HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et en Communication

par

Gilles Muller

Contribution à la conception de systèmes d'exploitation adaptatifs et extensibles : du micro-noyau à l'évaluation partielle

soutenue le 9 Octobre 1997 devant le jury composé de

M.	Jean-Pierre Banâtre	Président
MM	Roland Balter	Rapporteur
	David Powell	Rapporteur
	Calton Pu	Rapporteur
MM	Michel Banâtre	Examinateur
	Charles Consel	Examinateur

À Marylène,

D'Arras en halte, oh humaine, où Liévin échoue!
Chiale semaine ... y Gilles de Rennes chez Diderot ...
    ouate tout doux.
Chiffrer de sombre eau ... Ouïe d'Aoudh ... Denys
    brette
si huppe bémol sans délit ... Enceinte aime tout bête.

Mots d'heures: gousses, rames. *The d'Antin Manuscript*, Poème 26,
Discovered, edited and annotated by *Luis d'Antin van Rooten*.

Remerciements

Ce document d'habilitation présente un ensemble de travaux de recherche qui ont été effectués chronologiquement au sein des projets INRIA LSP (*Langages et systèmes Parallèles*), SOLIDOR et COMPOSE. Ces recherches ont été concrétisées par la réalisation de machines, systèmes d'exploitation et outils de transformations de programmes qui sont pour la plupart le résultat d'un véritable travail d'équipe. Par conséquent, je tiens en tout premier lieu à remercier tous les chercheurs et ingénieurs avec qui j'ai eu le plaisir de collaborer depuis mon arrivée à l'IRISA.

Jean-Pierre Banâtre a guidé mes premières activités de recherche dans le cadre du projet LSP. Je lui dois plus particulièrement ma formation initiale de chercheur et mon intérêt pour les systèmes distribués. Je lui exprime ma reconnaissance et le remercie de m'avoir fait l'honneur de présider ce jury.

Je remercie vivement Roland Balter et David Powell pour avoir accepté d'être rapporteurs de ce travail. Leurs commentaires avisés sur ce document m'ont été très utiles.

J'ai eu l'opportunité de collaborer avec Calton Pu, responsable du groupe Synthetix à l'OGI, sur le thème de la spécialisation de systèmes. Son regard sur la recherche en système est une source précieuse d'enseignements. Je le remercie pour la confiance qu'il me témoigne en ayant accepté de rapporter sur mon travail.

Michel Banâtre, responsable du projet SOLIDOR, m'a fait partager son intérêt pour les systèmes tolérant les fautes. Nos dicussions intenses ont profondément influé la conduite de mes recherches. Je le remercie d'avoir accepté d'être dans mon jury.

Je remercie Charles Consel, responsable du projet COMPOSE, pour la confiance qu'il m'a accordée. Nos échanges croisés entre les domaines de l'évaluation partielle et des systèmes d'exploitation sont une source constante d'inspiration créatrice. Son enthousiasme communicatif pour la recherche relève parfois du virus (informatique).

Les travaux présentés dans ce document ont été réalisés avec l'aide de plusieurs chercheurs que je tiens également à remercier : C. Bryce, P. Heng, M. Hue, R. Marlet, N. Peyrouze, Y. Prunault, B. Rochat, P. Sanchez, P. Stéfanelli, N. Volanski., et à l'OGI C. Cowan et A. Goel.

Je remercie Frédéric Leleu et Renaud Marlet pour leur soutien amical. Nos discussions épistémologiques ont été une source non négligeable d'inspiration au cours de la rédaction de ce document. Enfin, je souhaite remercier Sue et Peter Lee qui m'ont fait découvrir les poèmes parfois ésotériques de *Luis d'Antin van Rooten*.

Avant-propos

Ce document d'habilitation présente mon parcours scientifique au travers de plusieurs études ayant pour fil conducteur la conception de systèmes adaptatifs et extensibles. J'ai cherché à retracer l'évolution de cette idée, depuis l'expression de son besoin jusqu'à la description de solutions relevant des domaines du système ou des langages. Par conséquent, ce document ne présente pas l'ensemble de mes travaux. Ceux-ci peuvent être résumés dans ce qui suit.

J'ai débuté mon activité de recherche au sein projet INRIA LSP (devenu ensuite SOLIDOR) dans le cadre de mon DEA, puis de ma thèse. Je me suis intéressé à la conception d'architectures tolérant les fautes avec l'objectif de réduire leur coût de mise en œuvre, sans pour autant sacrifier la performance. Ceci m'a amené à proposer une implémentation performante du concept de mémoire stable sous la forme d'une carte matérielle appelée *mémoire stable rapide*. Le support de cette carte a été intégré au système d'exploitation distribué GOTHIC, et a entre autres été utilisé pour la conception d'un sous-système fiable de communication par messages.

Les résultats obtenus dans le cadre de ma thèse m'ont donné l'idée de généraliser l'utilisation de la technologie mémoire stable pour concevoir les différents services d'un système d'exploitation. Plus précisément, mon objectif lors de la conception du système distribué FTM était d'offrir la transparence de la tolérance aux fautes aux applications utilisateurs. Cet objectif a été atteint par la conception d'une méthodologie de construction de systèmes d'exploitation fiables. Cette méthodologie repose sur l'utilisation de la technologie micro-noyau et d'un ensemble extensible de serveurs fiables qui implémentent les points de reprise des applications.

La conception du FTM m'a permis de mieux comprendre l'influence des mécanismes bas-niveaux, tel que les micro-noyaux, sur le comportement global du système. En particulier, l'intégration de ces mécanismes avec l'architecture des réseaux est cruciale pour la performance des systèmes distribués. Ceci m'a amené d'une part à proposer une nouvelle architecture de micro-noyau (DP-MACH) et d'autre part à m'intéresser aux nouvelles technologies de réseaux haut-débit, notamment l'ATM.

Le souci de performance m'a amené ensuite à m'intéresser aux techniques de spécialisation de systèmes d'exploitation par transformation automatique de programmes. Plus précisémement, mes recherches récentes au sein du projet COMPOSE visent à concilier performance et généricité en *adaptant* un système générique à un contexte d'utilisation précis.

Table des matières

1 Introduction **9**
1.1 Taxinomie des systèmes d'exploitation extensibles et adaptatifs 10
1.1.1 Adaptabilité 10
1.1.2 Extensibilité 11
1.2 Présentation de ce mémoire 13

2 Extensibilité et tolérance aux fautes : l'approche FTM **17**
2.1 Conception d'un système tolérant les fautes et de faible coût : l'approche FTM 18
2.1.1 Comment réduire le coût matériel de la tolérance aux fautes 18
2.1.2 Mémoire stable transactionnelle : un support efficace pour la mémorisation des points de reprise 19
2.2 Systèmes d'exploitation extensibles à micro-noyaux : une approche à la mise en œuvre d'un système fiable 19
2.2.1 Points de reprise cohérents bloquants 21
2.3 Description du modèle de systèmes d'exploitation fiable 23
2.3.1 Modèle client-serveur fiable 23
2.3.2 Construction de points de reprise cohérents bloquants 26
2.3.3 Mise en œuvre du modèle FTM au dessus du micro-noyau Mach 3.0 28
2.4 Analyse de performance 29
2.4.1 Environnement de test 29
2.4.2 Calcul de nombres premiers 30
2.4.3 Mp3d 30
2.4.4 Micro-emacs 31
2.5 Leçons et conclusions 32

3 Support noyau pour la protection mémoire à grain fin **35**
3.1 Modèle d'un micro-noyau intégrant les domaines de protection 37
3.2 Mise en œuvre des domaines de protection 38
3.2.1 Modification du micro-noyau Mach 3.0 38
3.2.2 Schéma d'exécution de base du PPC 39
3.2.3 Optimisation du PPC 40
3.3 Évaluation de performance 42
3.4 Conclusion 44

4 Spécialisation automatique de composants système **45**
4.1 Le RPC de Sun et son optimisation 46
4.2 Opportunités de spécialisation dans le RPC Sun 48
4.2.1 Suppression de la sélection entre l'encodage et le décodage 48
4.2.2 Suppression du contrôle de non-débordement 49
4.2.3 Propagation du résultat 49
4.2.4 Bilan des opportunités de spécialisation 50
4.3 Spécialisation automatique au moyen de l'évaluateur partiel Tempo 51
4.4 Évaluation de performance 53
4.5 Discussion 57
4.5.1 Expérience de l'utilisation de Tempo 57
4.5.2 Spécialisation de programmes existants 58
4.6 Conclusion 59

5 Une approche déclarative à la spécialisation de programmes **61**
5.1 Les classes de spécialisation 62
5.2 Spécialisation d'un système de fichier 64
5.3 Compilation des classes de spécialisation 65
5.4 Aspects relatifs au support d'exécution 69
5.5 Conclusion 69

6 Conclusion et perspectives **71**

Chapitre 1

Introduction

Le système d'exploitation est le composant informatique qui encapsule les fonctions de la machine physique. Son rôle est d'offrir une machine virtuelle de niveau suffisamment élevé qui puisse ainsi simplifier la conception et la mise en œuvre des programmes des utilisateurs.

Les fonctions du système se sont continuellement accrues au cours des années, au prix d'une complexité toujours croissante. Au moniteur primitif masquant uniquement les fonctions d'entrée-sorties console et disque, se sont ajoutées progressivement des fonctions de mémoire virtuelle, de multi-tâches, de gestion des réseaux, de partage d'information entre machines, de gestion de l'environnement graphique, pour aboutir récemment à l'intégration du multimedia. Il en résulte que les systèmes d'exploitation modernes intègrent des fonctions de nature très diverses, possédant des contraintes d'implémentation parfois contradictoires.

En effet, cette diversité pose aux concepteurs d'un système, de nombreux problèmes d'efficacité relatifs par exemple à la taille mémoire occupée ou aux conflits entre les différentes politiques de gestion des ressources matérielles de la machine. Or, la contrainte de performance est un aspect essentiel du développement d'un système. Pour l'utilisateur, le système d'exploitation ne représente qu'un moyen et non pas une finalité. On compte ainsi dans un passé récent de multiples exemples d'échecs de systèmes par suite de performances insuffisantes, malgré leur haut niveau de fonctionnalité. Il est clair qu'un système doit maximiser l'utilisation des ressources matérielles vis-à-vis des applications et n'en utiliser pour ses besoins propres que le minimum.

La complexité des fonctionnalités conjuguée à la contrainte de performance font du système d'exploitation l'un des logiciels les plus difficiles à développer et à maintenir. D'un point de vue industriel, il en résulte une nécessité de pérennisation des développements. Ceci est renforcé par le fait que les architectures de machines évoluent de plus en plus rapidement et que les utilisations de l'informatique se sont très largement diversifiées. On trouve aujourd'hui des microprocesseurs et des noyaux de système jusque dans les téléphones cellulaires et les machines à laver. Au contraire des premiers systèmes qui étaient souvent liés à une machine donnée et à un domaine précis d'utilisation, il n'est plus imaginable économiquement de "jeter" un système d'exploitation dès l'obsolescence de

l'ordinateur sur lequel il est implémenté.

1.1 Taxinomie des systèmes d'exploitation extensibles et adaptatifs

Le besoin de systèmes aisément réutilisables a été le moteur de nombreuses recherches au cours de ces quinze dernières années. Le but de ces études a été la définition de systèmes à vocation universelle dont des instances peuvent être utilisées depuis le téléphone cellulaire jusqu'au super-calculateur. Deux propriétés, l'adaptabilité et l'extensibilité, sont caractéristiques de ces systèmes.

1.1.1 Adaptabilité

On considère qu'une application ou un système est **adaptatif** si celui-ci peut modifier son comportement en fonction de son contexte d'exécution. Le contexte regroupe des notions diverses qui peuvent aller de la configuration matérielle jusqu'à des propriétés sur les valeurs d'entrée ou sur les objets manipulés. La motivation essentielle d'un système adaptatif est d'offrir de meilleures performances ou fonctionnalités au cours de la vie du système. Un système adaptatif offre le choix entre différents algorithmes, ce qui permet de satisfaire au mieux les besoins d'un cas précis, tout en conservant une politique générale convenant à la plupart des situations.

De manière générale, le processus d'adaptation repose sur trois étapes : (i) introspection, (ii) analyse et prise de décision, (iii) action/sélection d'une politique :

Introspection. Cette étape est relative à l'observation du contexte. Les mécanismes relatifs à cette étape vont de la simple observation de variables locales jusqu'à des processus de récolte éventuellement distribués. On peut citer par exemple le cas d'un visualiseur MPEG adaptant le débit d'information transmis aux ressources disponibles de bout en bout [26].

Décision. Cette étape est relative à l'analyse des informations récoltées. Cette analyse conduit éventuellement à une prise de décision visant à adapter le système à la situation observée. Les éléments de décision sont par exemple, l'occurrence d'un événement, le dépassement d'un seuil, . . .

Action. Les actions visant à adapter le système sont diverses et dépendent de la nature de l'adaptation. Elles peuvent être relatives au noyau du système, à un processus unique ou à un ensemble de processus. Une action d'adaptation peut consister à modifier quelques paramètres d'exécution dans un cas simple, ou dans un cas plus complexe à sélectionner un nouvel algorithme convenant mieux au contexte observé. Cette sélection peut être qualifiée d'*interne* si les différentes politiques sont présentes dans le code du système qui est alors générique, ou d'*externe* si la sélection de la politique requiert le chargement dynamique d'une extension externe au système.

En fait, il existe de nombreuses variantes de systèmes adaptatifs en fonction de *quand, par qui*, et *comment* sont réalisées les trois étapes du processus d'adaptation :

Système adaptable. Un système est dit adaptable lorsque les étapes de l'adaptation sont contrôlées par une entité (processus ou humain) extérieure au système lui-même. Un système adaptable est par conséquent passif. Au contraire, un système adaptatif est actif puisqu'il possède des capacités internes d'introspection et de décision.

Système configurable. Le processus d'adaptation est réalisable soit en cours d'exécution, de manière dynamique, ou plus simplement avant l'exécution, de manière statique. Un système est dit configurable lorsque l'adaptation est effectuée de manière statique, à la compilation ou à l'édition de lien. La configuration permet d'assurer la portabilité d'un système sur des plate-formes différentes et une meilleure utilisation des fonctionnalités offertes par le matériel. Diverses méthodes de configuration ont été proposées ; celles-ci vont des pré-processeurs jusqu'à l'héritage dans les langages objets. Cette dernière approche est notamment utilisée dans des systèmes tels que Choices [25] et le micro-noyau Chorus [89].

Système spécialisable. La spécialisation est une action particulière d'adaptation qui permet de transformer un programme générique en une version adaptée à un contexte donné. Le gain escompté de la spécialisation est généralement relatif à la performance. Des approches manuelles à la spécialisation sont souvent utilisées pour optimiser des applications possédant de forts goulots d'étranglements (e.g., système, graphique, scientifique). Cependant, ces approches ont le défaut d'être trop spécifiques et sont susceptibles d'introduire des erreurs lors de la spécialisation. Dans la suite de ce document, nous présentons des outils permettant d'automatiser le processus de spécialisation à la compilation et à l'exécution.

La mise en œuvre des phases d'introspection et d'action du processus d'adaptation nécessite que les mécanismes internes du système soient observables, voire remplaçables si les mécanismes standard ne sont pas suffisants. En conséquence, un système adaptatif est souvent implanté au dessus d'un système sous-jacent possédant la propriété d'extensibilité.

1.1.2 Extensibilité

On considère qu'un système est **extensible**, s'il est possible d'y ajouter des fonctionnalités non anticipées lors de la conception. L'extensibilité est une propriété relative à l'architecture logicielle et à la structure d'un système. Le degré d'extensibilité se mesure en fonction de deux paramètres : la durée de vie des extensions et le niveau du système au sein duquel les extensions peuvent être ajoutées.

Les durées de vie possibles pour une extension système sont :

Permanente. Ceci est la forme la plus primitive d'extensibilité. L'extension est liée statiquement avec le reste du système lors de la configuration. Cette solution est classiquement utilisée dans le monde Unix pour permettre l'ajout de pilotes de périphériques.

Semi-permanente. On considère ce cas lorsque l'extension est relative à un service ou à un environnement utilisé par un ensemble d'applications (c.-à-d., personnalité Dos ou Unix). L'extension est alors dynamiquement chargée/déchargée par un utilisateur privilégié (c.-à-d., administrateur). Cette forme d'extensibilité est également utilisée pour charger dynamiquement des pilotes dans Windows95 (c.-à-d., *plug-and-play*) et des systèmes Unix modernes tels que Linux ou Solaris.

Celle de l'application/du service. Ce cas est relatif à l'utilisation d'une extension spécifique à une application donnée. Le chargement/déchargement de l'extension est alors intégré avec le lancement et la fin de l'application. Toutefois, il reste masqué au programmeur de l'application.

