shared *OoOmem* variable using a modeled `rcu_assign_pointer()` primitive. At the same time the updater stores the new pointer, the updater loads the previous value into its local registers. It is reclaimed after a grace period passes. The modeled `synchronize_rcu()` primitive is used to wait for such grace period to reach a quiescent state. After that point, the arena entry corresponding to the old pointer can be poisoned.

Memory reclamation is modeled by writing *poison* data into the arena entry. In a valid model, the read-side should never see such poison value in the memory location it reads. This method is used to detect incorrect publication and inappropriate use of reclaimed memory.

The updater loops do not need to be unrolled because the `synchronize_rcu()` primitive contains full memory barriers. Poisoning alone could spill in the next loop and overlap with stores to the newly allocated arena entry, but late-arriving poisoning stores are not relevant to the characteristics we validate.

The reader is modeled as one process entering two read-side critical sections. Each consists of a lock and unlock pair, between which are placed a `rcu_dereference()` and a read of the corresponding arena entry. The first critical section holds two nested read locks. The second critical section holds a single nesting-level read lock. Given the outermost code of successive read lock/unlock can spill on each other, such spilling caused by reordering, prefetching and optimizations is modeled by those two successive critical sections. The data read within each critical section is saved to the globally visible `data_read_first` and `data_read_second` variables to be used for verification.

The guarantees provided by the `RCU` algorithm are verified with the `LTL` formula presented in (7.1), which makes sure the reader never loads a poisoned arena entry. The Spin model-checker checks for states which do not verify this claim. It generates an error and presents a counter-example consisting of a faulty execution trace when the claim is not satisfied.

$$\mathbf{G}\left(\begin{array}{c} data_read_first \neq POISON \\ \wedge \\ data_read_second \neq POISON \end{array}\right) \tag{7.1}$$

To minimize the risk of modeling errors, we augment our models with error-injection regression tests. For each of the characteristic we need to validate, we create a model alteration which is known not to satisfy the characteristic. This permits to not only verify that the guarantees are sufficient to ensure model correctness, but also that the modeled guarantees are actually required, e.g. if a predicate is not satisfied, an error will occur.

Remove `smp_mb()`: The first error injected is to remove the memory barriers surrounding `synchronize_rcu()`. This is known not to meet the grace period guarantee, as it would let the pointer update spill over the whole grace period into the following quiescent state. This would therefore let poisoning occur before the pointer is updated.

Remove `smp_wmb()`: The second error we inject is to remove the write memory-barrier from the `rcu_assign_pointer()` primitive. This is known not to meet the publication guarantee, because the pointer could then be published before the newly allocated arena entry is populated with non-poisoned information.

Remove `smp_rmb()`: The third error is injected by removing the read memory-barrier from the `rcu_dereference()` primitive. On Alpha (and *only* on Alpha), this is known not to meet the publication guarantee because the reader's cache could be populated with the new pointer before the arena entry is updated. We therefore expect the reader to see poisoned data.

Single grace-period phase: The fourth error-injection test consists in altering `synchronize_rcu()` to only perform a single grace-period phase. This is expected not to meet the grace period guarantee by allowing a race condition between a reader and two consecutive updates.

The reader code is modeled in an infinite loop to verify updater's progress when facing a steady flow of readers. The reader and updater progress are tested in two different runs:

- For reader progress, a single progress statement is added between each reader loop execution.
- On the updater-side, progress statements are added in each update loop and an

infinite loop containing a progress statement is added at the end of the updater's process execution.

The weak fairness Spin option ensures that non-progress errors are flagged only for cycles containing at least one statement from each process, but not containing any special *progress* labels.

Remove `smp_mb()` from busy-loop: The fifth error injected is to remove the memory barrier placed in the updater's busy loop waiting for a grace period phase to complete. This injects an updater progress error by allowing the updater's cache to never read the eventually updated reader nesting counter. On real systems, the bounded size of the buffers between the CPU, cache and memory interconnects ensures that the remote nesting counter is updated, but given our virtual architecture model assumes infinitely-sized buffers, an explicit memory barrier must be placed in the busy-loops to ensure data is being read.

The signal-based memory barrier is modeled as a derivation of the general model by changing the updater-side memory barriers for a primitive which sends a signal and waits for memory barrier execution from the reader-side, all this between two memory barriers. On the read-side, the memory barriers are modeled by verifying, between each instruction execution, if the execution status tokens appears to be in sequential execution order. If execution appears in sequential order between two instructions, the reader process chooses to either ignore any memory barrier request or to service any number of memory barrier requests by issuing memory barriers and informing the updater-side of the completion.

Due to the added complexity and therefore state-space size explosion, modeling of read-side in signal handers nesting over the updater and reader thread is performed using a different model which assumes a sequentially ordered architecture with the *OoOmem* weakly-ordered memory framework. Given that signal handlers have the property to order the core execution before and after they execute, it allows using the safety and progress characteristics proven with the *OoOisched* framework as lemma.

7.6.1 State-Space Compression

Given that the state-space required to perform the verification can increase quickly, the following state-space compression techniques were used to perform the verifications.

Running the Spin model-checker to verify specific LTL formulas transformed into *never claims* permits checking for safety while performing *Partial Order Reduction* [84]. This model-checking approach discards relative statement ordering which does not matter for the property to verify. This reduces the state-space size tremendously with a very small performance impact, while preserving the safety and liveness properties of LTL.

Accepting a small performance impact (perceived slowdown of a factor 1.3 on our models), the COLLAPSE compression [82] can be used to reduce the state-space required to perform verification by separating the state into sub-components. The compression comes from the fact that one state configuration for a specific process tends to reoccur in different global data and other process states. It uses separate descriptors as key to encode and search global data objects and data objects belonging to each process. Each time the same process state is encountered, it can be encoded with the same per-process state descriptor instead of saving the whole state, which saves precious state-space. A global state descriptor, used to identify the overall state, therefore consists of a state vector made of the global and per-process descriptors. This lossless compression preserves the complete state-space.

Another possible lossless compression technique, the minimized DFA (Deterministic Finite Automaton) encoding [82], can further diminish the state-space size by leveraging the high degree of similarity between the different states. It represents the state-space using an encoding similar to BDDs (Binary Decision Diagrams) [90, 91]. However, this compression technique incurs a prohibitive performance impact. Our tests on large models show that state-space exploration is about 10 times slower. Given that the non-compressed execution of some verifications already takes about 24 hours, the computation time required for DFA compression is considered to be beyond our available computation time resources.

7.6.2 RCU Model-Checking Results

This section presents the Spin model-checker results for three models: the general purpose user-space RCU model, the signal-based RCU model and the modeling of signal-handler read-side. Test run results are presented along with the resources required to perform the verification. These verifications are performed on a Intel Core2 Xeon 2.0 GHz with 16 GiB RAM.

The only test result we really care about is whether the verification succeeds or fails. For the unaltered model and for progress verifications, the `LTL` claim or progress property are expected to hold. Such successful verification are denoted as `PASS`. For each error-injection run presented in Section 7.6, the expected result is that the model-checker should detect the injected error, denoted as `INJECT`. The notation `FNEG` would indicate that the model-checker was blind to an injected error. This constitutes a “false negative”. These are not errors as such, as not every bug necessarily results in a failure on every architectures modeled. Finally, the notation `FAIL` indicates that the model-checker detected a bug in the algorithm.

Additional information about the time and memory required to run these verifications is only provided to show the amount of computational resources needed for such verification. The only requirement is that execution time and memory used fit within our available resource limits.

Tables 7.1 and 7.2 present the result of the general-purpose `RCU` model-checking using the Alpha and Intel/PowerPC virtual architectures, respectively. The safety and progress verifications are successful, and all error-injections generate expected errors, except one: on the Intel/PowerPC architecture, no error is generated when removing the `smp_rmb()` on the read-side. This shows that no read barrier is required on these architectures due to the fact that dependent loads are not reordered.

Table 7.1 General-purpose `RCU` verification results for the Alpha architecture

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GiB)	Time
Unaltered model (safety)	PASS	1.06	2h33m
Remove `smp_mb()`	INJECT	1.07	1h06m
Remove `smp_wmb()`	INJECT	0.96	1h14m
Remove `smp_rmb()`	INJECT	0.52	9m
Single grace-period phase	INJECT	0.66	26m
Reader progress	PASS	1.76	10h38m
Updater progress	PASS	1.76	9h23m
Remove loop `smp_mb()`	INJECT	0.47	1m

Table 7.3 presents the verification result of the signal-based `RCU` model for the Alpha virtual architecture. This verification ensures signal-based memory barriers

Table 7.2 General-purpose RCU verification results for the Intel/PowerPC architectures

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GiB)	Time
Unaltered model (safety)	PASS	0.82	4m
Remove `smp_mb()`	INJECT	1.96	16m
Remove `smp_wmb()`	INJECT	0.79	5m
Remove `smp_rmb()`	FNEG	0.82	6m
Single grace-period phase	INJECT	0.61	1m
Reader progress	PASS	1.28	23m
Updater progress	PASS	1.28	23m
Remove loop `smp_mb()`	INJECT	0.47	0m

provide the memory ordering guarantees and that no livelock nor deadlock can occur. Progress verification requires to use the `COLLAPSE` Spin option to compress the state-space size. It takes 3.5 days to complete the updater progress verification. To reduce the required CPU time, the reader progress and the updater progress error-injection are performed on a simplified read-side model with only a single, non-nested, critical section. Updater progress has also been verified using this simpler model and resulted in a successful progress verification.