Intra-application/service. Ce cas extrême peut être considéré pour une application adaptative. Il est alors nécessaire que le programmeur puisse gérer dynamiquement les extensions système. En conséquence, les primitives de sélection, chargement et déchargement d'extensions doivent alors être accessibles au programmeur.

Le niveau du système au sein duquel peut être ancré une extension conditionne l'étendue des fonctions pouvant être modifiées. D'un point de vue architecture, l'extensibilité nécessite de rendre visibles des interfaces auparavant cachées. La tendance des recherches est à repousser de plus en plus bas le niveau des interfaces exportées. On peut distinguer les niveaux suivants pour le support des extensions :

Processus. Cette solution est utilisée pour mettre en œuvre certains services système qui peuvent être implémentés au dessus de l'interface standard sans recourir à des modifications au sein du noyau. Cette approche est notamment utilisée pour mettre en œuvre certains services distribués dans Unix, tel que NIS [85] ou le serveur du système de fichier distribué NFS [91].

Serveur. Cette solution a été introduite par les systèmes à base de micro-noyaux tels que Mach [1], Chorus [89] et Amoeba [66]. L'objectif poursuivi lors de la conception de ces systèmes est de réduire le nombre de services implantés au sein du noyau et de les exporter vers des processus spéciaux appelés serveurs. Les services implémentés par des serveurs sont par exemple la gestion de fichiers, la communication réseaux, la personnalité Unix, ... Le micro-noyau peut lui-même être considéré comme un serveur particulier offrant les notions d'espace d'adressage, de fil de contrôle, d'objet mémoire et de message. Le message est le support de communication uniforme entre le micro-noyau et les serveurs. Dans un système reposant sur micro-noyau, le grain d'utilisation de l'extensibilité est le serveur.

Mécanismes internes du noyau. Les principales limites des micro-noyaux sont (i) le manque de performance introduit par la communication par messages, (ii) la granularité relativement importante des extensions et (iii) l'impossibilité de redéfinir les fonctions même du micro-noyau. Cette dernière limitation est particulièrement contraignante lors de la mise en œuvre de fonctionnalités de type multimedia qui imposent une gestion des ressources avec contraintes de réservation [5, 34]. Pour lever ces limitations, des systèmes récents tels que SPIN [16] et l'exokernel Aegis [40] offrent la possibilité de charger des extensions directement au sein du noyau

et permettent la redéfinition de fonctions de bas-niveau tels que l'ordonnancement ou l'allocation mémoire.

Il est à noter que les formes les plus fines d'extensibilité soulèvent de nouveaux problèmes relatifs à la sûreté des extensions, notamment lorsque celles-ci peuvent être chargées au sein du noyau. Pour prévenir une modification non contrôlée du noyau, diverses solutions ont été suggérées. Wahbe *et al.* [103] proposent d'insérer des instructions dans le code assembleur vérifiant la validité des accès mémoire. Cette technique est utilisée dans l'exokernel Aegis [40]. En ce qui concerne SPIN, Bershad *et al.* [16, 53] utilisent le typage fort de Modula-3. Plus récemment, Lee et Necula ont décrit une autre technique reposant sur l'utilisation de preuves associées au code (*proof-carrying-code*) [78, 77]. L'apport de cette dernière technique est de proposer une approche générale à la vérification de propriétés.

1.2 Présentation de ce mémoire

Ce document présente une synthèse de nos travaux dans le domaine des systèmes extensibles et adaptatifs. Les problèmes que nous adressons, et donc les solutions, relèvent de domaines différents qui vont des noyaux de système jusqu'au support langage.

Extensibilité et tolérance aux fautes

Le besoin d'extensibilité nous a été initialement posé lors de la conception du système tolérant aux fautes FTM [67]. La conception d'un tel système pose un problème double : (i) comment assurer la tolérance aux fautes et (ii) comment mettre en œuvre un tel système sans pour autant tout concevoir à partir de la machine nue. Ce dernier problème est particulièrement important dans le contexte d'un système réel, puisqu'il est aujourd'hui presque impossible de redévelopper un système propriétaire pour des raisons de coût et de compatibilité avec l'existant.

La solution que nous avons retenue repose sur la propriété d'extensibilité apportée par le concept de micro-noyau. Plus précisément, nous étendons le système avec des *serveurs fiables* qui assurent la continuité de service. Un serveur fiable gère une ressource système et est capable, après une défaillance, de reprendre son exécution à partir d'un point de reprise local. La cohérence globale du système est assurée par un modèle à points de reprise cohérents bloquants. La caractéristique principale de ce modèle est de préserver la propriété d'extensibilité du système : de nouveaux serveurs fiables peuvent être ajoutés dynamiquement. Par ailleurs, nous bénéficions des autres avantages apportés par les micro-noyaux, notamment en matière de portabilité et d'indépendance vis-à-vis de la machine.

Domaines de protection

La conception d'un système complexe comme le FTM, et sa mise en œuvre, sont des étapes nécessaires dans l'acquisition de la connaissance des problèmes réels qui se posent au sein d'un système d'exploitation. Notamment, le fait que le grain d'extension au dessus

du micro-noyau soit limité à la notion de serveur pose un problème de performance. Celui-ci est dû principalement à la communication par messages et au besoin de conversion des structures de données entre les espaces d'adressages des différents serveurs.

Ceci nous a amené à proposer un nouveau modèle de micro-noyau dans lequel les notions d'espace de protection et d'espace d'adressage sont découplées l'une de l'autre. En d'autres termes, un processus peut être composé de plusieurs *domaines de protection* partageant un même espace d'adressage. Le bénéfice majeur de ce modèle de noyau est que la communication entre deux domaines de protection est plus performante qu'entre deux serveurs car elle ne nécessite pas de conversion d'adresses. Par ailleurs, l'unicité de l'espace d'adressage permet également de réaliser des optimisations complémentaires au niveau de l'unité de gestion mémoire (MMU), lors du changement de domaines de protection.

La structuration d'une application en plusieurs domaines de protection est une alternative à l'utilisation de serveurs lorsque ces derniers ne sont pas partagés par plusieurs applications. Plus généralement, le domaine d'application des domaines de protection est la construction d'applications à partir de composants logiciels ne possédant pas le même niveau de confiance. Notamment, certains domaines de protection peuvent posséder l'attribut superviseur et sont ainsi à même d'accéder directement au noyau et au matériel. De ce fait, les domaines de protection superviseurs sont une solution élégante au support d'extensions systèmes spécifiques à une application. Ce modèle de micro-noyau a donné lieu à une implémentation, DP-Mach [22], qui a été réalisée par modification et extension du micro-noyau Mach. Outre un apport en terme de performance sur le changement de domaines de protection, ce travail montre que la mise en œuvre des domaines de protection peut être faite à partir d'un système standard avec relativement peu de modifications au sein du noyau.

Spécialisation automatique de composants systèmes

Dans un système reposant sur l'utilisation généralisée d'extensions système spécifiques aux applications, le problème qui se manifeste est l'automatisation de la mise en œuvre de telles extensions. L'écriture manuelle de composants *ad hoc* est bien sûr possible, mais cette solution ne peut être raisonnablement utilisée que par des concepteurs d'applications experts en système. Pour résoudre ce problème, nous avons proposé une solution reposant sur la spécialisation automatique de composants génériques. À ce titre, nous nous sommes plus particulièrement intéressés à l'application de la technique de l'*évaluation partielle*.

L'évaluation partielle [31] est une technique de transformation qui a pour but de spécialiser automatiquement un programme en fonction d'un sous-ensemble connu des données d'entrée (appelées *données statiques*). C'est une transformation de programmes qui préserve la sémantique du programme original dans la mesure où le programme spécialisé (appelé également *résidualisé*), appliqué aux données manquantes (dites *dynamiques*, c'est-à-dire inconnues), produit le même résultat que le programme original appliqué à toutes les données (cf figure 1.1).

Afin de permettre la spécialisation de composants système, nous avons participé à la conception de Tempo [32], un évaluateur partiel pour programmes C. Tempo a été spécifiquement conçu pour le domaine des programmes système ; c'est-à-dire que la précision des analyses mises en œuvre dans Tempo est dictée par les modèles de programmation que l'on

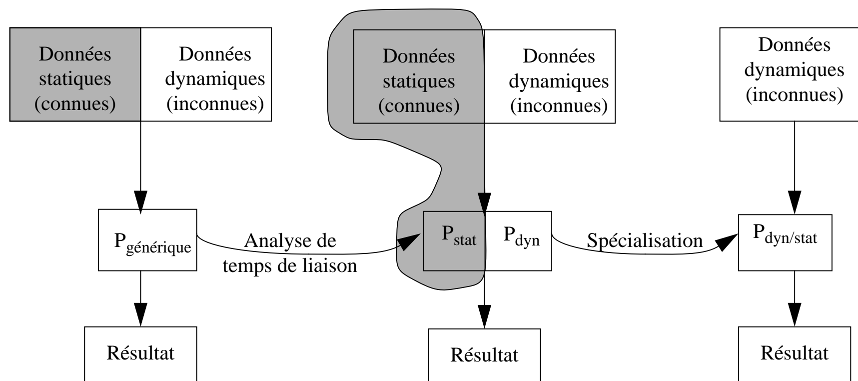

FIG. 1.1 – *Évaluation partielle*

rencontre dans les programmes système. Plus précisément, nous avons validé Tempo, via la spécialisation de la mise en œuvre de l'appel de procédure à distance (c.-à-d., RPC) développé par Sun en 1984. De part son architecture, le RPC de Sun recèle de multiples sources de généricité, ce qui en fait un très bon candidat pour la spécialisation ; nous obtenons un gain de performance allant jusqu'à 1,3 sur un appel complet (réseau inclus). Sur l'encodage des données au sein du talon, le facteur d'optimisation va jusqu'à 3,7 [75, 74, 69]. Les retombées de cette étude sont multiples. Les résultats obtenus montrent : (i) que la spécialisation automatique est une technique permettant la mise en œuvre performante de composants systèmes génériques, (ii) que le programmeur système peut se libérer en partie de la contrainte d'efficacité et programmer de manière plus générique.

Approche déclarative à la spécialisation

De manière générale, on peut distinguer plusieurs types de spécialisation : à la compilation, à l'exécution et une approche intermédiaire qui consiste à sélectionner à l'exécution une instance parmi plusieurs versions pré-spécialisées à la compilation. Tempo offre une approche uniforme à ces différentes formes de spécialisation en séparant l'analyse du programme de la spécialisation effective.

Dans le cas du RPC, la gestion de la spécialisation est relativement simple puisque celle-ci est effectuée uniquement à la compilation et qu'il n'existe qu'une version spécialisée du programme. Dans un cadre plus général, lorsque différents types de spécialisation sont mélangés, de nouveaux problèmes apparaissent : comment détecter les opportunités de spécialisation ? Sur quelle partie du programme faire porter la spécialisation ? Quand activer et désactiver une instance spécialisée ? Etc. Bien évidemment, la gestion de la spécialisation peut être programmée directement en C. Toutefois, c'est une tâche complexe, difficile à maîtriser par un programmeur non expert, qui demande une intrusion au sein du programme original et qui est par conséquent potentiellement source d'erreurs.

Pour simplifier la gestion de la spécialisation, nous avons introduit une approche déclarative dans le contexte de la programmation orientée objet [102]. Plus précisément, le

programmeur déclare quelles méthodes doivent être spécialisées et pour quels contextes. À partir de ces déclarations, un compilateur détermine comment les versions spécialisées sont produites et gérées. Le résultat de la compilation est une version étendue du programme source, capable d'activer la spécialisation lorsque nécessaire, et de remplacer les versions spécialisées d'une manière transparente.

Dans notre approche, l'unité de déclaration est la *classe de spécialisation*. Elle enrichit l'information concernant une classe existante. Le rapport entre les classes normales et les classes de spécialisation est défini par une forme d'héritage reposant sur les *classes de prédicats* introduites par C. Chambers [27]. Les bénéfices de cette approche sont (i) que la déclaration n'est pas intrusive et (ii) qu'il est possible d'exprimer une spécialisation de type incrémental [33] par héritage de classes de spécialisation.

Une implémentation d'un compilateur des classes de spécialisation a été réalisée pour le langage Java [46]. Ce compilateur prend en entrée du source Java étendu avec les classes de spécialisation et produit du Java standard. Le code Java est ensuite compilé vers du C, au moyen d'un traducteur de code intermédiaire (*bytecode*) [70]. Finalement, le code C produit peut être spécialisé au moyen de Tempo.

Structure du document

La structure de ce mémoire est la suivante. La description du modèle FTM, l'étude des performances du prototype et la présentation des leçons de cette expérimentation font l'objet du chapitre 2. La conception et l'évaluation de DP-Mach sont décrites dans le chapitre 3. Tempo et son utilisation pour spécialiser le RPC de Sun sont présentées dans le chapitre 4. Les classes de spécialisation et leur implémentation pour le langage Java sont décrites dans le chapitre 5. Nous concluons dans le chapitre 6 en décrivant les perspectives soulevées par nos études.

Chapitre 2

Extensibilité et tolérance aux fautes : l'approche FTM

Un système informatique tolérant les fautes est défini par sa capacité à délivrer un service continu à ses utilisateurs même si un ou plusieurs de ses composants internes sont défaillants. La tolérance aux fautes est bien évidemment une propriété hautement désirable et la réponse naïve à la question "*qui a besoin d'un système tolérant les fautes ?*" est "*tout le monde*". Malheureusement, la mise en œuvre de la tolérance aux fautes repose sur des techniques de redondance [62] qui induisent des surcoûts en matériel ainsi qu'en temps d'exécution. En conséquence, relativement peu de systèmes informatiques parmi tous ceux utilisés quotidiennement sont tolérants aux fautes. Si la conception d'un système informatique est toujours la recherche d'un compromis entre le coût et la performance, la conception d'un système tolérant les fautes est encore plus complexe car le compromis recherché doit tenir compte du coût, de la performance et du niveau de fiabilité que l'utilisateur estime nécessaire pour le support de ses applications.

Ce chapitre présente la conception de FTM (*Fault Tolerant Multiprocessor*) [1], un système tolérant les fautes à usage général [13, 67]. L'objectif principal de cette étude était de construire un système de "faible coût" qui puisse être utilisé sur des stations de travail standard. Plus spécifiquement en matière de système, notre motivation était la création d'une méthodologie de conception de systèmes d'exploitation fiables et extensibles offrant la transparence de la tolérance aux fautes aux applications utilisateurs. En d'autres termes, porter une application sur FTM n'implique qu'une recompilation du code source, sans modification de celui-ci. Ces différents objectifs sont atteints par l'utilisation du micro-noyau Mach et d'un ensemble extensible de serveurs fiables qui implémentent les points de reprise des applications et offrent la continuité de service en dépit des défaillances des machines.

L'étude de FTM s'est effectuée en collaboration avec BULL et s'est étendue sur cinq années. Elle a entre autres recouvert la conception et la réalisation d'un prototype. De multiples aspects ont été abordés, allant de la conception de machines, la réalisation de cartes

1. La signification initiale de l'acronyme FTM a perdu par la suite une partie de sa signification, du fait de l'évolution de l'architecture matérielle vers des stations de travail distribuées. FTM a été développé dans le cadre d'un projet commun INRIA-BULL, financé en partie par la DRET via le contrat 90 346.

matérielles, et bien sûr la conception de systèmes d'exploitation. Dans ce chapitre, nous nous concentrons uniquement sur la présentation des aspects système. Le lecteur intéressé par la totalité des recherches réalisées pourra se référer aux documents [12, 67]. Ces recherches ont été effectuées en collaboration avec de nombreux chercheurs et étudiants, notamment : Jean-Pierre Banâtre, Michel Banâtre, Pack Heng, Mireille Hue, Nadine Peyrouze, Yves Prunault, Bruno Rochat, Patrick Sanchez et Patrick Stéfanelli.

Le reste de ce chapitre est structuré de la manière suivante. La section 2.1 est consacrée à la présentation globale de FTM. La section 2.2 décrit la problématique de la construction d'un système tolérant aux fautes et la solution choisie reposant sur le concept de micro-noyau. La section 2.3 présente le modèle de système fiable FTM. la section 2.4 décrit les performances du prototype construit. Nous concluons dans la section 2.5.

2.1 Conception d'un système tolérant les fautes et de faible coût : l'approche FTM

Notre objectif de construction d'un système tolérant les fautes et de faible coût a dicté les choix de conception de FTM, tant au niveau de l'architecture que du système d'exploitation. Nous détaillons maintenant ces choix.

2.1.1 Comment réduire le coût matériel de la tolérance aux fautes

Le surcoût matériel de la tolérance aux fautes dépend principalement du degré de réplication des composants du système et de la spécificité des composants employés. En conséquence, la construction d'un système tolérant les fautes et de faible coût impose la recherche de solutions minorant le degré de réplication des machines ; ceci a été réalisé de deux façons : (i) nous avons choisi de tolérer une seule défaillance matérielle à un instant donné, (ii) nous avons décidé de faire reposer le système d'exploitation sur un modèle à réplication passive des calculs et sur l'utilisation de points de reprise [38, 73]. Le premier choix est justifié par le fait que la fiabilité des stations de travail classiques a considérablement augmenté ces dernières années et que les défaillances matérielles deviennent relativement rares. De fait, la probabilité que deux stations défaillent au même moment est suffisamment faible pour être acceptable dans un environnement de réseau local. Notre second choix offre l'avantage qu'en mode de fonctionnement normal toutes les machines du réseau effectuent des travaux différents. En revanche, en cas de défaillance d'une station, les travaux qui s'exécutaient sur celle-ci sont relancés sur une machine secours à partir d'un point de reprise précédemment sauvegardé. Dans ce cas, on peut noter que la machine de secours exécute les travaux de la station défaillante en plus des siens.

Enfin, nous avons choisi de construire l'architecture FTM à partir de machines standard tel que des PC, des stations de travail ou des cartes industrielles (e.g, Multibus, VME). Cette décision a pour conséquence que la détection des défaillances repose soit sur la non-réponse de la machine à une requête extérieure (ex., détection de crash par chien de garde) ou sur des mécanismes matériels intégrés à la machine (ex., ECC, bit de parité). Dans ce dernier cas, la détection d'une erreur provoque l'arrêt de la machine par le système d'exploitation.

2.1.2 Mémoire stable transactionnelle : un support efficace pour la mémorisation des points de reprise

L'implémentation des techniques de points de reprise requiert la mémorisation de ceux-ci sur un support qui résiste aux défaillances des machines. La solution la plus usuellement utilisée est celle proposée par B. Lampson consistant à construire une mémoire stable à partir de disques [60]. Cette approche est appropriée lorsqu'il est nécessaire de gérer des structures relatives aux systèmes de fichiers (ex., méta-data), mais se révèle inefficace lorsqu'il est nécessaire de manipuler de petites structures telles que celles utilisées au sein d'un système d'exploitation.

La contrainte d'efficacité nous a amené à développer une nouvelle technologie de mémoire stable, appelée STM (c.-à-d., Stable Transactional Memory). STM est le produit de l'évolution du concept de mémoire stable rapide développé à l'IRISA au cours de précédentes études sur les systèmes distribués fiables : Enchère [7] et Gothic [8, 9, 11]. La STM repose sur l'utilisation de bancs de mémoire vive (i.e., RAM) situés dans deux machines indépendantes pour des raisons évidentes de disponibilité. La STM offre les deux entités de base suivantes : l'objet stable et la transaction. Plus précisément, un objet stable est défini comme une suite de mots mémoire contigus et une transaction comme un ensemble atomique d'opérations primitives sur des objets stables. Afin que les transactions et les objets STM puissent être utilisés aussi simplement que des objets C++ standard, nous avons conçu une bibliothèque d'interface C++ qui masque l'implémentation effective de la STM au programmeur [72]. Deux versions de la STM ont été successivement étudiées au cours du projet : une version matérielle offrant une protection à grain fin des objets stables et une version logicielle ne demandant pas de développement matériel spécifique et pouvant de ce fait être facilement portée sur des machines différentes. Une description détaillée des deux versions de la STM peut être trouvée dans [67].

Une STM peut être vue logiquement comme une mémoire possédant deux ports d'accès indépendants (voir figure 2.1), ce qui permet de construire un site stable en regroupant autour d'une STM, une machine primaire et une machine de secours. En mode de fonctionnement normal, le processeur de la machine primaire possède un accès exclusif à la STM et y mémorise les points de reprise de ses processus. En cas de défaillance, le processeur de la machine de secours avorte les transactions en cours afin de restaurer un état cohérent des objets stables (et des points de reprise), puis relance les processus interrompus à partir de leur précédent point de reprise. Il est à noter que les sites stables sont regroupés par paires : la machine primaire d'un site stable est la machine de secours de l'autre site stable et vice versa.

2.2 Systèmes d'exploitation extensibles à micro-noyaux : une approche à la mise en œuvre d'un système fiable

Nos objectifs de conception pour le système d'exploitation de FTM étaient de satisfaire les trois exigences suivantes : (i) offrir la tolérance aux fautes de manière transparente à l'utilisateur, (ii) fiabiliser l'ensemble des ressources du système, (iii) assurer la portabilité du système. La transparence de la tolérance aux fautes pour le programmeur d'applications

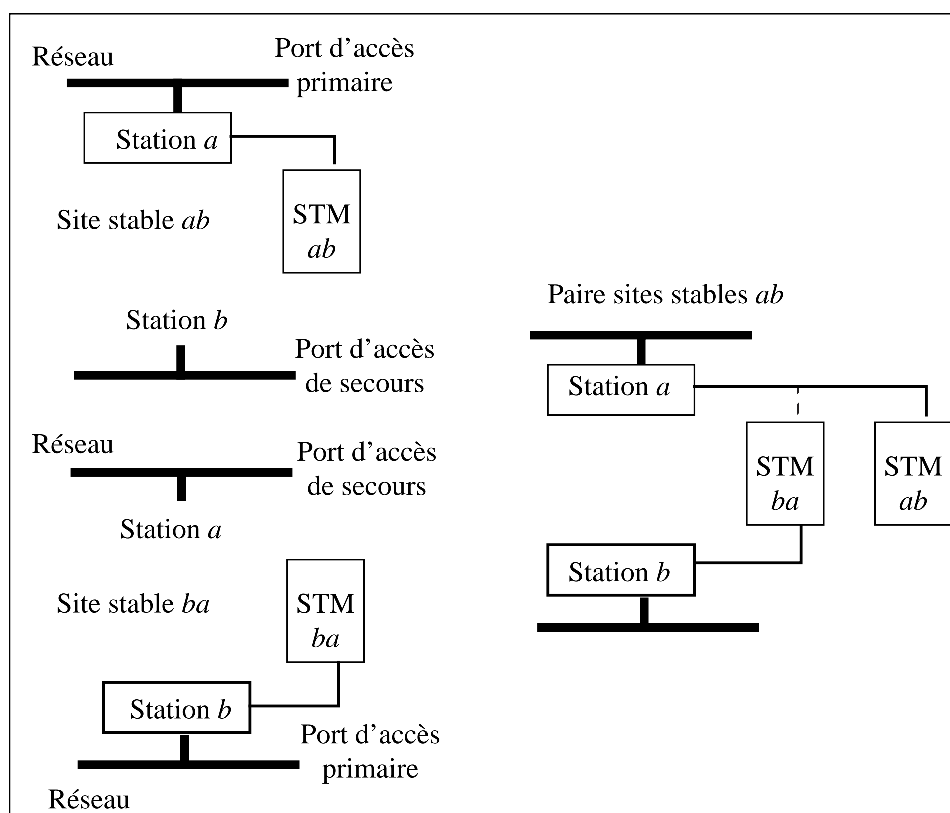

FIG. 2.1 – *Paire de sites stables*

lui permet de concevoir ou de ré-utiliser une application existante sans avoir à modifier celle-ci ou à écrire du code relatif à la gestion d'éventuelles défaillances. Il est à noter que la transparence est d'ores et déjà procurée par de nombreuses techniques de points de reprise à vocation de calcul scientifique [19, 39]. Cependant, la plupart de ces techniques ne gère uniquement que la ressource mémoire des processus et néglige les autres ressources systèmes comme les fenêtres et les fichiers ; ces approches ne permettent pas de fiabiliser des applications à vocation générale. En conséquence, notre seconde exigence était de permettre l'extension de la technique des points de reprise à l'ensemble des ressources et services systèmes. Ceci a été réalisé par la conception d'une méthodologie de construction de systèmes d'exploitation extensibles fiables.

Avant l'introduction de la technologie micro-noyau, l'intégration dans un système d'une nouvelle fonctionnalité telle que la tolérance aux fautes, ne pouvait être effectuée que de deux façons : (i) construire un nouveau système à partir d'une machine nue, (ii) modifier un système monolithique. La première solution est certes performante mais induit une charge de travail très importante. De plus, cette approche souffre d'une restriction de portabilité

car le système résultant n'est pas forcément réutilisable sur de nouvelles générations de machines. Modifier un système existant permet de résoudre certains problèmes de portabilité puisque les ingénieurs sont libérés des problèmes bas-niveau induits par la machine. Toutefois, le système développé reste dépendant des changements de version. Par ailleurs, les interactions entre les différentes parties d'un système monolithique ne sont pas toujours parfaitement claires et définies ; l'ajout et la mise au point de nouvelles fonctions sont donc malaisés.

Les micro-noyaux [1, 66, 89] ont ouvert une troisième voie : ils offrent une machine virtuelle bas-niveau et permettent la conception de systèmes d'exploitation modulaires et extensibles. Les systèmes d'exploitation reposant sur des micro-noyaux sont généralement structurés suivant un modèle client-serveur, c'est-à-dire que le système est composé d'un ensemble de serveurs implémentant chacun un service système. Il est possible d'ajouter de nouveaux services, au moyen de nouveaux serveurs, sans avoir à changer le noyau. Le portage de ce type de systèmes est simplifié puisque seul le micro-noyau doit être adapté à une nouvelle machine. Les services existants sont également préservés en cas d'évolution du micro-noyau, du moins tant que l'interface n'est pas modifiée.

Étant donné les avantages de la technologie des micro-noyaux, nous avons décidé d'implémenter la tolérance aux fautes, tout en préservant les avantages de cette technologie. En particulier, la préservation de l'extensibilité a entraîné la modification du classique modèle client-serveur pour y intégrer des mécanismes support de la tolérance aux fautes. Dans FTM, ceci a été réalisé par la conception de serveurs fiables qui implémentent des services systèmes continus (voir figure 2.2). Un serveur fiable gère une ressource système et est capable, après une défaillance, de reprendre son service à partir d'un point de reprise local mémorisé en STM. De ce fait, un point de reprise d'une application utilisateur est constitué d'une collection de points de reprise locaux des serveurs fiables ayant été utilisés par le processus de l'application (ex., serveur de segments mémoire, serveur de fenêtres, serveur de fichiers). La section suivante explique les raisons qui nous ont amené à retenir un modèle à points de reprise cohérents bloquants pour sauvegarder les points de reprise des applications utilisateurs.

2.2.1 Points de reprise cohérents bloquants

La sauvegarde de points de reprise est un sujet qui a été largement étudié au cours des années récentes. Les différentes techniques peuvent être classées en deux groupes : les méthodes reposant sur une sauvegarde indépendante (ou asynchrone) des points de reprise et les méthodes reposant sur une sauvegarde cohérente (ou synchrone). Dans les approches de type indépendant, les processus prennent des points de reprise sans se coordonner mutuellement. Les messages échangés sont éventuellement rejoués après une défaillance. Les défauts de ces approches sont qu'elles sont soit sensibles à l'effet domino [17, 104], soit que le ré-envoi de messages requiert des processus déterministes [19, 39, 56, 98]. Assurer le comportement déterministe d'un processus requiert l'élimination des sources de non-déterminisme au sein du système, telles que le parallélisme entre processus, les interruptions et les entrées-sorties projetées en mémoire [45]. En conséquence, la mise en œuvre de processus déterministes n'est pas toujours possible dans le cas général. De plus, les solutions sont dépendantes du matériel et ne sont pas applicables à tous les types de services

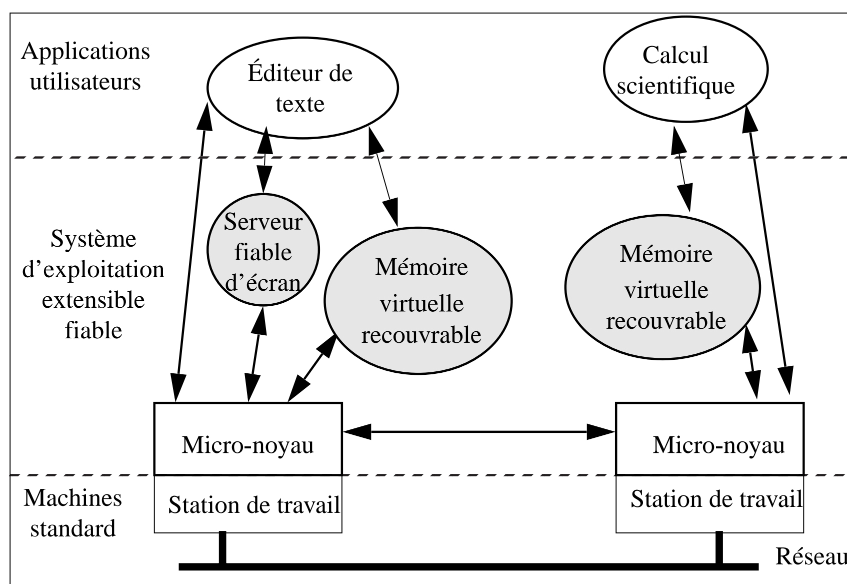

FIG. 2.2 – *Structure d'un système d'exploitation FTM*

système.

Avec les méthodes de sauvegarde de points de reprise cohérents, les processus se coordonnent lors de la sauvegarde d'un état local afin de garantir la cohérence de celui-ci [28]. Le recouvrement après une défaillance peut toujours être effectué à partir du point de reprise sauvegardé et ne nécessite pas de rejouer les échanges de messages. Nous avons retenu cette approche car elle ne nécessite pas de gérer les sources de non-déterminisme et peut ainsi être mise en œuvre sur tout type de machines (mono ou multiprocesseur) sans imposer de contraintes sur le micro-noyau. Les techniques de points de reprise cohérents sont elles-même divisées en deux sous-groupes : les techniques bloquantes et non-bloquantes. Dans les techniques bloquantes [59, 63, 99], les processus s'interrompent et se synchronisent avant de sauvegarder leur point de reprise. Afin de réduire la durée d'arrêt, plusieurs études ont cherché comment réduire le nombre de processus (et processeurs) et le nombre de messages échangés [2, 59].

Les méthodes de type non bloquant [36, 39, 64, 94] ont été appliquées avec succès aux applications scientifiques reposant sur des échanges de messages. Lorsqu'un processus prend (ou reçoit) la décision de sauvegarder un point de reprise, il sauvegarde un point de reprise temporaire et reprend son exécution. Les points de reprise temporaires deviennent définitifs ultérieurement lorsqu'il est certain que tous les processus ont sauvegardé leur point de reprise temporaire et qu'aucun message n'est en transit. Cependant, lorsqu'il s'agit de concevoir un système d'exploitation, les techniques non bloquantes rendent difficile voire impossible la gestion de ressources systèmes telles que les segments mémoire : les processus des applications doivent être stoppés pour assurer que les registres processeurs

sont cohérents avec les segments. Il faut également noter que les techniques non bloquantes sont plus complexes à mettre en œuvre que les techniques bloquantes notamment en raison de la nécessité d'implémenter un protocole de détection de terminaison de la sauvegarde et la journalisation des messages lors de l'exécution du protocole. Pour les raisons précédentes nous avons choisi un modèle à points de reprise cohérents bloquants.

Lors de la conception du modèle FTM, une attention particulière a été portée sur l'atténuation du défaut des modèles à point de reprise bloquant, à savoir la suspension des processus. Deux stratégies ont été étudiées [68, 88] : (i) réduction du nombre de serveurs engagés dans un point de reprise, (ii) minimisation de la taille des données sauvegardées. Les deux sections suivantes détaillent plusieurs aspects de notre expérimentation : la mise en œuvre du modèle client serveur et le protocole de sauvegarde de points de reprise.

2.3 Description du modèle de systèmes d'exploitation fiable

Cette section est consacrée à une description détaillée du modèle de construction de systèmes d'exploitation fiable. Nous présentons successivement le modèle client-serveur fiable, la construction de points de reprise cohérents bloquants et enfin la mise en œuvre du modèle sur le micro-noyau Mach 3.0.

2.3.1 Modèle client-serveur fiable

Le modèle FTM repose sur un ensemble de processus serveurs fiables accédés par des processus utilisateurs [88]. Un serveur abstrait une ressource système (ex., segment mémoire, fichier, écran) et est défini par son interface d'accès et son état interne. Les communications entre les clients et serveurs suivent le modèle d'appel de procédure à distance (*Remote Procedure Call*) [79].

Un processus (utilisateur ou serveur) est l'exécution d'un programme séquentiel. Le code d'un processus est composé d'une séquence d'instructions qui, soient modifient l'état interne, soient effectuent des RPC. L'état interne est composé des registres processeur, de la pile du processus et de structures de données statiques. Pour mettre en œuvre la tolérance aux fautes, nous associons à chaque processus un état persistant (c.-à-d., un point de reprise local) résistant à une défaillance unique de machine. Les états persistants des processus sont mémorisés en STM, que les processus soient utilisateurs ou serveurs. Cependant, la forme sous laquelle ils sont conservés diffère pour une raison de réduction du volume de l'état persistant et donc de la durée de sauvegarde du point de reprise.