Table 7.3 Signal-based RCU verification results for the Alpha architecture

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GiB)	Time
Unaltered model (safety)	PASS	5.21	1d14h18m
Remove `smp_mb()`	INJECT	0.59	13m
Remove `smp_wmb()`	INJECT	5.17	1d14h33m
Remove `smp_rmb()`	INJECT	1.20	2h43m
Single grace-period phase	INJECT	3.09	5h45m
Reader progress	PASS	0.89	2h02m
Updater progress	PASS	11.73	4d15h13m
Remove loop `smp_mb()`	INJECT	0.472	1m

As in the Alpha signal-based RCU verification, Intel/PowerPC model verification require to use the COLLAPSE compression to fit in the available memory. Here we notice that the test execution time for each progress verification is approximately 4 hours and uses about 10 GiB of memory. As in the general purpose RCU model, the smp_rmb() removal does not cause any error on Intel/PowerPC because the architecture does not reorder dependent loads.

Table 7.4 Signal-based RCU verification results for the Intel/PowerPC architectures

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GiB)	Time
Unaltered model (safety)	PASS	6.98	1h43m
Remove smp_mb()	INJECT	1.18	4m
Remove smp_wmb()	INJECT	6.98	1h42m
Remove smp_rmb()	FNEG	6.98	1h45m
Single grace-period phase	INJECT	4.28	16m
Reader progress	PASS	10.08	4h18m
Updater progress	PASS	9.88	4h18m
Remove loop smp_mb()	INJECT	0.58	3m

Modeling of read-side signal handler nested over a reader thread is presented in Table 7.5. This model executes a read-side critical section in a signal handler interrupting a reader thread. We proceed to this verification to model a read-side critical section in a signal handler, which generates execution traces where a nested signal handler could deadlock with an interrupted process, if the signal handler would busy-loop waiting for the interrupted process.

Table 7.6 is the results obtained by modeling an interrupting read-side signal handler critical section nested over the updater thread. It presents an interesting result: given all read-side critical sections are contained within signal handlers nested over the updater, no memory barrier is required to ensure correctness because no cache-line exchange is required. In fact, only a single process is executing.

For each of the unaltered models checked, model coverage includes all of the RCU model lines, but excludes some *OoOmem* model operations which are not useful in some contexts. For instance, the *OoOmem* “random” store to memory will never be

Table 7.5 General-purpose RCU signal-handler reader nested over reader verification (no instruction scheduling)

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GiB)	Time
Unaltered model (safety)	PASS	4.35	10m
Remove `smp_mb()`	INJECT	1.60	5m
Remove `smp_wmb()`	INJECT	0.78	2m
Remove `smp_rmb()`	INJECT	1.60	2m
Single grace-period phase	INJECT	0.57	0m
Reader progress	PASS	9.21	1h56m
Updater progress	PASS	9.15	1h03m
Remove loop `smp_mb()`	INJECT	0.51	0m

Table 7.6 General-purpose RCU signal-handler reader nested over updater verification (no instruction scheduling)

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GiB)	Time
Unaltered model (safety)	PASS	0.47	0m
Remove `smp_mb()`	FNEG	0.48	1m
Remove `smp_wmb()`	FNEG	0.47	1m
Remove `smp_rmb()`	FNEG	0.47	0m
Single grace-period phase	FNEG	0.47	0m
Reader progress	PASS	0.47	1m
Updater progress	PASS	0.47	1m
Remove loop `smp_mb()`	FNEG	0.47	1m

executed if a process never writes into a given variable. Error injection runs do not need to visit all the state space because they stop after the first error encountered. Therefore, these self-testing runs do not need to provide complete coverage.

7.6.3 RCU Verification Discussion

Results presented in Section 7.6.2 demonstrate that we were able to successfully verify the `RCU` algorithm models in various execution scenarios with affordable computation resources. The error-injection tests further demonstrate that the model is able to detect defects that do not respect the `RCU` guarantees.

In these tests, the number of updater has been limited to one given we protect updater critical sections using a mutual exclusion primitive already expected to be valid. The number of reader is also fixed to one because the updater waits, in turn, for each reader one after the other. The algorithm therefore does not contain any reader-reader data or control dependency.

As expected, model of the read-side signal handler nested over a `RCU` reader succeeds because the `RCU` read-side is executed with $O(1)$ computational complexity, which implies that it never busy-loops.

The simplified read-side in signal handler model does not perform instruction execution reordering. Given the proof provided by the previous verifications, the nested signal handler execution can be modeled as being serialized with the rest of the interrupted code because the operating system is called before and after the signal handler. The *OoOmem* model is however still used to appropriately take the out-of-order memory effects into account. This models the Alpha virtual architecture, which is a superset of the Intel/PowerPC virtual architecture, given it allows weaker memory ordering.

Verification of interrupting read-side signal handler critical section nested over the updater thread interestingly shows that the read-side signal handler can nest over the updater without causing progress error (no livelock nor deadlock). The single grace-period phase test shows no error. This can be explained by the fact that the execution trace which requires two grace-period phases involves the reader seeing two updater updates. This execution trace is impossible here because the updater is being interrupted by the nested read-side signal.

Error-injection tests have been very useful to ensure model completeness. For instance, trying the test-case presented in Section 7.5.2 on the *OoOmem* model showed its limitations. Changing the `smp_mb()` into consecutive `smp_rmb()` and `smp_wmb()` (which are free to be reordered) did not produce the expected error. This showed that we needed to model out-of-order instruction scheduling to properly represent this class of CPU instruction reordering effects, effectively leading to the creation of

the *OoOisched* model.

Another example where error-injection has been useful happened during the *OoOisched*-based RCU model creation. The *OoOisched* framework being based on an instruction scheduling loop, we can only use the model coverage information provided by Spin as indication that statements have been reached at least once, but it tells nothing about the execution orders visited. Instruction dependency implementation errors, which inhibited execution of some instructions incorrectly, were identified with the help of these error-injection tests.

We also created a model for uniprocessor execution of the RCU algorithm. The code generated for this model has the particularity that all memory barriers are replaced by compiler barriers, except `smp_read_barrier_depends()`, on Alpha, which is completely removed. In this model, a single processor cache is used by both the reader and the writer processes. No communication is required with main memory, given all accesses are going through the locally cached variables. Therefore, out-of-order memory updates are disabled. The results of the tests, not presented here for conciseness, show that simply using compiler barriers suffice to provide RCU safe against thread preemption on a uniprocessor system.

7.7 Framework Discussion

Compared to models used previously for RCU verification, the proposed framework covers more micro-architecture side-effects. This includes, for instance, effects of data prefetch. Moreover, the state-space size required by our framework has been shown to be manageable on current computers when modeling complex synchronization algorithms such as RCU. This shows that it should be applicable to other parallel algorithms with similar complexity level.

One of the major improvements of this modeling framework is to allow a more regular description of algorithms. It removes the need to account for low-level architecture side-effects directly in the algorithm model by providing artefacts which encapsulate the architecture behavior. This framework therefore minimizes the risk of modeling error.

Due to its ability to model the weakest ordering possible, altering the framework to model memory barriers specific to architectures such as Alpha, Intel, PowerPC and even Sparc is straightforward. Modeling specific architectures can by done by creating

the synchronization instructions implemented in a given architecture and modifying the behavior of the cache-memory synchronization to match the architecture behavior. For instance, the PowerPC "`lwsync`" instruction [2] can be modeled as two instructions. The first instruction needed is a `smp_rmb()` which depend on all prior loads, and upon which depends all following loads and stores. The second instruction is a `smp_wmb()`, which depends on all prior stores, but upon which only the following stores depend. Such flexibility in modeling the low-level synchronization primitives become very handy to model the Sparc "`membar`" primitive, which permits to only order either, some or all of:

- stores vs stores,
- stores vs loads,
- loads vs loads or
- loads vs stores.

In the case of the `RCU` algorithm model, we only need full memory barriers.

We are aware of one recently proposed compiler optimization not handled by our model. Value-speculative optimizations [92, 93] performed by the compiler could cause dependent loads to be performed out-of-order if the first data to read is speculated, which would permit to read dependent data in the wrong order. These dependency-breaking optimizations are outside of the proposed model scope. Work in progress for upcoming versions of the C++ standard include compiler mechanisms designed to selectively suppress value speculation [94, 95, 96].

7.8 Conclusion

To accurately model the low-level multiprocessor interactions at the architecture-level, we created a virtual architecture performing the most aggressive optimizations still meeting the instruction inter-dependencies. Memory access ordering is expressed by modeling a processor cache with extremely weak ordering. A model of instruction dependencies deals with the effects of out-of-order instruction execution.

Formal verification of both general-purpose `RCU` and signal-based `RCU` has been performed on this virtual architecture, therefore modeling the effects of out-of-order

2. `lwsync` - Lightweight synchronization: Orders loads with respect to subsequent loads and stores. Orders stores with respect to other stores. Does not order stores with respect to subsequent loads.

instruction execution and out-of-order memory accesses. The high complexity-level of these RCU algorithms caused by the high degree of parallelism and extremely relaxed consistency semantics can easily overwhelm human conception. This is why validation at the lowest level of interprocessor interaction is needed to certify that these algorithms perform the expected synchronization.