État persistant d'un serveur fiable. Un serveur possède un comportement cyclique (voir figure 2.3). Il attend les appels, les traite et retourne leur résultat. Lorsque le serveur est en attente de réception d'un appel, la pile et les registres processeurs ont une valeur fixe qui peut être aisément reconstruite par ré-exécution du code d'initialisation. Lors de la sauvegarde d'un point de reprise global cohérent, la sauvegarde de l'état local persistant du serveur est obligatoirement effectuée lorsque celui-ci est en attente de réception d'appel. Ceci nous permet de réduire l'état persistant du serveur uniquement aux structures de données statiques et de les implémenter directement par des objets stables en STM.

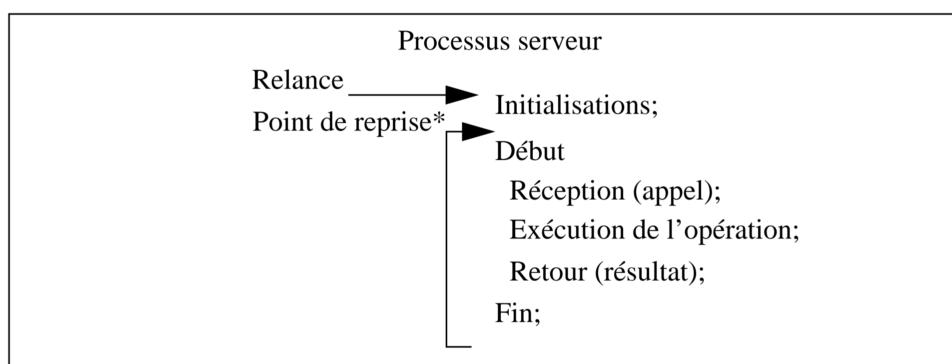

FIG. 2.3 – *Structure d'un serveur fiable*

Pour relancer un serveur fiable après une défaillance, les structures statiques sont d'abord restaurées dans un état cohérent, éventuellement en avortant la transaction qui les gère. À partir de la valeur de ces structures, le code d'initialisation est à même de détecter si l'exécution est relative à une relance après défaillance et de décider du traitement à effectuer.

État persistant d'un processus utilisateur. Aucune hypothèse ne peut être effectuée au sujet du comportement d'un processus utilisateur. En conséquence, l'état persistant d'un processus utilisateur doit contenir les registres du processeur et les différents segments mémoire. Une solution naïve pour sauvegarder ces segments serait d'allouer un gros objet stable. Cependant, une telle solution serait inefficace puisqu'un volume important de données non modifiées devrait être recopié à chaque point de reprise. Pour minimiser ce volume de données, nous avons utilisé une technique reposant sur la gestion de la mémoire virtuelle et la MMU du processeur. Cette approche a été introduite dans le système Targon [19]. Elle permet de déterminer l'ensemble des pages mémoire ayant été modifiées depuis le dernier point de reprise et qui doivent être recopiées dans l'état persistant en STM. Comme les micro-noyaux permettent de spécialiser le comportement de la mémoire virtuelle via la notion de paginateurs, cette technique peut être utilisée sans nécessiter de modification du micro-noyau. L'implémentation est réalisée par le moyen d'un serveur paginateur fiable qui gère les pages sous la forme d'objets stables [88].

Élément de récupération. Un élément de récupération est la séquence d'instructions exécutées par un processus (serveur ou utilisateur) depuis la sauvegarde du précédent état persistant. Un élément de récupération permet de nommer de manière unique l'incarnation courante d'un processus. La validation d'un élément de récupération résulte en la sauvegarde d'un état persistent (c.-à-d., point de reprise local) et la continuation du processus avec un nouvel élément de récupération. L'avortement d'un élément de récupération se traduit par la restauration de l'état persistant précédent et la relance du processus avec un nouvel élément de récupération. Une transaction STM est associée à chaque élément de récupération d'un processus utilisateur : les objets stables de l'état persistant du serveur sont

manipulés au sein de cette transaction ; la validation (resp. l'avortement) de l'élément de récupération entraîne la validation (resp. l'avortement) de la transaction.

Dépendances. Pour construire un point de reprise cohérent distribué, il est nécessaire de tracer les dépendances qui peuvent résulter de l'exécution d'une opération par un serveur pour le compte d'un client pouvant être lui-même un client ou un autre serveur. Plus précisément, les dépendances sont associées aux éléments de récupération des processus :

Un élément de récupération Er_2 dépend d'un élément de récupération Er_1 noté ($Er_1 \rightarrow Er_2$), si l'avortement de Er_1 implique l'avortement de Er_2. En d'autres termes, si Er_2 valide alors Er_1 ne peut plus avorter ultérieurement. Pour garantir la validité de la dépendance $Er_1 \rightarrow Er_2$, il est nécessaire de valider Er_1 avant Er_2. Par rapport à la dépendance $Er_1 \rightarrow Er_2$, Er_1 est appelé le prédécesseur de Er_2 et Er_2 le successeur de Er_1.

Marquage des dépendances Le programmeur d'un serveur doit associer l'un des deux attributs, $read$ ou $update$, à chaque opération du serveur ; cet attribut spécifie si l'opération modifie ou non l'état abstrait de la ressource système gérée par le serveur. Il est à noter que certaines opérations peuvent modifier la représentation réelle sans modifier l'état abstrait. Par exemple, la lecture d'un fichier remplit le cache en mémoire, mais laisse le fichier inchangé. En conséquence, les attributs des opérations ne sont pas propagés au travers d'appels imbriqués. Les situations suivantes peuvent résulter de l'appel d'une opération :

1. l'opération met à jour l'état du serveur,

2. l'opération consulte l'état du serveur, cet état ayant été modifié au sein de l'élément de récupération,

3. l'opération consulte l'état du serveur, cet état n'ayant pas été modifié au sein de l'élément de récupération.

Dans la première situation, l'état du serveur est modifié par l'opération. De ce fait, en cas de défaillance ultérieure du client, l'opération du serveur doit être défaite ; il en résulte la dépendance $Er_{client} \rightarrow Er_{server}$. De même, une défaillance du serveur annule les modifications effectuées par le client ; nous avons également la dépendance $Er_{server} \rightarrow Er_{client}$. Dans la seconde situation, le client consulte une version non persistante de l'état du serveur. Si ce dernier a une défaillance, l'état observé est perdu et par conséquent l'élément de récupération du client doit être avorté ; il en résulte une dépendance $Er_{server} \rightarrow Er_{client}$. Par rapport à la première situation, une défaillance du client n'affecte pas le serveur puisqu'aucune modification de son état n'a été effectuée par le client. Dans le troisième scénario, le client consulte l'état persistant du serveur. Même si ce dernier a une défaillance ultérieure, l'état observé existe toujours ; aucune dépendance n'est engendrée.

À chaque élément de récupération est associé un graphe local mémorisé en STM contenant l'ensemble des éléments de récupération dont il dépend. Le marquage des dépendances est intégré au RPC ; au retour de l'opération, si une dépendance est créée, celle-ci est ajoutée aux graphes du client et du serveur.

2.3.2 Construction de points de reprise cohérents bloquants

Notre exigence principale pour l'algorithme de sauvegarde de point de reprise était que le programmeur puisse être libre d'initialiser la sauvegarde d'un point de reprise. En conséquence, lancer la sauvegarde d'un point de reprise peut être effectué par tout élément de récupération qu'il soit un serveur ou une application utilisateur. Dans la suite, nous désignons cet élément de récupération comme l'initiateur. Pour construire un état global cohérent, les dépendances mémorisées dans tous les graphes sont utilisées pour calculer l'ensemble des éléments de récupération qui dépendent de l'initiateur ; ceci est effectué par un algorithme de poursuite. L'ensemble résultant est ensuite regroupé en une action atomique distribuée, l'initiateur devenant alors le coordinateur de l'action. Les dépendances sont aussi utilisées lors du recouvrement après défaillance pour calculer un état cohérent. Dans ce cas, l'action atomique construite est avortée. Il faut noter que suite à nos règles de dépendance, l'ensemble de cohérence construit à la suite d'une défaillance n'est pas le même que celui construit pour sauvegarder un point de reprise global cohérent.

Nous voulions que le comportement du système soit autant que possible similaire à celui d'un système non tolérant aux fautes : afin de permettre un partage des ressources, nous n'avons pas imposé de politique de sérialisation. De ce fait, les éléments de récupération de clients séparés peuvent être regroupés au sein du même ensemble de cohérence. Par exemple, dans la figure 2.4, le premier appel en lecture du client a ne créé pas de dépendance puisque l'élément de récupération du serveur vient d'être initialisé et que l'opération consulte l'état permanent. En revanche, le second appel du client a créé une dépendance car l'état du serveur a été modifié par le client b. De ce fait, lorsque le serveur prend la décision de sauvegarder un point de reprise, l'ensemble de cohérence contient les éléments de récupération des clients a, b et du serveur.

Dans la pratique, les décisions de sauvegarde de points de reprise sont effectuées soit régulièrement par une bibliothèque gérant l'élément de récupération des processus des applications, soit par des serveurs fiables tels que ceux qui gèrent les entrées-sorties. Il en résulte que très peu de programmeurs de serveurs ont à écrire l'instruction `checkpoint` et que dans une utilisation normale aucun programmeur d'application ne le fait.

Le protocole de point de reprise est implémenté en trois phases : `build_atomic_action`, `precommit` et `commit` qui sont mises en œuvre de façon complètement distribuée par des vagues de messages [87]. Le rôle de la phase `build_atomic_action` est de calculer un état distribué cohérent par transformation du graphe distribué d'élément de récupération en un arbre également distribué. Lors des deux autres phases, cet arbre est utilisé pour propager les vagues `precommit` et `commit`.

phase 1 (build_atomic_action) lorsqu'un processus utilisateur entre dans cette phase, il suspend son exécution pour éviter l'ajout de nouvelles dépendances. Toutefois, dans cette phase les serveurs continuent de traiter les appels. Pour construire l'arbre, une vague `build_atomic_action` est diffusée de l'initiateur vers tous les éléments de récupération desquels il dépend (c.-à-d., ses prédécesseurs). Lorsqu'il reçoit cette vague, un élément de récupération entre dans la phase `build_atomic_action`, devient un sous-nœud de l'émetteur de la vague et la propage à ses prédécesseurs. Si l'élément de récupération est déjà un sous-nœud, il renvoie juste un acquittement négatif à l'émetteur. Lorsque tous les prédécesseurs ont renvoyé leur acquittement

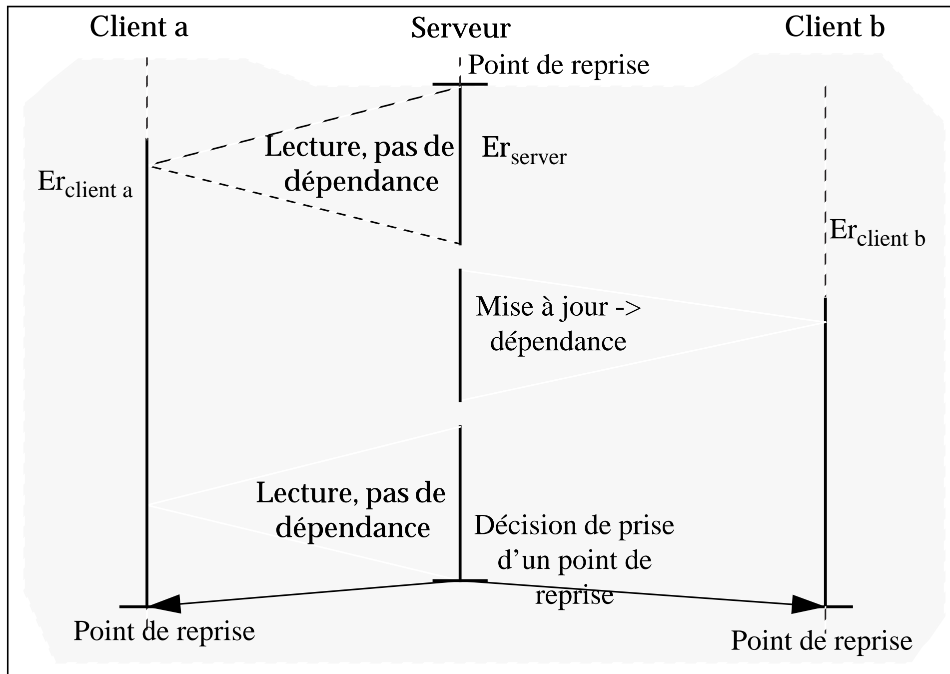

FIG. 2.4 – *Construction d'un état cohérent*

(positif ou négatif), l'élément retourne un accusé positif à l'émetteur de la vague. Seuls les prédécesseurs ayant répondu positivement sont gardés dans l'arbre final. La phase `build_atomic_action` se termine lorsque l'initiateur a reçu tous les acquittements de ses prédécesseurs.

phase 2, 3 (precommit, commit) les phases `precommit` et `commit` reposent sur le protocole classique de validation à deux phases [47], mis en œuvre par des vagues. Après réception de la vague `precommit`, un serveur ne traite plus les appels ; il effectue un `precommit` sur la transaction qui lui est associée, diffuse la vague, attend les acquittements de ses nœuds fils et renvoie un acquittement positif au nœud père. Lorsque l'initiateur a reçu tous les acquittements de ses nœuds fils et s'ils sont positifs, il exécute la phase de `commit`. Enfin, lors de la réception de la vague commit, les transactions STM sont validées et l'exécution reprend au sein d'un nouvel élément de récupération.

Il est possible qu'une vague `build_atomic_action` soit envoyée à un élément de récupération ayant déjà validé ou qui exécute les phases 1 et 2. Dans ce cas, il y a sérialisation de deux actions atomiques séparées : l'élément de récupération termine l'exécution du protocole de validation et renvoie un acquittement négatif pour ne pas être ajouté à la seconde action atomique. Notre algorithme gère également des situations plus complexes telles que le regroupement de plusieurs initiateurs ou la poursuite d'éléments de récupération. Une description détaillée peut être trouvée dans [67].

2.3.3 Mise en œuvre du modèle FTM au dessus du micro-noyau Mach 3.0

Le modèle FTM a été mis en œuvre au dessus du micro-noyau Mach 3.0 [1] en ajoutant des mécanismes permettant de gérer les défaillances [71]. En effet, les entités de base de Mach tel que les tâches, les ports et les objets mémoire sont perdus lors d'une défaillance de la machine. Afin de pouvoir relancer les applications et les serveurs fiables après une défaillance, nous avons développé les trois entités suivantes : la tâche stable, l'objet mémoire stable et le port persistant.

Tâche stable. Une tâche stable définit un environnement d'exécution ; cette entité est utilisée pour implémenter les serveurs fiables et les tâches utilisateurs. De manière similaire aux tâches standard, une tâche contient des droits d'accès et un espace d'adressage virtuel. Si le processeur qui exécute une tâche stable a une défaillance, l'environnement d'exécution est ré-installé sur le processeur de secours : les objets mémoire stables sont re-projetés à leur adresse précédente et un fil de contrôle est relancé, à partir du code d'initialisation pour les serveurs fiables (voir figure 2.3), et à partir du contenu sauvegardé du compteur ordinal pour les applications.

Objet mémoire stable. Un objet mémoire stable peut être assimilé à une STM logique. Il permet d'accéder à la STM réelle et d'y allouer des objets stables. Un objet mémoire stable est privé à une tâche stable ; il est toujours projeté à la même adresse virtuelle lors de la relance après défaillance.

Port persistant. Le port persistant possède essentiellement une fonction de nommage permettant de désigner les entités du modèle FTM avec un nom unique (c.-à-d., EUID) survivant aux défaillances. Dans une version primitive du modèle FTM [10], l'envoi et la réception de messages sur un port persistant possédaient la propriété d'être atomiques ; les messages étaient mémorisés en STM. Ultérieurement, nous avons relâché cette propriété car sa mise en œuvre était trop coûteuse et qu'elle n'était réellement utile que pour gérer les acquittements au sein des protocoles de point de reprise et de restauration après défaillance. Dans la version finale de FTM, la gestion des acquittements a été intégrée au sein des protocoles.

Finalement, il est à noter que ces trois entités ont été mises en œuvre par des bibliothèques et des serveurs spécialisés sans nécessiter de modifications du noyau.

2.4 Analyse de performance

Pour analyser les performances de FTM, nous avons utilisé des classes d'application représentatives de l'utilisation des stations de travail : calcul scientifique et outils de bureautique. Le calcul scientifique a été représenté par deux applications de longue durée : le calcul de nombres premiers et le programme `mp3d` issu de la suite de tests Splash [95]. Les outils de bureautique ont été illustrés par l'éditeur de texte `micro-emacs`.

Notre objectif était d'analyser la pénalité due à la tolérance aux fautes pour l'exécution sans défaillance d'un programme. On peut noter que cette pénalité ne se mesure pas de la même manière pour nos deux classes d'applications. Pour le calcul scientifique, nous avons mesuré le surcoût total en temps d'exécution tandis que pour les outils de bureautique, la pénalité est la durée de sauvegarde d'un point de reprise, c.-à-d., le temps pendant lequel l'utilisateur ne peut interagir avec l'outil.

2.4.1 Environnement de test

Nos tests comparent l'exécution du même programme, sur la même machine : (i) avec Mach 3/BSD (Mk75) et (ii) avec un système FTM issu de cette version de Mach. Le portage des programmes sur FTM a été effectué par simple recompilation sans modification des sources. Toutefois, pour analyser nos résultats de manière adéquate, nous avons instrumenté nos programmes avec des instructions `checkpoints`. Pour comprendre les sources d'inefficacité, nous avons mesuré divers paramètres tels que le surcoût total en temps d'exécution, la taille des points de reprise et la durée de la sauvegarde. Les tests ont été effectués sur une paire de sites stables construite à partir de mono-cartes industrielles MultibusII de puissance comparable à des Sun 3/60. La taille de la mémoire centrale est de 20 Mo.

La configuration logicielle utilisée par les applications est composée du processus de l'application, d'un paginateur de mémoire virtuelle recouvrable, et d'un serveur d'écran (pour micro-emacs). Le serveur de mémoire virtuelle gère trois segments pour l'application : les données, la pile et le tas. Nous avons un élément de récupération par serveur fiable et un pour l'application. En conséquence, un point de reprise met en jeu 4 éléments de récupération (5 pour micro-emacs). Dans notre configuration, la prise de point

Nombre de *Pr*	Durée totale (seconde)	Défauts de page	Surcoût en temps	Temps entre *Pr* (s)	Durée du *Pr* (s)	Taille du *Pr* (pages)
Mach, 0	1028	-	-	-	-	-
FTM, 0	1036	197	0,78 %	-	-	-
FTM, 4	1065	197	3,6 %	266	7,25	2(pile)+64
FTM, 7	1070	197	4,09 %	153	4,8	2+32
FTM, 13	1076	197	4,67 %	83	3	2+16
FTM, 25	1092	197	6,23 %	43,5	2,25	2+8
FTM, 49	1125	197	9,44 %	25	1,8	2+4
FTM, 98	1192	197	16 %	12	1,6	2+2

TAB. 2.1 – *Résultat du calcul de 400 000 nombres premiers*

de reprise est déclenchée par l'élément de récupération de l'application qui de ce fait devient coordonnateur. L'exécution du protocole de point de reprise donne lieu à l'échange de 5 messages entre le coordonnateur et un autre élément de récupération : 2 pour la phase `build_atomic_action`, 2 pour la phase de `precommit` et 1 pour la phase de `commit`. De ce fait, 15 messages Mach sont échangés (20 pour micro-emacs). Lors de la sauvegarde d'un segment de mémoire virtuelle, un minimum de 3 messages Mach sont échangés entre le serveur et le micro-noyau : une requête de `flush` du serveur vers le noyau, 1 message `data_return` pour chaque tranche contiguë de 256 Ko [2] du noyau vers le serveur et un message de fin de flush du noyau vers le serveur. Comme la taille d'une page mémoire est de 8 Ko, nous avons un message Mach pour chaque groupe de 32 pages.

2.4.2 Calcul de nombres premiers

Notre calcul de nombres premiers repose sur le crible d'Eratosthène. Ce programme possède un comportement de producteur de mémoire. Comme un entier est long de 4 octets, une page mémoire contient 2048 nombres premiers. La table 2.1 détaille les résultats du calcul de 400 000 nombres premiers en faisant varier la fréquence des points de reprise en fonction du nombre de pages de nombres premiers produites.

Pour un utilisateur le choix de la fréquence des points reprise correspond essentiellement au temps de calcul qu'il accepterait de perdre en cas de défaillance. Le surcoût en temps d'exécution de FTM est compris entre 3,6% et 16%. Le minimum est obtenu en effectuant un point de reprise toutes les 266 secondes, ce qui correspond à une perte maximale de 25% du calcul effectué. Le maximum de 16% est obtenu en effectuant un point de reprise toutes les 12 secondes, ce qui correspond à une perte potentielle de 1% du calcul effectué.

2.4.3 Mp3d

Mp3d est un programme de calcul scientifique représentatif de la simulation des fluides raréfiés [95]. Nous avons exécuté mp3d pour deux tailles d'espace de molécules, 10 000 et 30 000, sur 300 pas d'exécution (voir figure 2.5). Les points de reprise ont été sauvegardés

2. Ceci est la taille maximum permise par Mach.

tous les n pas, de manière à faire varier la quantité de calcul potentiellement perdue en cas de défaillance.

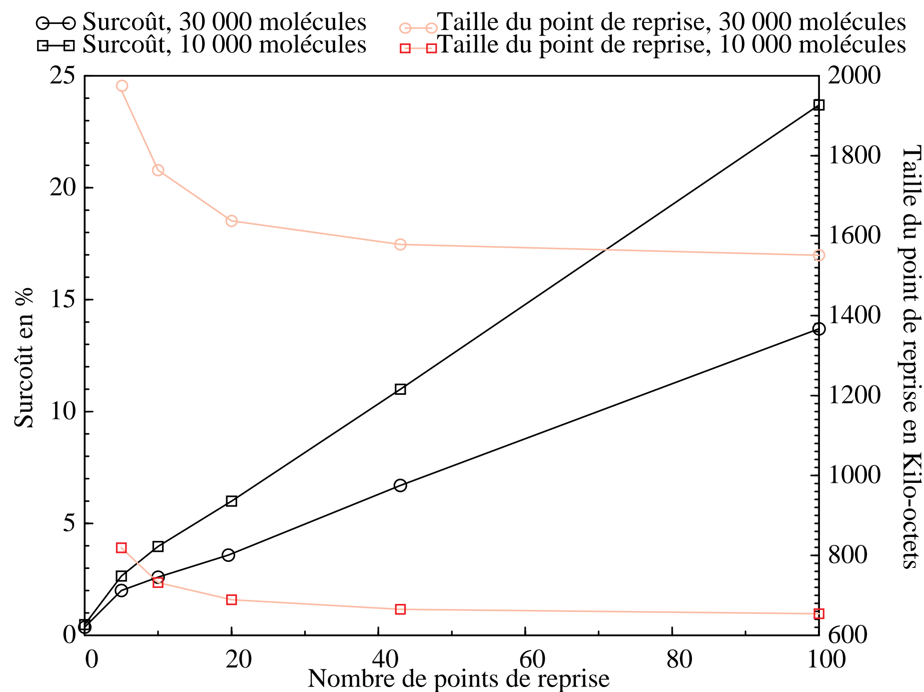

FIG. 2.5 – *Résultat de l'exécution de mp3d*

Le surcoût minimal mesuré (5 points de reprise) est de 2% et 2,6% pour respectivement 30 000 et 10 000 molécules. Le surcoût maximal (100 points de reprise) est de 13,7% et 23% pour respectivement 30 000 et 10 000 molécules. Le gradient du surcoût en temps d'exécution, dépendant du nombre de points de reprise, augmente lorsque le nombre de molécules diminue. Ceci est dû au fait que le temps d'exécution de `mp3d` augmente plus vite que l'espace mémoire consommé par le processus. Une autre observation pouvant être faite est que `mp3d` modifie pratiquement tout son espace de travail à chaque pas d'exécution. Il en résulte que la taille du point de reprise ne diminue pas avec l'augmentation du nombre de points de reprise.

2.4.4 Micro-emacs

Micro-emacs est un éditeur de texte assez répandu qui existe dans plusieurs versions pour des plateformes telles qu'Unix ou le PC. Nous avons choisi de porter cet éditeur car il est représentatif des outils de bureautique que l'on peut trouver dans le domaine public. Même si certains outils actuels fournissent une forme intégrée de point de reprise par l'utilisation de fichiers disques, ce n'est pas le cas le plus commun. De plus, si le système

	Défauts de page	Temps entre *Pr* (s)	Messages *data_return*	Durée du *Pr* (s)	Taille du *Pr* (pages)
Texte non modifié	10	15	3	$0,98 < d < 1,3$	3
Texte modifié	10	15	5	$1 < d < 1,4$	8

TAB. 2.2 – *Exécution de micro-emacs*

d'exploitation utilisé n'offre pas de système de fichiers transactionnel, la sauvegarde du point de reprise repose sur une solution ad-hoc. Il en résulte que la cohérence et la disponibilité du point de reprise ne sont pas toujours assurées en cas de défaillance [92]. Pour cette raison, nous pensons que la fonction “point de reprise” est de la responsabilité du système d'exploitation et doit y être intégrée.

Micro-emacs est représentatif des applications qui utilisent des entrées-sorties : lors de la reprise après défaillance, l'utilisateur doit voir sur son écran une représentation cohérente avec l'état interne de son texte. Cette fonctionnalité est offerte au moyen d'un serveur d'écran fiable qui émule une console VT100. Ce dernier est simplement mis en œuvre en mémorisant une copie de l'écran en STM. Lors de la reprise, le serveur d'écran remet à jour automatiquement la fenêtre de l'utilisateur avec un contenu cohérent avec les segments mémoire du processus de l'éditeur.

Pour un éditeur, le temps d'exécution est une mesure qui n'a aucun sens. Le critère important est la durée d'exécution du point de reprise, car pendant cette période l'application est suspendue et ne réagit pas aux interactions de l'utilisateur. Toutefois, les caractères entrés par l'utilisateur sont mis en tampon par le système et traités après la sauvegarde. Nos mesures ont été réalisées avec l'environnement FTM initialisé de manière à sauvegarder un point de reprise toutes les 15 secondes. Si le texte n'est pas modifié, la durée d'un point de reprise est approximativement d'une seconde avec une variation de 30% due à l'ordonnancement. Il est à noter que même si l'utilisateur ne modifie pas le texte, 3 pages sont modifiées par la boucle d'attente de caractères. Si le texte est modifié par entrée de caractères, en écartant le cas d'importantes modifications, 8 pages sont modifiées augmentant ainsi le temps approximatif de sauvegarde d'un point de reprise de 0.1 seconde.

2.5 Leçons et conclusions

L'objectif principal de FTM était la conception de systèmes tolérant les fautes de faible coût. Plus spécifiquement en matière de système d'exploitation, nous voulions assurer la transparence de la tolérance aux fautes au programmeur d'applications. Nous concluons maintenant en tirant les leçons de cette étude.

Système fiable extensible

Du point de vue d'un concepteur d'un système tolérant aux fautes, le problème est d'éviter d'avoir à concevoir et à développer un système complet à partir de la machine nue. Ceci nous a amené à concevoir un système extensible reposant sur les concepts de micro-noyau et de serveurs fiables. L'utilisation d'un micro-noyau préserve la pérennité des dé-

veloppements en cas de changement de l'architecture. L'extensibilité permet de configurer le système aux besoins des applications ; seuls les serveurs fiables réellement utilisés sont nécessaires dans une configuration donnée du système.

Dans la conception du modèle client-serveur fiable, nous avons pris soin de préserver la propriété d'extensibilité. Lors de la sauvegarde d'un point de reprise, un état cohérent est construit de manière dynamique sans requérir la spécification d'une configuration logicielle statique. Cependant, le prix algorithmique de l'extensibilité est que la sauvegarde d'un point de reprise nécessite l'exécution d'un protocole en trois phases au lieu de deux [59]. En revanche, notre modèle ne dégrade pas les performances du RPC : l'exécution d'un RPC FTM nécessite le même nombre de messages qu'un système Mach standard. La tolérance aux fautes n'engendre un surcoût que lorsqu'un point de reprise est sauvegardé.

Enfin, il est à noter que l'expérience acquise dans l'étude de FTM, nous a permis de concevoir d'autres services tolérants aux fautes pour des environnements plus spécifiques. En coopération avec Gilbert Cabillic et Isabelle Puaut, nous avons implémenté une version optimisée de notre algorithme de points de reprise pour une mémoire virtuelle partagée s'exécutant sur une plateforme multiprocesseur Mach/Paragon [23].

Par ailleurs, nous avons développé, en coopération avec Nadine Peyrouze, un serveur de fichiers fiable pour le protocole NFS [82, 76]. Ce serveur est similaire aux serveurs fiables FTM et repose sur l'utilisation d'un cache stable, issu de la STM, implémenté par logiciel. Il est à noter que ce serveur est deux fois plus rapide que le serveur SUN d'origine et offre un débit de 25% supérieur.

Vers un nouveau modèle de micro-noyau

Ainsi que nous l'avons montré dans le paragraphe précédent, la préservation de la propriété d'extensibilité se traduit par une augmentation du nombre de messages Mach échangés lors de la sauvegarde d'un point de reprise. La performance des communications noyau est par conséquent critique dans le modèle FTM.

La relative inefficacité des communications entre tâches est en autres due à la présence d'espaces d'adressages différents ce qui impose des conversions d'adresse potentiellement sources d'erreur. Comme il l'a été montré dans le système Opal [29], s'il est nécessaire qu'un client et serveur soient protégés l'un de l'autre, la séparation des espaces d'adressage n'est pas pour autant obligatoire.

Cela nous a amené à réfléchir à un nouveau modèle de noyau dans lequel les notions d'espace d'adressage et de protection sont découplés l'une de l'autre. Cette étude fait l'objet du chapitre suivant.

Chapitre 3

Support noyau pour la protection mémoire à grain fin

Dans un environnement de programmation moderne, une application est composée de modules (ex, bibliothèque graphique, base de données, ...) qui sont hétérogènes de multiple façons. Ceux-ci peuvent être écrits dans différents langages de programmation, ils peuvent provenir de différentes sources (ex, différents groupes de développement, du domaine public, de vendeurs de logiciels), ils peuvent être à différents stades de développement (ex, déverminage, test final), enfin ils peuvent être relatifs à différents niveaux du système d'exploitation (ex, application, bibliothèque, extension système).

Une conséquence de cette hétérogénéité est qu'une application est souvent composée de modules de différents niveaux de confiance. Il est de ce fait nécessaire de protéger les données d'un module vis-à-vis des autres. Un premier exemple est la protection des données d'une base de données vis-à-vis d'une bibliothèque graphique. Un second exemple est l'implémentation d'extensions système (ex, serveur fiable de FTM) ou des environnements d'exécution de langages : ces modules doivent avoir la visibilité sur le reste de l'application, mais l'inverse n'est pas vrai. L'implémentation de cette protection à grain fin implique que le système d'exploitation soit à même d'*isoler* les différents modules sans pénaliser la *performance des communications* inter-modules.

Ceci nous a amené à proposer un nouveau modèle de micro-noyau dans lequel les notions d'espace de protection et d'espace d'adressage sont découplées l'une de l'autre. En d'autres termes, une tâche est composée d'un ou plusieurs *domaines de protection* partageant un espace d'adressage unique. Un domaine de protection (DP) regroupe un ensemble de régions mémoire (c.-à-d., code et données) avec la propriété qu'une région mémoire peut seulement être accédée par un fil de contrôle s'exécutant au sein du domaine de protection. Le changement de domaines de protection s'effectue par un *appel de procédure protégé* (PPC).

Le bénéfice de ce modèle de noyau est la préservation de l'association à une application d'un espace d'adressage unique. Le partage de données est ainsi facilité, puisqu'il ne requiert plus de conversion des noms (adresses). Il en résulte moins d'erreurs de programmation et une plus grande efficacité à l'exécution.

Pourquoi enrichir le noyau avec des domaines de protection ?

Ainsi que nous l'avons décrit dans la section 1.1.2, il existe d'autres approches à la protection que la modification du noyau. L'utilisation d'un langage fortement typé tel que Modula-3 permet de prévenir des accès incontrôlés. Toutefois, cette technique n'est applicable que si tous les modules de l'application sont écrits dans des langages fortement typés. L'isolation de fautes [103] par insertion d'intructions vérifiant la légalité des adresses référencées est intéressante car indépendante du langage utilisée. Cependant l'intérêt de cette technique est limitée par le surcoût qu'introduit la vérification des adresses référencées.

Dans le contexte d'un noyau standard, la seule solution consiste à placer les modules de l'application dans différentes tâches. À titre d'exemple dans les systèmes à micro-noyaux, l'utilisation du concept de tâche en tant qu'unité de protection a été utilisée pour implémenter des sous-systèmes protégés (ex, personnalités Dos et Unix, systèmes de fichiers, couches de communication). Cependant, utiliser des tâches multiples au sein d'une application complique la programmation et rend coûteuse la communication inter-domaines. En conséquence, les programmeurs d'application choisissent souvent d'ignorer la protection pour leurs applications.