Future work in this area could involve modeling value-speculative compiler optimizations, to enable detection of ordering problems which can occur when dependent memory accesses can be reordered dependency-breaking compiler optimizations.

Modeling these algorithms on this virtual architecture lets us demonstrate that all invocations of this algorithm primitives will behave appropriately and that porting it to yet unforeseen architectures will work as expected.

Acknowledgements

We owe thanks to Nicolas Gorse, Etienne Bergeron and Alexandre Desnoyers for reviewing this paper, to Maged Michael and Alan Stern for many illuminating discussions, and to Kathy Bennett for her support of this effort.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851. This work is funded by Google, Natural Sciences and Engineering Research Council of Canada, Ericsson and Defence Research and Development Canada.

Legal Statement

This work represents the views of the authors and does not necessarily represent the view of Ecole Polytechnique de Montreal or IBM.

Other company, product, and service names may be trademarks or service marks of others.

Chapter 8

Complementary Results

This chapter presents supplementary research results not discussed in the core thesis articles. This includes discussion of the work performed on *Kernel Markers*, *Tracepoints* and *Immediate Values*, as well as presentation of *latency* benchmarks, *real-time* determinism impact discussion and *formal verification* of the `LTTng` tracer.

8.1 Kernel Markers

The article "`LTTng`: Tracing across execution layers, from the Hypervisor to user-space" [28] refers to *Linux Kernel Markers*, created as part of this research, and now integrated in the mainline Linux kernel.

The initial motivation to create this static instrumentation infrastructure is because the `Kprobe` mechanism, a predating dynamic instrumentation mechanism, adds a large performance overhead because it depends on breakpoints. Section 5.5.8 presented the benchmarks justifying the choice of static over dynamic instrumentation, mainly due to its lower overhead.

Comparing static instrumentation with `Kprobe`-based instrumentation found in SystemTAP also provides a second motivation to use static instrumentation: its ability to follow more easily source code changes in an open source project like the Linux kernel.

This instrumentation mechanism allows us to enable the instrumentation of the Linux kernel at the source-code level. It consists essentially of a C preprocessing macro which adds, in the instrumented function, a branch over a function call. By doing so, neither the stack setup nor the function call are executed when instrumentation is not enabled. At runtime, each marker can be individually enabled, which makes the branch execute both the stack setup and the function call.

The design of this infrastructure is voluntarily biased to minimize the performance overhead when tracing is disabled. We give a hint to the compiler to position the

instructions executed only when tracing is enabled away from cache lines involved in standard kernel execution by identifying the branch executing stack setup and function call as unlikely (using the gcc `__builtin_expect()`).

The performance impact of the marker mechanism is small, albeit globally hard to notice with performance benchmarks. Most of it comes from the added branch, although the added register pressure, and transforming some leaf functions into non-leaf functions due to added function calls, are also adding to performance overhead.

A drawback of the Linux Kernel Markers is that it limits type verification to scalar types due because its `API` is based on format strings. However, this allow us to easily add new markers to source-code by modifying a single line. This limited type-checking, can be problematic if pointers must be dereferenced by the tracer code. For instance, a pointer to a structure could be passed as parameter, with the intent of letting the probe access specific fields in this structure. However, the Linux Kernel Markers only permit us to export a universal pointer (`void` pointer), leaving any specific type-checking impossible.

A second issue is that the Markers hide the instrumentation in the source code, keeping no global registry of the instrumentation. It is thus hard to impose namespace conventions and to keep track of instrumentation modification without monitoring the whole kernel tree.

8.2 Tracepoints

After experimenting with the Linux Kernel Markers, we decided to correct the two downsides of this infrastructure. Hence, we created Tracepoints to deal with this problem. They are extensively based on the Linux Kernel Markers code, with major modifications performed to support full type checking. The Tracepoints are now integrated in the Linux kernel and already used extensively.

The main difference between Tracepoints and Kernel Markers is that tracepoints require an instrumentation declaration in a global header. It allows type-aware verification of the tracer probes (callbacks) connected to the instrumentation site by declaring both the instrumentation call and the probe registration and unregistration function within the same declaration macro, which is aware of the types expected. It thus ensures that both the caller and the callee types will match. For example, we have the following tracepoint declaration in a global header to instrument scheduler

activity:

```
#include <linux/tracepoint.h>

DECLARE_TRACE(sched_switch,
        TP_PROTO(struct rq *rq, struct task_struct *prev,
                 struct task_struct *next),
                TP_ARGS(rq, prev, next));
```

Used to instrument the context switch function:

```
DEFINE_TRACE(sched_switch);

static inline void
context_switch(struct rq *rq, struct task_struct *prev,
                    struct task_struct *next)
{
        [...]
        trace_sched_switch(rq, prev, next);
        [...]
}
```

It also provides the needed global instrumentation registry: all global tracepoint declarations are kept in the `include/trace/` directory of the Linux kernel tree.

The rest of the infrastructure is similar to Linux Kernel Markers. The commit log of Tracepoints merge into the Linux kernel [1] shows that performance overhead of added tracepoints is very small, with kernel scheduling overhead benchmarks, `hackbench`, showing results from a degradation of less than 2 % to acceleration of a similar amount, which can be attributed to the operating system noise and cache effects.

8.3 Immediate Values

As pointed out in Section 8.1 on `Kernel Markers`, one of the main performance overhead of disabled static instrumentation techniques presented above is caused by

1. `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=97e1c18e8d17bd87e1e383b2e9d9fc740332c8e2`

reading a value from memory to test and branch over the disabled stack setup and function call.

To overcome this problem, the *Immediate Values* infrastructure, which is presented in [28], replaces the standard memory read, loading the condition variable, by a constant folded in the immediate value encoding of an instruction operand. This removes any data memory access to test for disabled instrumentation by keeping all the information encoded in the instruction stream. However, this involves dynamically modifying code safely against concurrent multiprocessor accesses. This requires either stopping all processors for the duration of the modification, or using a more complex, yet more lightweight, core synchronization mechanism, our choice being the temporary breakpoint bypass [97].

An objective of this infrastructure is to minimize interference with the optimizations when compiling the Linux kernel with gcc. Therefore, all constraints gcc imposes on what can be done in inline assembly gcc-extensions must be taken into account. Jumping outside of assembly statements is forbidden. Given that the function call performed by the instrumentation is produced by the compiler, it cannot be added to the inline assembly.

Initial experiments to replace the immediate value load and test by a static jump (itself jumping to either the enabled or disabled branch) are promising. However, proper implementation of this code flow activation technique requires compiler-level modification, because knowledge of basic block location is required and inline assembly is not permitted to jump to gcc-generated code. A team of kernel and compiler developers at Redhat already started to tackle the task of extending gcc to add this new feature.

8.4 Analysis of LTTng Latency Impact

Benchmarks presented in Chapter 5 mainly address the question of system throughput impact caused by the `LTTng` tracer. However, in some systems, response time is more critical than throughput. This aspect is addressed by performing a comparative study of network response-time benchmarks in the presence and absence of tracing. We choose to measure network latency impact to characterise the tracer because it is a typical application where latency impact must be kept low. Web servers and domain name servers, which must answer queries quickly, are a good example of this

application class.

We determine the tracer impact on the average network response time of a computer by measuring the packet round-trip time of 100,000 *ping* echo requests. This test involves two hosts, one initiating the request and the second answering to it. The round-trip time consists in the time it takes for the packet to be generated by `ping`, sent to the network card through the operating system, sent over the network, received by the second host's operating system kernel, and sent back to the originating host through a similar route.

The repetitive nature of the test might show lower latencies than standard production systems due to the high cache locality of the workload. Hence, to make this test more representative of a real-life operating system, a workload is executed in the background, precisely to trash the processor caches and branch prediction. The chosen workload is a cache-hot Linux kernel build spread across all the machine cores.

The first latency test realised is performed in a setup minimizing the network effect where both the sender and the receiver are on the same computer, using the local host loopback interface. An 8-core Intel Xeon, clocked at 2.0 GHz is used. The number of events recorded per packet is identified by manually inspecting the recorded trace. The 95 % confidence interval for the difference between the two means found in Table 8.1, is $[8.88, 9.12]$ μs, which means that flight recorder tracing of 27 events adds a latency overhead on local host communication between 8.88 and 9.12 μs, with a 95 % certainty. This corresponds to an **added latency between 328 and 338 ns per event**, which is about 666 cycles. It is higher than the overhead measured with micro-benchmarks in the result section of Chapter 5, which is 119 ns per events for this architecture. The difference between these latency results and the micro-benchmarks measurements can be attributed to processor pipeline, branch prediction and cache effects, which are higher in the latency test due to lower temporal and spacial locality than a tight loop calling the tracer.

Table 8.1 Tracer latency overhead for a ping round-trip. Local host, Linux 2.6.30.9, 100,000 requests sample, at 2 ms interval

Test	Events / round-trip	avg. (μs)	std.dev. (μs)
No tracing	–	40.0	12.8
Flight recorder tracing	27	49.0	14.3

Similar results, presented in Table 8.2 are obtained by sending *ping* echo requests from a remote host over a 100 Mb/s network. The number of events generated on the traced receiver side for each echo request is 7. The 95 % confidence interval for the difference between these two means is $[1.56, 2.85]$ μs. Therefore, with 7 events per request, the added latency impact is between 223 and 407 ns per event, which is consistent with the measurements from the local host `ping` test. The confidence interval of network testing is much larger that the local host test due to an higher standard deviation on the measurements.

Table 8.2 Tracer latency overhead for a ping round-trip. 100 Mb/s network, tracing receiver host only, Linux 2.6.30.9, 100,000 requests sample, at 2 ms interval

Test	Events / round-trip	avg. (μs)	std.dev. (μs)
No tracing	–	256.10	73.3
Flight recorder tracing	7	258.31	73.3

Hence, the analysis of these measurements allows us to affirm that the 95 % confidence interval of the tracer latency impact on a busy system is between 328 and 338 ns per event on the Intel Xeon E5405.

8.5 Analysis of LTTng Real-Time Impact

Real-time impact of algorithms can be categorized following the guarantees they provide. The terms used to identify such guarantees evolved through time in the literature [36, 37]. The terminology used in this thesis is detailed in [37]. The strongest non-blocking guarantee is wait-free, which ensures each thread can always make progress and is thus never starved. A non-blocking lock-free algorithm only ensures that the system as a whole always makes progress. This is made possible by ensuring that at least one thread is progressing when concurrent access to a data structure is performed. An obstruction-free algorithm offers an even weaker guarantee than lock-free: it only guarantees progress of any thread executing in isolation, meaning that all competing concurrent accesses can be aborted. Finally, a blocking algorithm does not provide any of these guarantees.

The `LTTng` kernel tracer synchronization at the probe site is wait-free, the strongest possible form of non-blocking guarantee. Given the use of per-`CPU` buffers and pre-

emption disabling, the only possible concurrency comes from interrupts. Therefore, if we know the interrupt frequency (as we should in a real-time system), we can put an upper bound to the number of times a buffer space reservation can be re-started, hence meeting wait-free guarantees. All accesses to trace-control data structures are protected by read-side `RCU`, which is also wait-free.

However, if the algorithm is either deployed in user-space where preemption cannot be disabled cheaply (without going to the kernel), or uses a global buffer shared amongst processors, then the `LTTng` synchronization mechanism only provides non-blocking lock-free guarantees for the write-side. The system as a whole is guaranteed to make progress, because at least one compare-and-swap will succeed amongst each concurrent attempts. However, it is not wait-free in this configuration, because a slow thread may be starved by other concurrent faster threads writing to the buffer, causing the slower thread to never succeed reserving space.

The read-side of the buffering algorithm is blocking: if a writer thread is stopped between its space reservation and commit operations, a partially committed sub-buffer will stop readers from accessing it until the stopped writer thread completes its commit. However, the most important guarantees for a tracer are those affecting the write-side, and thus read-side real-time guarantees are allowed to be weak.

It is important to consider the worse-case execution time of the tracer to provide an accurate upper-bound on the constant time per event added. Section 8.4 presents the *average* latency impact, which cannot be used as an upper bound. It is recommended to consider a sub-buffer switch occurring at each event, reading and writing to cache-cold memory, executing cache-cold instructions, as the worse-case scenario. It is possible to generate a test-case with these worse-case performances by forcing an explicit sub-buffer switch, and by invalidating all the `CPU` cache-lines after each commit. Such a modified `LTTng` tracer takes 6349 ns per event on the Intel Xeon, which is 20 times higher than the average 328–338 ns latency. Part of this cost (238 ns) is due to sub-buffer switch [2]. The main slowdown is caused by access to instructions and data from memory rather than cache. Ensuring this upper execution-time bound more strictly could additionally require disabling other processor features, such as the branch prediction buffers, which make real-time behavior harder to predict. For hard real-time systems, restarts caused by nested interrupt handlers should also be added

2. The 238 ns added by sub-buffer switch is the difference between a cache-hot event write with forced sub-buffer switch (357 ns) and a cache-hot event write (119 ns).

to the worse-case probe execution time.

8.6 Formal verification of LTTng

The formal verification of `LTTng` buffering algorithms is divided in a presentation of the model, followed by the results of correctness, progress and reentrancy verification. This corresponds to a verification of the following properties: absence of corruption, real-time determinism and `NMI` reentrancy.

Algorithms with lock-free and wait-free properties guarantee a predictable real-time impact on the system behavior. As demonstrated in Chapters 4 and 5, using a lock-less buffering scheme also permits to diminish the performance impact compared to locking-based alternatives. However, these come at the expense of increased algorithmic complexity. To make matters worse, modern architectures adds complex memory ordering requirements to ensure proper synchronization. With standard locking, the appropriate memory barriers are added to the locking primitives to ensure correct memory ordering, but lock-free algorithms must deal with these concerns explicitly. These must be taken into account in the wait-free scheme design. To ensure that the `LTTng` tracer indeed meets the real-time and correctness guarantees claimed, we resort to formal verification by model-checking.

We created a Promela model abstracting the `LTTng` buffering scheme to enable the verification of safety and progress properties on this model. The verification by a model-checker proves that all possible model execution traces are free from starvation:

- of the system as a whole (lock-free),
- of each individual thread (wait-free).

It also permits to verify that all model execution traces are free from data corruption between:

- concurrent writer threads,
- reader and writer threads,
- writer thread and nested interrupt handler writer,
- reader thread and nested interrupt handler writer.

This permits to conclude that the `LTTng` model is safe and ensures progress.

8.6.1 Modeling

Due to the state-space explosion inherent to the model, modeling an abstraction of the LTTng buffering scheme is required to keep the state-space size within bounds manageable by the model-checker. This abstraction must cover all the algorithm paths, with a limited amount of redundancy.

Keeping the model state-space within manageable limits has been one of the major challenges of LTTng model-checking. I managed to keep all verifications within 4 GiB of memory using the following techniques. I limited the number of sub-buffers to two, with two entries each. The counters for a buffer could therefore be represented on 2 bits. One extra bit has been added to represent the eventual side-effects of higher order bits. All counters are kept within this range using a modulo operation. Subtractions are done in the range of positive values to ensure Promela does not apply modulo on negative values.

The buffer content, and thus the verification for racy concurrent access to a buffer location, is represented with 1 flag per entry. When a process accesses a buffer location, it checks if this flag is raised with an assertion, sets the flag, and then clears it. This scheme is used to detect incorrect concurrent accesses to a shared memory region.

The buffer model includes an extra sub-buffer owned by the reader process in *flight recorder* mode. It is exchanged atomically with each writer's sub-buffer before accessing them for reading by first checking if the writer has raised the reference flag in the top-level sub-buffer structure pointer. This scheme exchanges entries in a table mapping the buffer offsets to the actual location of the race-verification flags, which corresponds to the memory locations used for read and write. It is important to understand that buffer space reservation synchronization counters are orthogonal to the actual references to physical buffer locations. Therefore, the top-level sub-buffer pointers can be exchanged without modifying the reserve and commit counters.

Modeling multiple concurrent processes happens to be highly space-consuming. An approach where only pairs of individual threads interacting are activated is used. Verification is thus performed in multiple model configurations, each verifying the interaction between different processes. This helps to reduce the state-space required, which is especially important for progress verification. Each process consists of an infinite loop, either reading from or writing to the buffer. Modeling an infinite loop is important for progress verification, because it is based on the presence or absence

of non-progress cycles.

Representation of interrupts is done as a variation of the process model. Interrupts are represented by a separate thread instance waiting to be awakened between each atomic statement executed by the interrupted process. Control only returns to the interrupted process once the interrupt handler has completed its execution.

Another challenge faced has been to ensure that the model coverage is adequate. The Spin model-checker indicates which statements has been executed, and which have not, but does not provide information about state coverage. For instance, it does not indicate if the model reached a 3-bit counter overflow, which could present corner-cases. We proceed to verification of the model coverage by error-injection. We verify that the model-checker reaches a state where 17 events were written by adding an assertion. The same technique is used to ensure that 17 events were lost, and 17 events were read. The choice of the value 17 is based on the amount of events required to reach a commit counter 3-bit overflow. This ensures that all possible values within the 3-bit commit counter, counting from 0 to 7 for each of the two sub-buffers, were encountered at least once, and that the overflow state has been covered. This naturally includes overflow of the reserve and consumer counts, which overflow twice per commit-count overflow.

The model realised for `LTTng` is a simpler model than the framework presented in Chapter 7: it assumes sequentially consistent machines. Besides this assumption, concurrent execution of write-sides on multiple processors and from interrupt and `NMI` context are modeled, with concurrent readers, in non-overwrite and overwrite (`flight recorder`) modes. A model including the effects of memory reordering could be done, but would involve carefully ensuring that the state-space keeps a manageable size.

8.6.2 Correctness

Appropriate synchronization of multiple writers, a reader thread and interrupts is verified using the per-entry flag, which identifies that a buffer entry is being accessed. It permits detections of invalid concurrent accesses to portions of the ring-buffers and treats them as errors. This ensures that threads have exclusive access to ring-buffer slots.

This verification has been performed with the following scenarios, for both overwrite and non-overwrite modes:

– concurrent writer threads,

– reader and writer threads,

The verification has been successful for each scenario. The model-checker required up to 3.0 GiB of memory to perform an exhaustive model safety verification. Each verification required up to 2 minutes. The depth of the execution traces graph manipulated reached 2.1 million states. Nested interrupt verification required less resources, because they generate less possible states than two threads executing concurrently. Concurrent threads executing in overwrite mode required the most computational resources.