Dans un micro-noyau tel que Mach, la relative inefficacité des communications entre tâches est notamment due au nombre de fonctions à effectuer. Cette communication nécessite d'empaqueter les paramètres dans un message, d'ordonnancer les fils de contrôle, de commuter les espaces d'adressage et enfin de dépaqueter les paramètres. En adoptant un modèle de communication inter-domaines procédural par PPC, nous évitons de commuter les tâches. De plus, ce mécanisme tire également avantage du fait que les DP partagent un espace commun d'adressage : dans les cas où les DP se recouvrent et où le cache de traduction d'adresses de l'UGM (*Translation Lookaside Buffer*, TLB) contient des entrées visibles dans les DP appelés et appelants, nous évitons de recharger ces entrées du TLB dans le domaine appelé. Grâce aux différentes techniques d'optimisation que nous avons implémentées, un PPC peut être jusqu'à 6 fois plus rapide qu'un échange de messages entre deux tâches Mach 3.0.

Structure du chapitre

Le reste de ce chapitre est consacré à la description de DP-Mach, un micro-noyau Mach 3.0 [1] intégrant les domaines de protection. DP-Mach préserve la compatibilité binaire avec Mach 3.0, ce qui permet la réutilisation des logiciels existants. La section 3.1 présente le modèle du noyau. La section 3.2 décrit les points importants de l'implémentation, en particulier le PPC et les modifications apportées au micro-noyau Mach. Enfin, la section 3.3 présente une évaluation de performances pour une plateforme PC/i486 66Mhz. La conception et le développement de DP-Mach ont été effectués en collaboration avec Ciarán Bryce dans le cadre de sa thèse ; une description plus détaillée de la mise en œuvre peut être trouvée dans celle-ci [20].

3.1 Modèle d'un micro-noyau intégrant les domaines de protection

Entités de base

Dans notre modèle de noyau, la tâche représente presque la même abstraction que dans Mach : elle offre un environnement de nommage unique contenant un espace d'adressage et un espace de ports. De plus, elle inclut un ensemble de domaines de protection (DP). Un DP regroupe un sous-ensemble des régions mémoire d'une tâche. Un fil de contrôle s'exécute au sein d'un seul DP à un instant donné et peut seulement accéder les régions mémoire appartenant à ce DP. Comme toute abstraction Mach, un DP est nommé par un port.

Un fil de contrôle effectue un changement de DP au moyen d'un appel de procédure protégé (PPC) qui est un mécanisme similaire au LRPC [15]. Le changement de DP est effectué par appel d'un "trap" noyau ; au retour de ce trap le fil s'exécute au sein du domaine appelé. Un appel de procédure protégé ne peut être effectué que vers des points d'entrée prédéclarés lors de la création du domaine de protection.

Le nommage des ports est unique au sein d'une tâche. Toutefois, l'accès aux ports est contrôlé sur la base des DP. Une primitive noyau *dp_pass_port_right()* permet le transfert de droit sur un port d'un DP à un autre.

Outre les domaines de protection normaux, il existe deux sortes de DP particuliers :

- DP_ROOT : il existe un DP_ROOT par tâche. Celui-ci possède une visibilité sur toutes les régions mémoire de la tâche. Le DP_ROOT est le DP par défaut et est automatiquement créé à la création d'une tâche. Le DP_ROOT permet aux applications Mach standard de s'exécuter sans aucune modification de leurs binaires.

- *DP Superviseur* : c'est un DP dans lesquel un fil de contrôle s'exécute avec la priorité superviseur du processeur. Un DP superviseur possède l'accès à l'ensemble du noyau ainsi qu'au matériel. Ce type de DP offre une solution élégante pour la mise en œuvre d'extensions système possédant la même durée de vie que la tâche qui le contient [21].

Modules protégés

Du point de vue de l'application, un DP contient un *module protégé*. C'est une entité qui peut uniquement être accédée par les procédures de son interface. Un talon est associé à chaque module protégé. Le talon empile les paramètres de la procédure appelée et appele le trap *dp_call()*. Cet appel noyau effectue la commutation de domaines et relance le fil de contrôle dans le DP appelé. Un PPC possède la signature suivante :

ppc (dp_port, entry_point, proc_id, procedure_parameters,[capability])

Le paramètre *entry_point* identifie le module protégé ; c'est la valeur de l'adresse du point d'entrée dans le DP. Le paramètre *proc_id* identifie la procédure du module protégé qui va être appelée. Le paramètre *capability* est optionnel ; son but est de contrôler l'accès au module protégé. Dans notre modèle, une capacité est implémentée par un mot de

passe (c.-à-d., un nombre aléatoire). Cette capacité est vérifiée par le module appelé lors de chaque PPC. Le fait que le mécanisme de capacité soit implémenté dans le module protégé permet l'implémentation d'une politique de contrôle d'accès propre à chaque module (éventuellement abscente). Il est à noter qu'une implémentation similaire des capacités a été utilisée dans les systèmes Amoeba [66] et Opal [29].

Les modules protégés sont le plus souvent créés par le chargeur du système d'exploitation. Le chargeur crée la tâche et les régions mémoire projettées, puis il crée les DP contenus dans la tâche et initialise les points d'entrée en utilisant respectivement les appels *dp_create()* et *dp_register()*. Enfin, le chargeur rend visibles les régions mémoire au sein des DP.

3.2 Mise en œuvre des domaines de protection

Cette section décrit les modifications apportées au sein du micro-noyau Mach 3.0 pour gérer des DP, puis l'implémentation du PPC et ses optimisations possibles.

3.2.1 Modification du micro-noyau Mach 3.0

Les modifications de Mach concernent principalement le sous-système mémoire même si quelques changements mineurs ont été apportés aux sous-systèmes de gestion des ports et des fils de contrôle. Le sous-système mémoire de Mach repose sur les structures suivantes : VM_MAP, PMAP, K_OBJECTS et PAGES.

- VM_MAP : cette structure représente l'espace d'adressage de la tâche. Elle contient une liste de structure VM_MAP_ENTRY décrivant chaque région mémoire de la tâche. Une structure VM_MAP_ENTRY définit les valeurs d'héritage de la région, son gestionnaire mémoire, la taille et un pointeur sur le K_OBJECT contenant les pages de la région actuellement mappée en mémoire. Le module logiciel *vm_map* implémente les opérations de mémoire virtuelle de Mach telles que *vm_protect()*, *vm_allocate()*.

- K_OBJECT : cette structure est utilisée pour gérer les régions actuellement projetées en mémoire physique. Un K_OBJECT contient principalement le port du paginateur externe associé à la région et une liste des pages physiques.

- PAGES : les listes de pages contiennent des descripteurs de toutes les pages de la mémoire physique. Outre l'état de la page, un descripteur de page contient l'adresse physique de la page et un pointeur sur le paginateur externe auquel la page virtuelle appartient.

- PMAP : cette structure décrit les associations adresse virtuelle/adresse physique de la tâche. Il existe une seule structure PMAP par espace d'adressage. Le module *pmap* est dépendant du processeur. Son rôle est d'implémenter les opérations suivantes : trouver l'adresse physique associée à une adresse virtuelle, effacer cette association ou en ajouter de nouvelles, trouver les adresses virtuelles associées à une adresse physique, changer les droits de protection, et vérifier si une page de mémoire virtuelle a été lue ou écrite.

Le noyau dans Mach possède son propre espace d'adressage et sa propre stucture PMAP ; l'espace noyau est projeté dans tous les espaces d'adressage des tâches à la même adresse virtuelle dans la partie haute de l'espace d'adressage virtuel de chaque tâche.

Modifications apportées au sous-système mémoire

L'un de nos objectifs d'implémentation était de limiter les modifications dans la mesure du possible aux modules *vm_map* et *pmap*. Après modification, la structure PMAP référence un espace d'adressage contenant plusieurs DP. Plusieurs opérations ont été ajoutées dans le module *pmap* dans le but de créer, détruire, activer et désactiver des DP ainsi que pour changer la visibilité d'une région mémoire au sein d'un DP.

Le module *vm_map* a été modifié comme suit : (i) chaque structure VM_MAP_ENTRY contient une liste de DP dans lesquels une région est visible ; (ii) une structure supplémentaire, DP_MAP, est allouée par tâche et contient une liste de structures DP_MAP_ENTRY décrivant chacun des DP de la tâche.

Les UGM de processeurs CISC typiques tels que ceux des familles MC68000 et i486 n'implément pas la notion de domaine de protection. Pour ces processeurs, l'espace d'adressage est linéaire et est implémenté par un arbre de translation d'adresses. Un fil de contrôle s'exécutant dans une tâche possède un accès direct à toutes les pages mémoire actuellement résidentes en mémoire physique et référencées par l'arbre de la tâche. L'implémentation d'un DP consiste à lui associer un arbre de translation qui est évidemment un sous-ensemble de l'arbre de translation global de la tâche (c.-à-d., DP_ROOT). Changer de DP revient à changer le pointeur racine sur l'arbre de translation.

Lorsqu'un défaut de page est levé, le noyau consulte la structure VM_MAP de la tâche pour déterminer à quelle région mémoire, et donc à quel objet mémoire la page appartient. Le noyau vérifie que la page n'est pas actuellement en mémoire avant d'envoyer une requête au paginateur externe. L'entrée dans l'arbre de translation est mise à jour une fois que la page a été amenée en mémoire principale. Seul le PMAP du DP qui a généré le défaut est mis à jour ; si la page est visible dans d'autres DP, leurs arbres de traduction ne seront mis à jour que s'ils génèrent un défaut de page. Comme la page peut être déjà en mémoire, la résolution du défaut est moins coûteuse. Cette stratégie de *résolution paresseuse* est similaire à celle de Mach [1].

3.2.2 Schéma d'exécution de base du PPC

L'implémentation du PPC repose sur une optimisation du LRPC [15] pour un espace d'adressage unique. Le LRPC est lui-même une optimisation du RPC pour des espaces d'adressage situés sur le même processeur, pour lesquels les paramètres transférés sont non-complexes. La différence essentielle entre le PPC et le (L)RPC est que le PPC ne nécessite pas de procédures spéciales pour encapsuler des paramètres complexes de type pointeur (comme des listes chaînées) puisque ces entités sont nommées de façon unique par les domaines appelant et appelé.

L'implémentation du PPC utilise trois structures de données : une *A-stack*, une *E-stack* et *un segment de liaison*. Une A-stack est une région mémoire projetée dans les domaines appelant et appelé ; elle est utilisée pour transférer les paramètres. Une E-stack est une région mémoire allouée pour la pile du fil de contrôle ; de ce fait un fil de contrôle possède

une séquence de E-stacks, une pour chacun des domaines qu'il a traversés. Cependant, un fil de contrôle peut uniquement accéder à la E-stack de son domaine courant. Le *segment de liaison* est une entité visible par le noyau, qui pour chaque PPC conserve l'adresse de retour (PC) du fil de contrôle et l'adresse du pointeur de pile pour le domaine appelant.

Phase 1 - compilation et chargement. Pendant la compilation et le chargement, les talons de communication sont générés pour les modules appelant et appelé. Pour éviter (en C) la double copie des paramètres lors d'un appel de procédure (une copie sur la pile du fil de contrôle, suivie d'une copie sur la A-stack), les talons sont implémentés par des macros de code assembleur. Le talon choisit une A-stack qui est libre. Dans le cas où les DP appelant et appelé ne sont pas disjoints, l'utilisation des A-stacks n'est pas toujours nécessaire. C'est en particulier le cas lorsque le DP appelé est une extension de l'appelant, puisque l'appelé peut alors lire les paramètres directement. Les paramètres de retour doivent donc être placés dans la mémoire visible de l'appelant. À la création d'un DP, le noyau alloue de la mémoire dans le DP pour les E-stacks. Le nombre de E-stacks dans un DP représente le degré maximum de parallélisme du DP.

Phase 2 - connexion. Cette phase installe un canal de communication entre les DP. Les A-stacks sont projetées dans les deux domaines en utilisant la primitive *dp_make_visible()*. Le nombre de A-stacks représente le nombre maximum de PPC simultanés dans le domaine. La connexion est normalement effectuée lors du premier PPC entre les deux domaines.

Phase 3 - appel & retour. Lorsqu'un fil de contrôle exécute un PPC, le talon de communication associé choisit une A-stack libre, empile les paramètres et appelle le trap noyau *dp_call()*. Dans le trap *dp_call()*, le noyau sauvegarde le pointeur de pile et l'adresse de retour dans le *segment de liaison*, choisit une nouvelle E-stack, change la visibilité mémoire du DP, vérifie la validité de l'*entry_point* et de l'identificateur de la procédure, puis reprend l'exécution du fil de contrôle à l'adresse de la procédure appelée. En revenant d'un PPC, le fil de contrôle exécute un trap *dp_return()* qui récupère l'ancien pointeur dans la pile et retourne son adresse, change la visibilité mémoire du DP et reprend l'exécution du fil de contrôle. Finalement, le talon appelant libère la A-stack.

3.2.3 Optimisation du PPC

Deux optimisations du PPC sont réalisables grâce à l'unicité de l'espace d'adressage de l'application : (i) l'optimisation des appels aux DP superviseurs, (ii) l'invalidation sélective des entrées du TLB.

Optimisation du PPC pour les DP superviseurs

Le TLB contient généralement un attribut pour chaque entrée indiquant si l'entrée est visible en mode utilisateur ou en mode superviseur. On peut remarquer que dans ce cas précis, l'UGM offre une fonctionnalité similaire à celle d'un *Protection Lookaside Buffer* [58]. De ce fait, la visibilité mémoire est complètement gérée par l'UGM lorsque la priorité du

CPU est changée à l'intérieur du trap *dp_call*. Par ailleurs, il n'y a pas besoin d'allouer une A-stack puisque le DP superviseur possède une visiblité sur l'appelant. Comme la routine est obligatoirement considérée sûre, il n'y a pas besoin non plus d'allouer une nouvelle E-stack.

Optimisation du PPC par invalidation sélective du TLB

Dans le schéma de base du PPC, un changement de DP est implémenté en changeant le pointeur racine de l'arbre de translation de l'UGM, invalidant ainsi toutes les entrées du TLB sur certains processeurs CISC (ex, i486). Cependant, quand les DP appelant et appelé se chevauchent, certaines entrées TLB sont invalidées inutilement (une entrée qui vient juste d'être invalidée peut être rechargée). Il a été mesuré qu'environ 25% du surcoût dans un appel LRPC nul est dû aux défauts de TLB initiaux dans le domaine appelé [15]. Par conséquent, il est parfois préférable de modifier l'arbre de translation puis d'invalider sélectivement les entrées du TLB plutôt que d'effectuer une invalidation globale.

Le surcoût dû au TLB dans le cas d'une invalidation globale est égal au coût de l'invalidation du TLB plus le coût des défauts sur les entrées dans F, où F est l'ensemble des adresses utilisées par le DP appelé qui génèreraient le chargement d'une entrée dans le TLB si celui-ci était vide. Les chargements suivants ne nous intéressent pas puisque le coût de ces défauts n'est pas influencé par le fait que le TLB soit invalidé globalement ou sélectivement. Soient tlb_{glob} le coût de l'invalidation globale du TLB et tlb_{miss} le coût (en termes de cycles processeur supplémentaires) de l'accès à un mot mémoire quand son adresse virtuelle n'est pas dans le TLB. # dénote la cardinalité d'un ensemble. Le surcoût d'une invalidation globale du TLB est alors :

$$tlb_{glob} + (\#F * tlb_{miss})$$

Dans le cas général, le surcoût pour l'appelant d'une invalidation sélective est le coût de l'invalidation des entrées du TLB non visibles dans le DP appelé, plus le coût des défauts de TLB sur l'ensemble F dont les éléments ne sont pas encore présents dans le TLB, plus le coût de la modification des entrées de l'arbre de translation d'adresse (PTE) pour refléter la nouvelle visibilité :

$$\#(N \setminus C) * tlb_{sel} + ((\#F - \#(F \cap C)) * tlb_{miss}) + (D * y)$$

Les paramètres sont : tlb_{sel}, le coût de l'invalidation sélective d'une entrée dans le TLB. N est l'ensemble des entrées dans le TLB suivant la commutation d'un DP mais avant que l'exécution ne reprenne dans le DP appelé. C est un sous-ensemble de N, contenant les entrées TLB qui sont valides dans le DP appelé. y est le coût de l'écriture d'une nouvelle valeur dans un PTE. D est la différence en taille de page entre les deux DP, c'est-à-dire le nombre de pages visibles dans le DP appelée mais n'appartenant pas à l'appelant. Le terme $\#(F \cap C)$ dépend du comportement de la fonction appelé et est égal au nombre d'entrées de l'appelant qui sont ré-utilisées. Sa valeur est contenue dans l'intervalle [0 .. #F]. Renommons-le $\alpha\#F$ avec α dans [0 .. 1]. Avec ces notations, une invalidation sélective est rentable lorsque :

$$(\#(N \setminus C) * tlb_{sel}) - (\alpha\#F * tlb_{miss}) + (D * y) < tlb_{glob}$$

Analyse d'un PPC dans un DP englobant

Les DP englobants sont utiles pour l'implémentation de sous-systèmes sûrs qui ne requièrent cependant pas une exécution dans un DP superviseur. À titre d'exemple, nous pouvons citer les serveurs fiables FTM, des environnements d'exécution pour langage objets, ou des personnalités système. Nous analysons maintenant le gain de l'invalidation sélective de l'appel d'un DP englobant pour un processeur i486.

Lors de l'appel, il n'y a pas besoin d'invalider le TLB puisque les adresses de l'appelant sont également visibles dans le DP appelé. $(N \setminus C)$ est égal à l'ensemble vide (N contiendra les entrées noyau après le trap bien que ces dernières ne peuvent évidemment pas être utilisées par le DP appelé). Pour le i486, y vaut 2 : une instruction pour charger la valeur du PTE dans un registre, puis une seconde instruction qui l'écrit à l'adresse du PTE. tlb_{glob} vaut 4. Pour une page en mémoire physique, tlb_{miss} requiert 13 cycles processeur dans le cas le plus rapide (le pire pour nous), et 29 dans le cas le plus lent (quand les bits *referenced* et *dirty* du descripteur de la page physique doivent être positionnés par le processeur). Par conséquent, l'invalidation sélective est meilleure lorsque :

$$D < ((\alpha \# F * 13) + 4)/2$$

Le cas le plus défavorable se produit quand α vaut 0 ; D doit être égal à 1. Ceci est peu probable puisqu'il y a partage entre les deux DP à cause du passage de paramètres. Le cas le plus favorable se produit quand α vaut 1. Pour un #F typique égal à 20, D doit être inférieur à 132 pour que l'invalidation sélective soit plus efficace. En pratique, α dépend essentiellement de la taille et du mapping mémoire des paramètres. Sur un i486, la taille du TLB est de 32. Par ailleurs, le noyau consomme quelques entrées lors de l'exécution d'un trap et de la modification du PTE, réduisant ainsi la valeur de α. Finalement, il doit être mentionné que nous devons utiliser l'adressage absolu pour invalider une entrée du TLB. Cette adresse peut être récupérée à partir de la phase d'édition de liens.

Lors du retour, F s'applique aux entrées chargées par le DP appelant (c.-à-d., les résultats retournés par la procédure). La seule différence avec la formule précédente est que le premier terme pour l'invalidation sélective $(\#(N \setminus C))$ devient D ; le noyau ne peut pas savoir quelles adresses DP visibles dans le DP appelé ont été chargées dans le TLB pendant l'appel de procédure, aussi il doit toutes les invalider (l'invalidation sélective de TLB est de 11 cycles quand l'entrée est présente et de 12 quand elle ne l'est pas). Pour un ensemble F de taille 20, une invalidation sélective devient plus efficace quand D est plus petit que 20 pour un α minimal, et 0 pour un α maximal. La section suivante présente une évaluation réelle du PPC.

3.3 Évaluation de performance

L'implémentation de DP-Mach a été effectuée par modification de la version OSF Norma MK 78 de Mach 3.0. La plupart des modifications ont été effectuées en C ; seules quelques procédures ont été implémentées en assembleur pour empiler les paramètres dans les traps *dp_call()* et *dp_return()*. Pour évaluer l'efficacité du PPC vis-à-vis d'autres solutions, nous avons mesuré le temps d'exécution d'une fonction `Max()`, qui calcule le maxi-

No.	Implémentation	Temps d'exécution (μs)	Accélération
1	Tâches séparées	100,9	-
2	PPC (sans optimisation TLB)	24,2	**4,1**
3	PPC (avec optimisation TLB)	16,2	**6,2**
4	Bibliothèque utilisateur	0,8	**125**
5	Appel système (Trap)	6,7	**15**
6	Appel système (Port)	48,8	**2,1**

TAB. 3.1 – *Analyse comparative du temps d'exécution d'un PCC*

Tâche	Temps d'Exécution (μs)
Gestion de la A-Stack	1,5
Trap *dp_call()*	11,1
Trap *dp_return()*	11,1
Procédure `Max ()`	0,5
Total	24,2

TAB. 3.2 – *Analyse détaillée du temps d'exécution d'un PCC*

mum de deux entiers. Les résultats de l'exécution pour une plateforme PC/i486 66 Mhz sont présentés dans le tableau 3.1.

La première ligne du tableau 3.1 correspond à une émulation des DP par des tâches Mach. Le PPC est émulé par l'envoi d'un message sur un port et un ordonnancement optimisé par la technique du *handoff scheduling* [18].

La seconde ligne du tableau 3.1 donne le temps d'exécution d'un PCC à la fonction `Max()`, cette procédure étant exécutée dans un DP disjoint de celui du module appelant. Une analyse détaillée du temps d'exécution de ce PCC est présentée dans le tableau 3.2. Il est à noter que la fonction `Max()` est exécutée à partir d'un TLB vide puisque celui-ci est invalidé globalement dans les traps *dp_call()* et *dp_return()*. À titre de comparaison, le coût de l'exécution de la fonction `Max()` avec un TLB initialisé (dans une boucle) est d'environ 0,2 μs. On peut constater que le PCC est 4 fois plus rapide que l'émulation du PCC par un envoi de messages entre tâches.

La troisième ligne du tableau 3.1 donne le temps d'exécution de la fonction `Max()` dans un DP englobant celui de l'appelant. Le TLB est invalidé sélectivement. La procédure est appelée dans une boucle. Il est à noter que le contenu du TLB est encore valide pour l'appelant au retour de la procédure. Par ailleurs, le temps d'exécution des deux traps est réduit puisque les entrées du TLB correspondant au noyau ne sont pas invalidées. Cette implémentation est plus de 6 fois plus rapide que l'émulation du PCC. Toutefois, on peut noter que le PPC optimisé est encore 20 fois plus lent que l'utilisation d'une bibliothèque utilisateur (voir ligne 4).

La cinquième ligne donne le temps d'exécution de la fonction `Max()` lorsqu'elle est implémentée par un appel système reposant sur un trap. Les appels système Mach implémentés de cette manière incluent entre autres *mach_thread_self* qui retourne le port du fil

de contrôle courant. Le temps d'exécution est très similaire à celui d'un PPC vers un DP superviseur.

La sixième ligne du tableau 3.1 correspond à une implémentation par un appel système reposant sur un port : l'appel est implémenté par un talon qui envoie un message sur un port ; ce message est finalement traité par le noyau, et le résultat est renvoyé de la même manière. Il est à noter que ces deux dernières solutions ne permettent pas le chargement dynamique d'extensions système — le noyau doit être à nouveau compilé, lié et redémarré.

Finalement, notons que comme la durée de l'exécution du DP s'accroît, le rapport PCC TLB-optimisé/non-optimisé s'amenuise. Toutefois, un PPC non-optimisé est toujours plus performant qu'un échange standard de messages Mach.

3.4 Conclusion

Les apports de cette étude sont multiples. Notre modèle de noyau offre une alternative à la structuration en tâches d'applications composées de modules de différents niveaux de confiance. Le gain immédiat est relatif aux performances puisqu'un PPC est jusqu'à 6 fois plus rapide qu'un échange de messages Mach. De plus, le processus de développement est plus fiable car on évite de recourir à des conversions de pointeurs entre tâches.

Le noyau intégrant les domaines de protection, DP-Mach, est compatible au niveau binaire avec le noyau Mach standard, ce qui permet la réutilisation des applications existantes. Enfin, il est à noter que peu de modifications ont été apportées à Mach pour y intégrer les domaines de protection. Bien qu'une partie du mérite vienne de la structuration modulaire du noyau Mach, notre expérience montre que la protection à grain fin peut être aisément étendue à d'autres types de noyaux.

Du noyau extensible au système adaptatif

Du point de vue de l'extensibilité, l'apport ce travail est d'offrir une solution élégante au support d'extensions système de même durée de vie qu'une application, via la notion de domaines de protection superviseurs. L'extension système est chargée de manière transparente au programmeur au lancement de l'application et est supprimée lorsque l'application s'arrête.

On conçoit aisément que la conception d'extensions système ne relève pas des compétences du programmeur ordinaire d'applications. En fait, on aimerait disposer d'extensions génériques adaptables aux conditions précises d'utilisation d'une application. Bien évidemment, cette approche n'est intéressante que si le processus d'adaptation est réalisé de manière automatique. Ces motivations nous ont amené à nous intéresser aux techniques de transformation de programmes et plus précisément à l'évaluation partielle.

Chapitre 4

Spécialisation automatique de composants système

Dans un système reposant sur l'utilisation généralisée d'extensions système spécifiques aux applications, le problème qui se manifeste est l'automatisation de la construction de telles extensions. L'écriture manuelle de composants *ad hoc* est bien sûr possible, mais cette solution ne peut être raisonnablement utilisée que par des concepteurs d'applications experts en système.

Pour répondre à ce problème, nous avons proposé une solution reposant sur la spécialisation de composants génériques. Cette solution n'est bien sûr intéressante que si le processus de spécialisation est automatique. Ceci nous amené à participer à la conception d'un spécialiseur de programmes C (Tempo), qui est plus particulièrement conçu pour la spécialisation de programmes système.

Notre approche peut être appliquée à deux classes de composants adaptatifs : soit des composants conçus et développés spécifiquement dans un but de spécialisation, soit des composants systèmes existants qui recèlent des opportunités de spécialisation. Dans ce dernier cas, la spécialisation permet de réutiliser et d'optimiser du code existant tout en préservant la sémantique du code original.

Ce chapitre montre l'intéret de la spécialisation d'un composant système du commerce, en l'occurence le RPC de Sun. Dans la section 4.1, nous présentons l'architecture logicielle du RPC de Sun. La section 4.2 étudie les opportunités d'optimisation qu'il recèle. La section 4.3 décrit comment ces opportunités peuvent être exploitées automatiquement par Tempo pour spécialiser le programme original. La section 4.4 présente les gains de performance obtenus grâce à la spécialisation. Nous analysons en section 4.5 les problèmes rencontrés dans l'utilisation de Tempo. Enfin, nous concluons dans la section 4.6.

Le travail décrit dans ce chapitre a été effectué en étroite collaboration avec les membres du groupe Compose ayant participé à la conception de Tempo et, dans le cadre d'une coopération avec l'Oregon Graduate Institute, avec Calton Pu et Ashvin Goel.

4.1 Le RPC de Sun et son optimisation

Le protocole RPC de Sun a été introduit en 1984 pour faciliter l'implémentation des services distribués. Ce protocole est devenu un standard *de facto*. On le trouve aujourd'hui dans de nombreux services distribués, tels que NFS [65] et NIS [85]. Comme les réseaux sont souvent de nature hétérogène, il est nécessaire de communiquer en utilisant un format de données indépendant des machines, ce qui implique l'utilisation de protocoles d'encodage et de décodage.

Les deux principales fonctionnalités du RPC Sun sont :

1. Un générateur de talon (`rpcgen`) qui produit les fonctions talon du client et du serveur. Les fonctions talon traduisent les paramètres de l'appel de procédure en un format de message indépendant de la machine appelé XDR, puis les messages XDR en paramètres de la procédure. Outre le RPC, le protocole XDR est également utilisé dans des environnements de calcul distribué hétérogènes tels que PVM [43] pour un modèle à messages et Stardust [24] pour un modèle à mémoire partagée distribuée.

2. La gestion de l'échange de messages à travers le réseau.

Concrètement, le code du RPC Sun possède une architecture constituée de micro-couches, chacune consacrée à une fonction précise. Par exemple, certaines micro-couches servent à écrire les données lors de l'encodage, à lire les données lors du décodage et à gérer les protocoles de transport tels que TCP ou UDP. Chaque micro-couche est générique ; elle peut être implémentée de diverses façons. De ce fait, l'organisation en micro-couches du code RPC est caractéristique des logiciels système modulaires.

Un exemple simple

À titre d'exemple, pour illustrer cette architecture en micro-couches du code RPC Sun, nous considérons une fonction `rmin` qui envoie deux entiers à un serveur distant et qui retourne leur minimum.

Le programmeur utilise `rpcgen` (le compilateur de talon du RPC) pour compiler une spécification de l'interface de la procédure `rmin` en un ensemble de fichiers sources C. Ces fichiers implémentent du côté client, l'envoi de l'appel et la réception du résultat, et du coté serveur, la réception de l'appel et le retour des résultats. Pour mettre en valeur le code effectivement exécuté, la figure 4.1 présente une trace d'exécution d'un appel à `rmin` (nous ne listons pas tous les fichiers générés par `rpcgen`.[1])

Performance du RPC

Le RPC est un des paradigmes de base sur lesquels reposent la plupart des systèmes distribués. De ce fait, la performance de ce composant est critique, et de nombreuses recherches ont été menées sur son optimisation [30, 54, 51, 80, 93, 100]. La plupart de

1. Par souci de clarté, nous omettons quelques détails dans le listing du code : les déclarations, les arguments et instructions "inintéressants", la gestion des erreurs, les conversions de type (*casts*), et un niveau d'appel de fonction.

```
arg.int1 = ...                          // Positionne le premier argument
arg.int2 = ...                          // Positionne le second argument
rmin(&arg)                              // Interface utilisateur RPC générée par rpcgen
  clnt_call(argsp)                      // Appel de procédure générique (macro)
    clntupd_call(argsp)                 // Appel générique à UDP
                                        // Encodage de l'identificateur de la procédure
      XDR_PUTLONG(&proc)                //   Encodage générique en mémoire, stream... (macro)
        xdrmem_putlong(lp)              //   Écriture dans le tampon de sortie
                                        //       et vérification de non-débordement
          htonl(*lp)                    //   Sélection entre Big et little endian (macro)
      xdr_pair(argsp)                   // Fonction talon générée par rpcgen
                                        //   Encodage du premier argument
        xdr_int(&argsp->int1)           //     Sélection dépendant de la machine
                                        //         sur la taille de l'entier
          xdr_long(intp)                //     Encodage et décodage générique
            XDR_PUTLONG(lp)             //     Encodage en mémoire
              xdrmem_putlong(lp)        //     Ecriture dans le tampon de sortie
                                        //         et vérification de non-débordement
                htonl(*lp)              //     Sélection entre Big et little endian (macro)
                                        //   Encodage du second argument
        xdr_int(&argsp->int2)           //     Sélection dépendant de la machine
                                        //         sur la taille de l'entier
          xdr_long(intp)                //     Encodage et décodage générique
            XDR_PUTLONG(lp)             //     Marshaling générique en mémoire
              xdrmem_putlong(lp)        //     Écriture dans le tampon de sortie
                                        //         et vérification de non-débordement
                htonl(*lp)              //     Sélection entre Big et little endian (macro)
```

FIG. 4.1 – *Trace abstraite de la partie encodage d'un appel distant à* `rmin`

ces études se traduisent par la conception de nouveaux protocoles qui ne sont pas compatibles avec un standard existant tel que le RPC Sun. Par ailleurs, le problème de la ré-implémentation d'un protocole qui est spécifié uniquement par son implémentation est que ses caractéristiques (et même ses bogues) peuvent être perdues, résultant en une implémentation qui n'est pas compatible.

Optimisation de logiciel existant

Une alternative à la ré-implémentation d'un composant système est de dériver systématiquement une version optimisée à partir du code existant. Un premier avantage de cette approche est que les versions dérivées restent compatibles avec le standard. Un second avantage est que le processus de dérivation systématique peut être répété pour des machines et systèmes différents.

La question qui se pose naturellement est : existe-t'il des opportunités significatives pour la dérivation de versions optimisées à partir de composants système existants ?

En fait, de nombreux composants système existants sont connus pour être génériques et structurés en couches et en modules. Ceci introduit différentes formes d'interprétation qui sont d'importantes sources d'inefficacité ainsi qu'il a été montré, par exemple, pour le

cas du système de fichiers HP-UX [84]. Dans le RPC Sun, cette généricité prend la forme de différentes couches de fonctions qui interprètent des descripteurs (c.-à-d., des structures de données) pour déterminer les paramètres du processus de communication : la sélection du protocole (TCP ou UDP), le fait d'encoder ou de décoder, la gestion des tampons, ...

Il est important de souligner que la plupart de ces paramètres sont connus pour tout appel de procédure distante. Cette information peut être utilisée pour générer du code spécialisé dans lequel ces interprétations ont été éliminées. En fait, le code résultant est adapté à une situation spécifique.