8.6.3 Real-Time Impact

Providing hard real-time guarantees usually require to audit each component of the system to take into account priority inversions caused by blocking. As shows the detailed analysis of the priority inversion problems that occurred on the Mars Pathfinder [98], tracer tools are often the only type of diagnostic tool able to debug systems facing real-time deadline problems in the field.

However, in order for this type of tool to be reliable and useful in such systems, the real-time behavior of the system must be, ideally, left unchanged when the tracer is active. Blocking on mutual exclusion semaphores is therefore out of question, because it would completely change the priority inheritance chains between the processes due to an added shared resource: the trace buffers.

The approach chosen for `LTTng` to deal with real-time is to never block on any resource whatsoever when called from the instrumented kernel. This has been achieved by resorting to wait-free algorithms to manage synchronization between multiple concurrent contexts. The probe executes a bounded number of instructions, including `CAS` loops, which can be restarted by a well-defined set of concurrent operations, performed either by local interrupt handlers or a limited number of reader processes.

This class of algorithms provides the following guarantees: each individual thread is guaranteed to never be starved by any other concurrent process. Hence, the system as a whole, as well as each individual thread, are ensured to always progress. This encompasses the weaker "lock-free" guarantee. Both of these algorithm classes imply that the algorithm never blocks.

I performed the formal verification that this guarantee is actually met by using

the same Promela model of the LTTng buffering scheme used for correctness verification. Spin permits to verify for non-progress loops in the program. Verifying for system-wide progress (lock-free) is done by adding progress statements in each thread executed in the model. Weak fairness must be disabled, because the verifier would not consider loops where one thread is stopped. A stricter verification, for wait-free guarantees, is performed by only adding a progress label to a single thread and by running the verification only considering non-progress for infinite execution cycles where the thread of interest is being enabled and executed. This permits to verify that a single process is starvation-free, and thus that the algorithm is wait-free.

The result is that LTTng is wait-free when each CPU writes in its own buffers and that LTTng is only lock-free when multiple processors can write to the same buffer or if preemption is left enabled on a single processor. This is caused by the ability for a fast writer thread to continuously succeed at reserving buffer space, while a slower thread would indefinitely fail, and thus starve. Therefore, the kernel-level LTTng have wait-free guarantees, and user-space implementations, due to their inability to disable preemption when reserving buffer space, are only lock-free.

Kernel-level wait-free verification has been performed by modeling, in both overwrite and non-overwrite modes:

- one writer, one reader thread,
- one writer, with a nested interrupt handler writer.

User-level lock-free verification has been performed by modeling two writer threads writing in a shared buffer. The system-wide progress verification succeeds, but the per-process progress validation fails, due to space reservation starvation.

The overwrite mode has two additional synchronizations. The first is that the writer can push the reader position when the buffer is full. This is performed with a CAS operation on the reader's position. It has been verified that the CAS loop can never cause writer starvation, because the reader's position can only be concurrently changed by both the reader and the writer in the same direction. Hence, the writer can only retry this "push" operation for an amount of times bounded by the number of sub-buffers per buffer, which is the maximum number of updates which can make the writer thread restart before the reader empty the buffer and blocks.

The second synchronization added by the overwrite mode involves letting the reader thread exchange individual sub-buffers with its own extra sub-buffer before reading them. This exchange is performed with a CAS instruction which verified if

the *use* flag is set by the writer. If the sub-buffer is in use, the reader will retry later. On the writer side, a CAS instruction is used in a busy-loop to set and clear the *use* flag (for portability, because an atomic instruction to set a mask is not available for all architecture in the Linux kernel). Given that the reader can only ever exchange a sub-buffer and cause the writer to restart once (further accesses to that sub-buffers are impossible, because the reader will block on the writer position), the reader thread cannot starve the writer.

The model involving one reader and one writer thread in overwrite mode required significantly more resources than the other verifications. Resources required to perform this verifications reached 1.5 GiB of memory, but required the use of BDDs (*Binary Decision Diagrams*) to encode the state-space more compactly, leading to a much slower execution of the model-checker, which took 25 hours to verify the model. The search depth reached 3.0 million execution steps.

8.7 Reentrancy

Verification of the LTTng buffering scheme NMI-reentrancy is performed using the same model used to ensure lock-free and wait-free guarantees of the algorithm.

The correctness verification presented in Section 8.6.2 ensures that no data corruption can occur when two concurrent threads, either two writers or one reader and one writer, execute. Proving that NMI handlers nesting on threads cannot corrupt the trace buffers is a special case of modeling two concurrent threads. The model involving two standard writer threads and the model involving one reader and one writer ensure that up to an infinite number of interrupts executing between two atomic steps do not cause corruption.

The progress verification presented in Section 8.6.3, which guarantee lock-free behavior for user-space implementations and wait-free behavior for kernel implementations, suffice to ensure that NMIs nesting on threads cannot cause deadlocks. Absence of deadlock due to one or multiple interrupts between two atomic steps is verified by the standard thread models, where system progress is ensured when two threads (either two writers or one reader and one writer) execute concurrently. This includes the case where one thread is stopped and the other thread (representing the interrupt) executes forever. Therefore, writer interrupts nested over either the reader or the writer cannot cause any system starvation nor deadlock.

Modeling of nested interrupt handler impact on wait-free guarantees is performed by verifying the interrupt model progress with weak fairness. This type of scheduling is required to model the effect of interrupts on a local processor, which could not, for instance, be scheduled like an ordinary thread. Therefore, the `LTTng` buffering algorithm could be considered lock-free with respect to interrupts, because they cause the writer thread to restart. However, given the wait-free and lock-free definitions apply to two concurrent threads and not to interrupts nesting over them, the wait-free guarantee applies integrally to the `LTTng` buffering scheme. For all practical purposes, interrupt rate is usually known in a real-time system, which is not necessarily true for concurrent thread execution speed.

Chapter 9

General Discussion

This chapter recalls the tracer properties identified as research objectives and explains, by construction, how the tracer respects them. This is followed by a discussion of the application domains enabled by satisfying these properties. Finally, contributions to other scientific projects and to the Linux kernel are presented to demonstrate the scientific and industrial impacts of this research.

9.1 Tracer Properties

After studying the needed properties, we implemented an innovative tracer meeting all our requirements. The properties we identified led the tracer design choices, allowing us to fulfill requirements not met by previously existing tracers. Ensuring that the `LTTng` tracer respects the:

- latency,
- throughput,
- scalability,
- real-time,
- portability,
- and reentrancy

properties requires that each tracer component executing in the traced execution context respects them. The components to consider are: *Tracepoints*, *Linux Kernel Markers*, *Immediate Values*, trace clock, tracing control and the `LTTng` buffering scheme.

The *Tracepoints* and *Linux Kernel Markers* use the `RCU` synchronization mechanism to deal with concurrency from multiple execution contexts. Only an `RCU` read-side lock is required on the traced execution context to protect the array of callbacks to call. Enabling and disabling the instrumentation sites is performed with an atomic modification of the condition variable, which does not require any kind of synchro-

nization whatsoever. As we have shown in Chapter 7, `RCU` read-side meets all the needed properties.

The *Immediate Values*, an optimization of the *Tracepoints* and *Linux Kernel Markers* condition variable, perform dynamic code modification. Dealing with Intel and AMD erratas with respect to code modification in multiprocessor environment, however, involves core synchronization. This is performed by using a scheme involving a breakpoint executing bypass code and `IPI`s sent to all other processors to ensure they execute a core serializing instruction, as shown in Section 8.3. Using this scheme allows us to perform code modification while letting the modified code be executed through the bypass. This therefore satisfies the same wait-free guarantees as `RCU`, which ensures deterministic real-time and allows `NMI` reentrancy.

The time-source used is typically a direct register read, which involves no synchronization. However, when only 32-bit time-stamp counters are available, extending them to 64 bits is necessary. As presented in Section 4.6.3, we propose a `RCU`-based trace clock to offer such a 64-bit time-base to these architectures. The traced execution context only need to use `RCU` read-side locking, which satisfy our properties.

Tracing control operations, which includes the creation and deletion of active trace sessions and modification of their state while tracing is active is also synchronized using `RCU` read-side.

Last but not least, the `LTTng` buffering scheme has been verified to meet all of the low-latency, high-throughput, scalability, deterministic real-time impact (wait-free), portability, and reentrancy properties.

By demonstrating that each tracer component respects these properties, we can affirm, by construction, that they are also satisfied by the tracer as a whole. As a result, the `LTTng` tracer goes further than existing tracing solutions, which each satisfy only a few of these properties.

9.2 Tracer Application Domains

This section associates each property met by the tracer to the application domains reached, and gives examples of users requiring tracing in each of these domains, some of which are already using `LTTng`. This demonstrates that this research has an important impact on the industry in all of the application domains targeted by the identified properties.

The properties of low-latency, low-throughput impact, and linear scalability, targets commercial servers running Linux. The Google servers are a good example of systems requiring such a tracing solution and willing to integrate a tracer. They need to enable tracing on their **live production servers** in order to be able to reproduce and solve performance issues and bugs. This requires that the **overhead and disturbance** of the tracer must be **minimal**. The `LTTng` tracer, by meeting the aforementioned properties, meets these low-impact requirements and is a viable solution for Google servers.

Soft real-time applications at Autodesk use `LTTng` within their development. These rely on meeting soft real-time constraints, which require the tracer to have a low impact both on system throughput and real-time response on multi-core computers. This type of property is also needed by Ericsson for their telecommunication equipment. Siemens also rely on `LTTng` internally for the development of some of their products running Linux.

Ensuring that the code executed by the probe is wait-free enables tracing of real-time systems without changing their behavior in a non-predictable way. `LTTng` is already integrated as the tracing solution for Wind River Linux, Monta Vista and STLinux distributions. These distributions also benefit from the portability of `LTTng` by allowing tracing of various computer architectures.

Portability also enables Nokia to use `LTTng` for the development of their Maemo internet tablets and phones based on the ARM OMAP3 architecture.

Reentrancy of the tracer code for contexts ranging from thread context to interrupt and `NMI` contexts is important to allow large instrumentation coverage. This benefits Linux kernel developers who need to extend kernel instrumentation: they do not have to understand the tracer internals to ensure the instrumentation they add will not crash the kernel when they enable tracing. Hence, this contributes to the Linux end users by ensuring a more stable tracing infrastructure which can be trusted.

All these applications of `LTTng` demonstrate that it fills a tracing need in many industry application fields for low-impact kernel tracing. The consequence of our research is to improve multi-core system debugging facilities, providing a tool which helps to find performance bottlenecks, hence speeding up applications by finding all sorts of inefficient resource usage. This helps improving response time, real-time response, system throughput and energy efficiency.

9.3 Contributions

This research had other impacts than those directly related to the LTTng tracer. The *Local Atomic Operations*, *Kernel Markers* and *Tracepoints*, individually contributed to other fields and other projects. This section shows the influence of these contributions. We show that user-space tracing, which is outside of the main scope of this research, has been pioneered by this research. User-space tracing is discussed in this section.

9.3.1 Local Atomic Operations

After identifying the benefit of per-CPU local atomic operations in Chapter 4, we identified that extending the Linux kernel to support these operations was beneficial. For example, per-CPU counters and CPU-local data shared between interrupt handlers and thread context can be accelerated using this technique. Therefore, we implemented them for all architectures supported by Linux, using the slower SMP-aware atomic operations as a fall-back when needed. We also added documentation of these operations to the Linux kernel tree to explain their usage and semantics.

Local atomic operations are meant to provide fast and highly reentrant per-CPU counters. They minimize the performance cost of standard atomic operations by removing unneeded inter-CPU synchronization. This is achieved by removing the LOCK prefix and memory barriers normally required to synchronize across CPUs.

Having fast per-CPU atomic counters is interesting in many cases: it does not require disabling interrupts to protect from interrupt handlers and it permits coherent counters in NMI handlers. It is especially useful for tracing purposes and for various performance monitoring counters.

The documentation written is now part of the Linux kernel tree [1]. The per-CPU atomic operations we created have been added to the per-architecture `local.h` files in the mainline Linux tree.

We expect to see a much broader use of these primitives in other components of the Linux kernel because they are now accessible to other kernel programmers, and because they are now documented with a well-defined semantic.

1. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/local_ops.txt

9.3.2 Kernel Markers

The `Kernel Markers` are a static instrumentation mechanism allowing kernel programmers to add instrumentation at the source-code level. They are the equivalent of `printk()` in the Linux kernel for tracing instrumentation. They are principally used for instrumentation prototyping. The improvement brought by this infrastructure over the existing `Kprobes` is to enable instrumentation added directly at the source-code level. This allows us to identify of local variables by the instrumentation, hence guaranteeing their availability for extraction when tracing is enabled.

It allows us to extract richer and more precise information from the kernel, enabling improved debugging and tracing kernel facilities. This infrastructure has been added to the Linux kernel mainline. It has been used to replace the `KVM` (*Kernel-based Virtual Machine*) and `SPE` (*Synergistic Processing Element*) ad-hoc instrumentations present in the mainline kernel.

9.3.3 Tracepoints

Tracepoints are an evolution of the *Linux Kernel Markers*, which provide stricter type verification. Their benefit is also a more organized management of tracepoints by keeping a global registry along with the kernel headers.

They have also been integrated to the Linux kernel, and are used extensively by the Ftrace tracer. Using tracepoints allows us to decouple the tracing facilities from the actual source-code instrumentation. The `Tracepoint` infrastructure allows many concurrent tracers to be connected, which makes it a suitable core facility for kernel instrumentation. It benefits Linux kernel developers by letting them organize kernel instrumentation in a global tracepoint registry. Consequently, this also lets tracers have a single common information source to consider, and thus enables kernel-wide tracing.

As of today, 104 tracepoints are present in the mainline Linux kernel and major development trees, some created by myself, but the vast majority contributed by other Linux kernel developers.

9.3.4 Fast User-space Tracing

User-space tracing aims at allowing tracing from lesser-privileged execution contexts. Its usefulness consists in letting application and library developers instrument their own user-space applications to gather more precise information about their behavior to augment a system-wide trace. The objective is to obtain traces of both kernel and user-space execution, allowing analysis to be performed on combined traces. This will open the possibility to analyze and pinpoint bugs and performance issues resulting in interactions between user-space applications, libraries and kernel-space.

One approach consists in using a system call to enable to use the kernel tracer as-is to record events. However, this approach suffers from the overhead of the system call trap, which is in many cases slower than tracing itself. This is why writing directly to a shared ring-buffer from user-space is a more appealing solution.

Multiple prototypes of user-space tracers have been realised by porting the kernel infrastructure to user-space context. The papers presented at the Linux Symposium [27, 28] present two different aspects of user-space tracing. The first paper [27] presented a port of the `LTTng` kernel tracer to user-space, where a library was responsible to perform the user-level tracing through a memory buffer shared between the application and forked processes. This work was mostly intended to be a prototype. The second paper [28] presented a user-level instrumentation prototype: a port of the Linux Kernel Markers to user-space. These two prototypes, and the kernel-level `LTTng` development performed since then, served as basis for the `UST` tracing library currently being developed.

The work on user-level `RCU` synchronization presented in Chapter 6 has been conducted with the primary objective of addressing the synchronization needs from `UST` (*User-Space Tracer*). Given the fact that the user-space tracer implementation is derived from the kernel `LTTng` tracer, it relies on the same synchronization primitives to provide consistent tracer data structure access. Therefore, the lack of a proper `RCU` implementation needed by the tracer library needed to be addressed to ensure that the user-space tracer could benefit from performance characteristics similar that of the kernel tracer.

User-space tracing opens a very promising research area by allowing collection of detailed system-wide information. One actual limitation identified by modeling the user-space buffering scheme is the fact that wait-free progress guarantees are lost due to the inability to disable preemption, leaving only lock-free guarantees. It

opens an interesting research area for scheduler interactions of real-time user-space applications.

9.4 Scientific Studies Using LTTng

Our research has interesting impacts on scientific studies in other computer research fields. It contributed to research which benefits from a very efficient tracing tool. This section presents experiments where the `LTTng` tracer contributed to solving problems and identifying solutions.

`LTTng` has been used to find the power variations over time in disk operations to `NOR` and `NAND` flash devices at the driver level. It helped correlate device accesses with `CPU` activity to find which type of memory requires less energy. This work has been presented in article [99].

`LTTng` has also been used to trace user behavior over a longer time-period [100]. Over a week, user habits have been studied by tracing the `exec()` and `exit` system calls. They used this information to see which applications are most likely to be run concurrently.

Paper [101] describes the architecture of an operating system for future information appliances. They use `LTTng` as their tracing infrastructure to feed information to an anomaly detection service.

Hicham Marouani, in his work on internal clock drift estimation [102, 103], used the `LTTng` kernel tracer to keep track of network packet arrival and monitor a `GPS`-based reference clock to measure the precision of various techniques to synchronize time across nodes.

Monitoring of kernel execution in the Lemona project [104] uses hooks inspired from the Tracepoint and Kernel Marker mechanisms to allow probes to be dynamically connected while keeping the performance impact low when disabled.

All these contributions of the `LTTng` tracer demonstrate that kernel-wide tracing is useful to scientific research.

Chapter 10

Conclusion and Recommendations

The core realization of this research is the creation of innovative synchronization algorithms enabling the implementation of the LTTng tracer for the Linux operating system. This tracer satisfies properties of low-impact on the operating system scalability, throughput and average latency, deterministic real-time response impact, portability to various architectures and high degree of reentrancy. Benchmarks and formal verification have shown that each of the tracer components meets these properties. The LTTng tracer meets requirements that the existing tracer predecessors only met partially, which enables tracing of the Linux operating system, whose flexibility allows its use in a large spectrum of application fields.

It has been possible to achieve these goals by carefully choosing, creating, designing and implementing synchronization mechanism, which are RCU for read-side synchronization and the custom LTTng buffering scheme to synchronize writes. Both provide linear scalability and wait-free algorithmic guarantees, which are useful for tracing multi-core systems, as well as ensuring real-time and reentrancy guarantees.

The original scientific contributions of this research include:

- the creation of the LTTng buffer synchronization algorithm,
- the creation of an RCU-based trace clock,
- the design of a complete kernel tracer based on wait-free, linearly scalable and NMI-safe algorithms,
- application of self-modifying code techniques to efficiently manage instrumentation activation,
- improvements to the RCU synchronization mechanisms in user-space context,
- creation of a generic architecture model for formal verification of parallel algorithms, modeling weakly-ordered memory accesses and instruction scheduling.

All these contributions enabled the creation of a kernel tracer meeting all the research objectives.

The appropriate response to the industry and open source community tracing

requirements is demonstrated by the fact that some tracer components we created (*Tracepoints* and *Linux Kernel Markers*) are integrated to the mainline Linux kernel and that the LTTng tracer has a large community of users and contributors, namely Google, IBM, Ericsson, Autodesk, Wind River, Fujitsu, Monta Vista, STMicroelectronics, C2 Microsystems, Sony, Siemens, Nokia, and Defence Research and Development Canada.

As a conclusion of this research, we can affirm that tracing heavy workloads on a mainstream operating system running on multi-core architectures can be achieved with only a minimal impact on the system's throughput and average latency, while preserving entirely the scalability, real-time response, portability and reentrancy of the operating system. The implementation realized permits instrumentation coverage of the entire operating system kernel, including NMI handlers.

Analysis of system-wide trace information involves the collection of traces from both the kernel and user-levels. Following the promizing results of early experiments done on fast user-space tracing, it is now worth stabilizing an infrastructure to provide fast production-level user-space tracing to Linux users.

The heavy workloads which can now be traced on production systems enables the collection of information to analyse and solve performance and behavior problems in today's complex computer systems. It it now worth exploring analysis made possible by extracting this information by modeling the operating system to perform trace-driven analysis.

Due to its usefulness for system monitoring, identification of performance bottlenecks and debugging, having tracing active at all times on production servers and appliances is a natural decision for systems developers *if* the performance penality is low enough. This research demonstrated clearly that the impact of LTTng tracer, when active, is low enough that it can be used on heavily loaded production systems without hurting performances prohibitively.

List of References

[1] M. Desnoyers and M. R. Dagenais, "Synchronization for fast and reentrant operating system kernel tracing," *Software – Practice and Experience*, to appear: re-submitted after review.

[2] M. Desnoyers and M. R. Dagenais, "Lockless multi-core high-throughput buffering scheme for kernel tracing," *ACM Transactions on Computer Systems (TOCS)*, to appear: submitted.

[3] M. Desnoyers, M. R. Dagenais, P. E. McKenney, A. Stern, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to appear: submitted.

[4] M. Desnoyers, M. R. Dagenais, and P. E. McKenney, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to appear: submitted.

[5] P. Stenstrom, E. Hagersten, D. Lilja, M. Martonosi, and M. Venugopal, "Trends in shared memory multiprocessing," *Computer*, vol. 30, no. 12, pp. 44–50, Dec 1997.

[6] M. Johnson, *Superscalar microprocessor design*, ser. Prentice Hall series in innovative technology. Prentice Hall, 1991, [Online]. Available: http://www.worldcat.org/isbn/0138756341. [Accessed: October 19, 2009].

[7] R. Love, *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 6–8.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 205–218.

[10] R. W. Wisniewski, R. Azimi, M. Desnoyers, M. M. Michael, J. Moreira, D. Shiloach, and L. Soares, "Experiences understanding performance in a com-

mercial scale-out environment," in *European Conference on Parallel Processing (Euro-Par)*, 2007.

[11] ARM Limited Technical Staff, *"ARM11 MPCore" Processor Technical Reference Manual. Revision r1p0.* ARM Limited, 2006.

[12] U. Rüde, "Technological trends and their impact on the future of supercomputers," in *High Performance Scientific and Engineering Computing, volume 8 of Lecture notes in Computational Science and Engineering.* Springer-Verlag, 1999, pp. 459–471.

[13] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20(1), pp. 46–61, January 1973.

[14] J. J. Labrosse, *Microc/OS-II.* R & D Books, 1998.

[15] P. A. Laplante, *Real-Time Systems Design and Analysis.* John Wiley & Sons, 2004.

[16] P. Malhotra, "Issues involved in real-time rendering of virtual environments," Master's thesis, Faculty of Virginia Polytechnic, 2002.

[17] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *Micro, IEEE*, vol. 23, no. 2, pp. 22–28, 2003, [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1196112. [Accessed: October 19, 2009].

[18] S. Baskiyar and N. Meghanathan, "A survey of contemporary real-time operating systems," *Informatica (Slovenia)*, vol. 29, no. 2, pp. 233–240, 2005, [Online]. Available: http://www.informatica.si/PDF/29-2/12_Baskiyar-ASurveyofContemporary...pdf. [Accessed: October 19, 2009].

[19] K. Yaghmour and M. R. Dagenais, "The Linux Trace Toolkit," *Linux Journal*, May 2000, [Online]. Available: http://www.linuxjournal.com/article/3829. [Accessed: October 19, 2009].

[20] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: detecting the unexpected in distributed systems," in *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation.* Berkeley, CA, USA: USENIX Association, 2006, pp. 9–9.

[21] (2008, January) Tracingsummit2008. [Online]. Available: http://ltt.polymtl.ca/tracingwiki/index.php/TracingSummit2008. [Accessed: October 19, 2009].

[22] (2009, July) Tracingminisummit2009. [Online]. Available: http://ltt.polymtl.ca/tracingwiki/index.php/TracingSummit2009. [Accessed: October 19, 2009].

[23] J. Corbet. (2008, September) Ks2008: Tracing. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/298685/. [Accessed: October 19, 2009].

[24] M. Bligh, R. Schultz, and M. Desnoyers, "Linux kernel debugging on Google-sized clusters," in *Proceedings of the Ottawa Linux Symposium*, 2007.

[25] M. Desnoyers and M. Dagenais, "OS tracing for hardware, driver and binary reverse engineering in Linux," *CodeBreakers Journal*, pp. Vol. 4, No. 1., 2007.

[26] M. Desnoyers and M. Dagenais, "OS tracing for hardware, driver and binary reverse engineering in Linux," in *Recon Conference Proceedings*, 2006.

[27] M. Desnoyers and M. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Proceedings of the Ottawa Linux Symposium*, 2006.

[28] M. Desnoyers and M. R. Dagenais, "LTTng: Tracing across execution layers, from the hypervisor to user-space," in *Proceedings of the Ottawa Linux Symposium*, 2008.

[29] M. Desnoyers and M. Dagenais, "LTTng, filling the gap between kernel instrumentation and a widely usable kernel tracer," in *Linux Foundation Collaboration Summit*, 2009, [Online]. Available: http://events.linuxfoundation.org/archive/lfcs09_desnoyers_paper.pdf. [Accessed: October 19, 2009].

[30] M. Desnoyers and M. Dagenais, "Low disturbance embedded system tracing with Linux Trace Toolkit Next Generation," in *ELC (Embedded Linux Conference)*, 2006.

[31] M. Desnoyers and M. Dagenais, "Deploying LTTng on exotic embedded architectures," in *ELC (Embedded Linux Conference)*, 2009, [Online]. Available: http://tree.celinuxforum.org/CelfPubWiki/ELC2009Presentations?action=AttachFile&do=view&target=desnoyers-celf2009-paper.pdf. [Accessed: October 19, 2009].

[32] P. E. McKenney, J. Triplett, and J. Walpole. Relativistic programming. [Online]. Available: http://wiki.cs.pdx.edu/rp/. [Accessed: October 19, 2009].

[33] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX*, 2004, [Online].

Available: http://www.sagecertification.org/events/usenix04/tech/general/full_papers/cantrill/cantrill_html/index.html. [Accessed: October 19, 2009].

[34] J. Corbet. (2007, August) On DTrace envy. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/244536/. [Accessed: October 19, 2009].

[35] P. E. McKenney, "Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels," Ph.D. dissertation, OGI School of Science and Engineering at Oregon Health and Sciences University, July 2004, [Online]. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf. [Accessed: October 19, 2009].

[36] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings 15th ACM Symposium on Principles of Distributed Computing*, 1996, pp. 267–275.

[37] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *In Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003, pp. 522–529.

[38] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, and al, "K42: building a complete operating system," in *EuroSys '06: Proceedings of the 2006 EuroSys conference*, 2006, pp. 133–145.

[39] R. W. Wisniewski and B. Rosenburg, "Efficient, unified, and scalable performance monitoring for multiprocessor operating systems," in *Supercomputing, ACM/IEEE Conference*, 2003, [Online]. Available: http://www.research.ibm.com/K42/papers/sc03.pdf. [Accessed: October 19, 2009].

[40] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Proceedings of the Ottawa Linux Symposium*, 2005, [Online]. Available: http://sourceware.org/systemtap/systemtap-ols.pdf. [Accessed: October 19, 2009].

[41] A. Nataraj, A. Malony, S. Shende, and A. Morris, "Kernel-level measurement for integrated parallel performance views: the KTAU project," in *IEEE International Conference on Cluster Computing*, 2006.

[42] Oprofile: A system-wide profiling tool for Linux. [Online]. Available: http://oprofile.sourceforge.net. [Accessed: October 19, 2009].

[43] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the guts of kprobes," in *Proceedings of the Ottawa Linux Symposium*, 2006.

[44] LTTng website. [Online]. Available: http://www.lttng.org. [Accessed: October 19, 2009].

[45] Intel Corporation Technical Staff. (2006, September) Intel 64 and IA-32 architectures software developer's manual.

[46] J. Wetzel, E. Silha, C. May, B. Frey, J. Furukawa, and G. Frazier, *PowerPC Virtual Environment Architecture, Version 2.02*, February 2005, [Online]. Available: http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html. [Accessed: June 7, 2009].

[47] J. Heinrich, *MIPS R4000 Microprocessor User's Manual, Second Edition.* MIPS Technologies, Inc., 1994.

[48] ARM Limited Technical Staff, *Cortex-A8 Technical Reference Manual, revision r1p1.* ARM Limited, 2006.

[49] Intel Corporation. (2004, October) IA-PC HPET (high precision event timers) specification. [Online]. Available: http://www.intel.com/hardwaredesign/hpetspec_1.pdf. [Accessed: October 19, 2009].

[50] G. Hillier. (2008) System and application analysis with LTTng. [Online]. Available: Siemens Linux Inside, http://www.hillier.de/linux/LTTng-examples.pdf. [Accessed: June 7, 2009].

[51] J. Corbet. (2008, July) Tracing: no shortage of options. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/291091/. [Accessed: October 19, 2009].

[52] J. Corbet. (2007, August) Kernel Markers. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/245671/. [Accessed: October 19, 2009].

[53] T. Zanussi, K. Y. R. Wisniewski, R. Moore, and M. Dagenais, "RelayFS: An efficient unified approach for transmitting data from kernel to user space," in *Proceedings of the Ottawa Linux Symposium*, 2003, pp. 519–531, [Online]. Available: http://www.research.ibm.com/people/b/bob/papers/ols03.pdf. [Accessed: October 19, 2009].

[54] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," in *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999, pp. 87–100.

[55] J. P. Hennessy, D. L. Osisek, and J. W. Seigh II, "Passive serialization in a multitasking environment," Washington, DC, p. 11, February 1989.

[56] V. Jacobson, "Avoid read-side locking via delayed free," September 1993, private communication.

[57] A. John, "Dynamic vnodes – design and implementation," in *USENIX Winter 1995*. New Orleans, LA: USENIX Association, January 1995, pp. 11–23.

[58] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998, pp. 509–518.

[59] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, 2007.

[60] K. A. Fraser, "Practical lock-freedom," Ph.D. dissertation, King's College, University of Cambridge, 2003.

[61] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux," *IBM Systems Journal*, vol. 47, no. 2, pp. 221–236, May 2008.

[62] P. E. McKenney, "Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels," Ph.D. dissertation, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[63] P. E. McKenney. (2008, January) What is RCU? part 2: Usage. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/263130/. [Accessed: October 19, 2009].

[64] E. Polyakov. (2009, April) The elliptics network. [Online]. Available: http://www.ioremap.net/projects/elliptics. [Accessed: October 19, 2009].

[65] T. Jinmei and P. Vixie, "Implementation and evaluation of moderate parallelism in the BIND9 DNS server," in *Proceedings of the annual conference*

on USENIX Annual Technical Conference, Boston, MA, February 2006, pp. 115–128, [Online]. Available: https://www.usenix.org/events/usenix06/tech/full_papers/jinmei/jinmei.pdf. [Accessed: June 3, 2009].

[66] M. Herlihy, "Wait-free synchronization," *ACM TOPLAS*, vol. 13, no. 1, pp. 124–149, January 1991.

[67] P. E. McKenney, "Using a malicious user-level RCU to torture RCU-based algorithms," in *linux.conf.au 2009*, Hobart, Australia, January 2009.

[68] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *Transactions of Computer Systems*, vol. 9, no. 1, pp. 21–65, February 1991.

[69] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *ACM Queue*, September 2008.

[70] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *HPCA Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 97–108.

[71] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, "Scalable and reliable communication for hardware transactional memory," in *PACT Proceedings of the 17th international conference on Parallel architectures and compilation technique*, 2008, pp. 144–154.

[72] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commericial hardware transactional memory implementation," in *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Washington, DC, USA, March 2009, p. 12.

[73] P. E. McKenney. (2007, August) Using Promela and Spin to verify parallel algorithms. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/243851/. [Accessed: October 19, 2009].

[74] P. E. McKenney and S. Rostedt. (2008, April) Integrating and validating dynticks and preemptable RCU. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/279077/. [Accessed April 24, 2008].

[75] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," in *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem*

Proving in Higher Order Logics. Berlin, Heidelberg: Springer-Verlag, pp. 391–407.

[76] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of power and ARM multiprocessor machine code," in *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*. New York, NY, USA: ACM, 2008, pp. 13–24.

[77] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[78] D. A. Schmidt, "Data flow analysis is model checking of abstract interpretations," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Diego, California, 1998, pp. 38–48.

[79] B. Steffen, "Data flow analysis as model checking," in *Lectures Notes in Computer Sciences. Theoretical Aspects of Computer Software: TACS*, vol. 256, 1991, pp. 346–364, [Online]. Available: http://www.springerlink.com/content/y5p607674g6q1482/. [Accessed: June 4, 2009].

[80] M. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," in *In Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, 1994, pp. 62–75.

[81] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and Software Verification, Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[82] G. J. Holzmann, "The SPIN model checker," *IEEE Transactions on Software Engineering*, 1997.

[83] J. van Leeuwen, Ed., *Handbook of theoretical computer science (vol. B): formal models and semantics*. Cambridge, MA, USA: MIT Press, 1990.

[84] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8(1), pp. 39–64, 1996, [Online]. Available: http://www.dcs.warwick.ac.uk/~doron/ps/full.ps. [Accessed: June 3, 2009].

[85] P. E. McKenney, "Memory ordering in modern microprocessors, part II," *Linux Journal*, July 2005, [Online] Available: http://www.linuxjournal.com/article/8212. [Accessed: June 3, 2009].

[86] R. E. Kessler, “The Alpha 21264 microprocessor,” *IEEE Micro*, vol. 19(2), pp. 24–36, March 1999, [Online] Available: http://www.cs.virginia.edu/~dp8x/alpha21264/paper/kessler.pdf. [Accessed: June 3, 2009].

[87] J. L. Peterson, “Petri nets,” *ACM Computing Surveys (CSUR)*, vol. 9(3), pp. 223–252, 1977.

[88] K. Jensen, “Coloured petri nets,” *Petri Nets: Central Models and Their Properties*, vol. 254, pp. 248–299, 1987.

[89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct 1991, [Online]. Available: http://doi.acm.org/10.1145/115372.115320. [Accessed: October 19, 2009].

[90] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Transactions on Computers*, vol. C-35(8), pp. 677–691, 1986.

[91] R. E. Bryant, “Symbolic boolean manipulation with ordered binary decision diagrams,” *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[92] C.-Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, “Value speculation scheduling for high performance processors,” *SIGPLAN Not.*, vol. 33, no. 11, pp. 262–271, 1998.

[93] C. ying Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, “Software-only value speculation scheduling,” Dept. of Electrical and Computer Eng., North Carolina State University, Tech. Rep., June 1998.

[94] P. E. McKenney. (2007) C++ data-dependency ordering: Atomics. [Online]. Available: ISO/IEC JTC1 SC22 WG21 N2359, http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2359.html. [Accessed: October 19, 2009].

[95] P. E. McKenney. (2007) C++ data-dependency ordering: Memory model. [Online]. Available: ISO/IEC JTC1 SC22 WG21 N2360, http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2360.html. [Accessed: October 19, 2009].

[96] P. E. McKenney. (2007) C++ data-dependency ordering: Function annotation. [Online]. Available: ISO/IEC JTC1 SC22 WG21 N2361, http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2361.html. [Accessed: October 19, 2009].

[97] M. Hiramatsu and S. Oshima, “Djprobes - kernel probing with the smallest overhead,” in *Proceedings of the Ottawa Linux Symposium*, 2006.

[98] M. P. F. S. C. E. Glenn Reeves. (1997, December) What really happened on mars? [Online]. Available: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html. [Accessed: October 19, 2009].

[99] T. Stathopoulos, D. McIntire, and W. Kaiser, "The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes," in *International Conference on Information Processing in Sensor Networks (ISPN)*, 2008, pp. 383–394, [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4505489. [Accessed: June 30, 2009].

[100] J. Hom and U. Kremer, "Execution context optimization for disk energy," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 255–264, [Online]. Available: http://portal.acm.org/citation.cfm?id=1450095.1450132. [Accessed: June 30, 2009].

[101] T. Nakajima, H. Ishikawa, Y. Kinebuchi, M. Sugaya, S. Lei, A. Courbot, A. van der Zee4, A. Aalto4, and K. K. Duk4, *An Operating System Architecture for Future Information Appliances*, 2008, vol. Volume 5287/2008, [Online]. Available: http://www.springerlink.com/content/g706532745127812/. [Accessed: June 30, 2009].

[102] H. Marouani and M. R. Dagenais, "Comparing high resolution timestamps in computer clusters," in *Canadian Conference on Electrical and Computer Engineering*, May 2005.

[103] H. Marouani and M. R. Dagenais, "Internal clock drift estimation in computer clusters," *Journal of Computer Systems, Networks, and Communications*, May 2008, [Online] Available: http://www.hindawi.com/journals/jcsnc/2008/583162.html. [Accessed: June 30, 2009].

[104] K.-M. Laventure and L. Malvert, "Lemona: Towards an open architecture for decentralized forensics analysis," Master's thesis, Macquarie University, 2008, [Online]. Available: lemona.googlecode.com/files/lemona-thesis-20081117.pdf. [Accessed: June 30, 2009].