Nous examinons maintenant les formes d'interprétation dans le RPC Sun, et comment celles-ci peuvent être optimisées par spécialisation.

4.2 Opportunités de spécialisation dans le RPC Sun

Le RPC de Sun repose sur différentes structures de données telles que `CLIENT` ou `XDR`. Les champs de ces structures maintiennent des informations relatives aux différentes couches du RPC : protocoles, tampons, contrôle d'encodage et de décodage, ... Certains de ces champs ont des valeurs qui peuvent être disponibles avant que l'exécution ait réellement lieu. Ces valeurs ne dépendent pas des arguments du RPC ; elles sont soit interprétées plusieurs fois, soit propagées à travers les couches du processus de codage/décodage. Comme ces valeurs sont disponibles avant l'exécution, elles peuvent être la source d'optimisations : les calculs dépendant seulement de valeurs connues (ou *statiques*) peuvent être réalisés pendant une phase de spécialisation. Le programme spécialisé consiste seulement en calculs dépendant des valeurs inconnues (ou *dynamiques*).

Nous décrivons maintenant des perspectives de spécialisation typiques du RPC Sun. Nous illustrons ces perspectives par des extraits de code réel, annotés pour montrer les calculs statiques et dynamiques. Dans les figures suivantes, les calculs dynamiques correspondent à des fragments de code imprimés en gras, alors que les calculs statiques sont imprimés en style Romain.

4.2.1 Suppression de la sélection entre l'encodage et le décodage

Nous examinons d'abord une perspective de spécialisation qui illustre une forme d'interprétation. La fonction en question, `xdr_long`, est présentée à la figure 4.2. Cette fonction permet soit d'encoder, soit de décoder des entiers longs. Elle sélectionne l'opération à réaliser à partir du champ `x_op` de son argument `xdrs`. Cette forme d'interprétation est utilisée dans des fonctions similaires pour d'autres types de données.

En fait, le champ `x_op` est connu à partir du contexte d'exécution (c.-à-d., processus d'encodage ou de décodage). Cette information, contenue dans la structure `xdrs`, peut être propagée vers le bas de façon interprocédurale par la fonction `xdr_long`. En résultat, l'envoi de `xdrs->x_op` est totalement éliminé ; la version spécialisée de cette fonction se réduit à seulement l'une des constructions de retour. Dans ce cas, la fonction `xdr_long()` spécialisée est suffisamment petite pour disparaître par dépliage (*inlining*).

```
bool_t xdr_long(xdrs,lp)               // Encodage ou décodage d'un entier long
  XDR *xdrs;                           //     Gestion de l'opération XDR
  long *lp;                            //     Pointeur sur la donnée à lire ou écrire
{
  if( xdrs->x_op == XDR_ENCODE )       // Si en mode encodage
        return XDR_PUTLONG(xdrs,lp);   //     Ecrire un entier long dans le tampon
  if( xdrs->x_op == XDR_DECODE )       // Si en mode décodage
        return XDR_GETLONG(xdrs,lp);   //     Lire un entier long dans le tampon
  if( xdrs->x_op == XDR_FREE )         // Si en mode libération
        return TRUE;                   //     Rien ne doit être fait
  return FALSE;                        // Retourne échec si le mode n'est pas reconnu
}
```

FIG. 4.2 – *Lecture ou écriture d'un entier long :* `xdr_long()`

4.2.2 Suppression du contrôle de non-débordement

Une seconde forme d'interpétation apparaît lorsque l'on vérifie que l'on ne déborde pas des tampons. Cette situation s'applique à la fonction `xdrmem_putlong` présentée à la figure 4.3. Au fur et à mesure que l'encodage s'effectue, l'espace restant dans le tampon diminue. Cette information est maintenue dans le champ `x_handy`. Comme dans le premier exemple, `xdrs->x_handy` est d'abord initialisé (c.-à-d., on lui donne une valeur statique), puis est décrémenté par des valeurs statiques et est testé plusieurs fois (lors de chaque appel à `xdrmem_putlong` et aux fonctions associées). Comme l'ensemble du processus de contrôle de non-débordement du tampon repose sur l'utilisation de valeurs statiques, il peut être évalué de manière statique à condition que le nombre de paramètres de la procédure soit fixe. Seules les écritures dans le tampon restent dans la version spécialisée (à moins qu'un débordement de tampon ne soit découvert).

Ce second exemple est particulièrement important. Il apporte premièrement un gain en performance significatif. Deuxièmement, la transformation décrite ici est strictement et systématiquement dérivée du programme original, par opposition à une élimination manuelle qui ne garantirait pas le contrôle de non-débordement du tampon.

4.2.3 Propagation du résultat

Le troisième exemple repose sur les résultats des deux exemples précédents. La valeur de retour de la fonction `xdr_pair` (présentée à la figure 4.4) dépend de la valeur de retour de `xdr_int`, qui à son tour dépend de la valeur de retour de `xdr_putlong`. Nous avons vu précédemment que `xdr_int` et `xdr_putlong` ont une valeur de retour statique. Par conséquent, la valeur de retour de `xdr_pair` est également statique. Si nous spécialisons l'appelant de `xdr_pair` (c.-à-d., `clntudp_call`) ainsi que cette valeur de retour, `xdr_pair` n'a plus besoin de retourner une valeur : le type de la fonction peut être transformé en `void`. La procédure spécialisée, avec les appels spécialisés à `xdr_int` et `xdr_putlong` dépliés, est présentée à la figure 4.5. La valeur réelle du résultat, qui est toujours `TRUE` indépendamment de l'argument `objp` dynamique (l'écriture de deux entiers ne provoque jamais de débordement du tampon), est utilisée pour supprimer un test

```
bool_t xdrmem_putlong(xdrs,lp)                // Copie d'un entier long dans le tampon
  XDR *xdrs;                                  //    Gestion de l'opération XDR
  long *lp;                                   //    Pointeur sur la donnée à écrire
{
  if((xdrs->x_handy -= sizeof(long)) < 0)     // Décrémente l'espace restant dans le tampon
    return FALSE;                             //    Retourne échec si débordement
  *(xdrs->x_private) = htonl(*lp);            // Copie dans le tampon
  xdrs->x_private += sizeof(long);            // Pointe sur le prochain emplacement
                                              //    de copie dans le tampon
  return TRUE;                                // Retourne succès
}
```

FIG. 4.3 – *Ecriture d'un entier long :* `xdrmem_putlong()`

supplémentaire dans `clntudp_call` (non décrit).

```
bool_t xdr_pair(xdrs, objp)                 // Encoder les arguments de rmin
{
  if (!xdr_int(xdrs, &objp->int1)) {        // Encoder le premier argument
    return FALSE;                           //    Eventuellement propager l'échec
  }
  if (!xdr_int(xdrs, &objp->int2)) {        // Encoder le second argument
    return FALSE;                           //    Eventuellement propager l'échec
  }
  return TRUE;                              // Retourner le code de succès
}
```

FIG. 4.4 – *Routine d'encodage* `xdr_pair()` *utilisée dans* `rmin()`

4.2.4 Bilan des opportunités de spécialisation

Le but de l'encodage est de copier les arguments dans le tampon de sortie. Pour réaliser cette tâche, le code minimal que nous pouvons dériver est celui qui est présenté à la figure 4.5. La même situation se produit pour le décodage, excepté que des tests dynamiques supplémentaires sont effectués pour assurer la validité et l'authenticité de la réponse du serveur.

Nous avons vu qu'une approche systématique pour spécialiser le code système peut permettre des simplifications significatives du code. Toutefois, il n'est pas possible de spécialiser manuellement le RPC pour chaque fonction distante, car le processus est long, fastidieux, et source d'erreurs. Cependant, comme le processus est systématique, il peut être automatisé. Nous présentons maintenant comment cette automatisation est réalisée au moyen de Tempo.

```
void xdr_pair(xdrs,objp)              // Encoder les arguments de rmin
{
                                      // Contrôle de débordement éliminé
  *(xdrs->x_private) = objp->int1;    // Appel spécialisé déplié
  xdrs->x_private += 4u;              //    pour l'écriture du premier argument
  *(xdrs->x_private) = objp->int2;    // Appel spécialisé déplié
  xdrs->x_private += 4u;              //    pour l'écriture du second argument
                                      // Code de retour éliminé
}
```

FIG. 4.5 – *Routine spécialisée d'encodage* `xdr_pair()`

4.3 Spécialisation automatique au moyen de l'évaluateur partiel Tempo

Tempo est un système de transformation de programmes basé sur *l'évaluation partielle* [31, 55]. Tempo prend en entrée un programme source $P_{générique}$ écrit en C ainsi qu'un sous-ensemble connu de ses arguments, et produit un programme source C spécialisé $P_{spécialisé}$, simplifié par rapport aux arguments connus. Une des caractéristiques innovantes de Tempo est d'offrir une approche uniforme à la spécialisation de programmes à la compilation et à l'exécution. Toutefois, dans le cadre de nos expériences sur le RPC, nous utilisons seulement la spécialisation à la compilation.

Le cœur d'un évaluateur partiel tel que Tempo est l'analyse de temps de liaison (voir figure 4.6). Tempo utilise une description du contexte de spécialisation (c.-à-d., les arguments statiques) pour analyser le programme $P_{générique}$ en divisant les contructions en parties *statique* et *dynamique* (voir figure 1.1). Lors de la phase de spécialisation, la partie statique de $P_{générique}$ est évaluée en utilisant des valeurs concrètes pour chaque entrée connue, tandis que la partie dynamique est *résidualisée* (copiée) dans le programme spécialisé de sortie. Le résultat de l'analyse temps de liaison est codé en un arbre d'action qui sert à générer un spécialiseur dédié à $P_{générique}$.

Plus l'analyse de temps de liaison est fine, plus elle révèle des constructions statiques (qui pourront être éliminées). Plus particulièrement, nous avons étudié les analyses de temps de liaison implémentées dans les évaluateurs partiels existants pour C [3, 4]. Celles-ci se sont avérées insuffisantes pour mettre à jour les opportunités de spécialisation décrites précédemment dans le paragraphe 4.2. En conséquence, pour pouvoir spécialiser efficacement le RPC, nous avons été amenés à introduire les raffinements suivants dans l'analyse de temps de liaison de Tempo [75, 50] :

– *structures partiellement statiques* : la figure 4.3 montre que certains champs de la structure `xdrs` sont statiques tandis que d'autres sont dynamiques. Une spécialisation efficace nécessite que nous soyons capables d'accéder aux champs statiques lors de la spécialisation. Sans une telle fonctionnalité, la structure toute entière doit être considérée dynamique de manière conservatrice, et les contrôles répétés de non-débordement du tampon ne peuvent pas être éliminés.

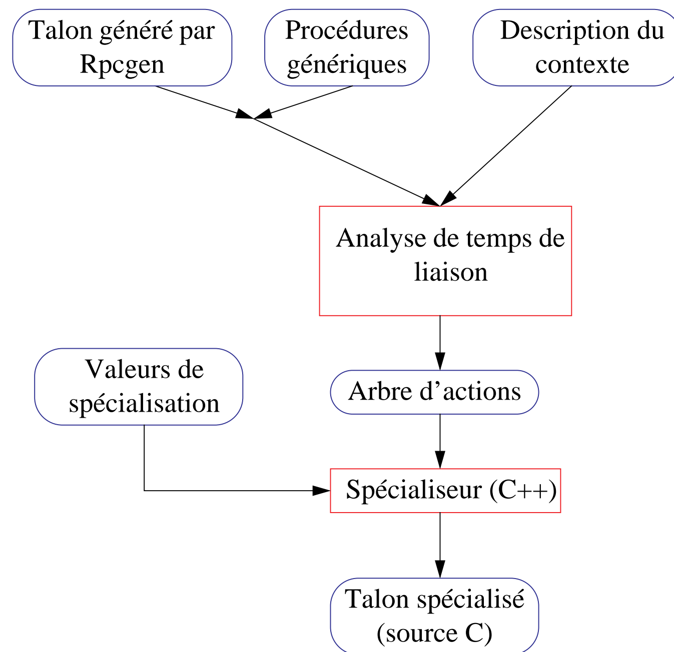

FIG. 4.6 – *Spécialisation "off-line" à la compilation*

- *sensibilité au flot* : l'éventualité d'erreurs lors du décodage d'un message reçu introduit des conditions dynamiques après lesquelles l'information statique est perdue. Cependant, il est encore possible d'exploiter l'information statique au sein de chaque branche des conditionnelles. Pour cela, le temps de liaison des variables (statiques ou dynamiques) ne doit pas être une propriété globale ; il doit dépendre du point du programme considéré.

- *sensibilité au contexte* : la fonction d'encodage d'un entier est généralement appelée avec des données dynamiques, représentant les arguments du RPC. Cependant, il existe un encodage d'un entier statique dans chaque appel de procédure distante : l'identificateur de la procédure. Il est utile de différencier les deux contextes d'appel, dans le but de ne pas perdre cette opportunité de spécialisation. Une spécialisation efficace nécessite que les appels aux mêmes fonctions avec différents contextes de temps de liaison se réfèrent aux différentes instances de temps de liaison de la définition de cette fonction.

- *retours statiques* : comme nous l'avons vu dans l'exemple de la section 4.2.3, le calcul du code de retour lors de la spécialisation s'appuie sur la capacité à connaître statiquement la valeur de retour d'un appel de fonction même si son argument et ses actions sur le tampon d'entrée/sortie sont dynamiques. Plus généralement, la valeur de retour d'une fonction peut être statique même si ses arguments et ses effets de bord sont dynamiques. Aussi, nous pouvons utiliser la valeur de retour d'un appel de fonction même si l'appel doit être résidualisé.

Tempo repose également sur plusieurs autres analyses de programmes, telles que l'analyse d'*alias* et l'analyse d'effets de bord. Tempo va plus loin que la propagation conventionnelle de constantes et le dépliage, au sens où il n'est pas limité à l'exploitation de valeurs scalaires *de façon intra-procédurale*. Tempo ne traite pas seulement les alias et les structures de données partiellement-statiques, il les propage aussi de façon *inter-procédurale*. Ces caractéristiques sont particulièrement critiques lorsque l'on traite du code système.

En fait, nous avons orienté la conception de Tempo afin qu'il soit plus particulièrement destiné aux logiciels système. Plus concrètement, le RPC est une des applications clés qui ont dirigé sa conception et son implémentation. Tempo est donc à même de spécialiser les opportunités de spécialisation décrites dans la section 4.2. En fait, l'analyse de temps de liaison de Tempo produit un résultat similaire aux figures 4.2 et 4.3.

4.4 Évaluation de performance

Après avoir expliqué les formes de spécialisation effectuées par Tempo, nous présentons l'évaluation du RPC optimisé.

Le programme de test. Nous avons spécialisé le code du client et du serveur de la version de 1984 du RPC de Sun. Le code RPC non spécialisé comprend environ 1500 lignes (hors commentaires) pour le côté client, et 1700 pour le côté serveur. Le programme de test, qui utilise les appels de procédure distants, émule le comportement de programmes parallèles

qui échangent des données structurées de taille importante. Ceci constitue un programme de test représentatif des applications qui utilisent un réseau de stations de travail comme des multiprocesseurs à grande échelle.

Plateformes de mesures. Nos mesures ont été effectuées sur deux plateformes différentes :

- Deux stations de travail Sun IPX 4/50 sous SunOS 4.1.4 avec 32 Mo de mémoire connectées avec un lien ATM à 100 Mbits/s. Les cartes ATM sont du modèle ESA-200 de Fore Systems. Cette plateforme est vieille de 3 ans et est sensiblement moins efficace que des produits récents, à la fois en termes de processeur, latence réseau et bande passante (c.-à-d., 155 Mbits / 622 Mbits).

- Deux machines Pentium PC à 166 MHz sous Linux avec 96 Mo de mémoire et une connexion réseau Fast-Ethernet à 100 Mbits/s. Il n'y avait aucune autre machine sur ce réseau pendant les expérimentations.

Le code spécialisé a été testé sur différents environnements dans le but de vérifier que les résultats que nous obtenons ne sont pas spécifiques à une plateforme particulière. Tous les programmes ont été compilés en utilisant la version 2.7.2 de `gcc` avec l'option `-O2`.

Évaluation des performances. Afin d'évaluer l'efficacité des spécialisations, nous avons effectué deux types de mesures : (i) un micro-test du processus d'encodage chez le client et (ii) un test de niveau application qui mesure le temps total écoulé dans un appel RPC complet (c.-à-d., aller-retour). Le programme de test client boucle sur un simple RPC qui envoie et reçoit un tableau d'entiers. Le but de la seconde expérimentation est de prendre en compte les caractéristiques architecturales telles que le cache, la mémoire et la bande passante du réseau, qui affectent la performance globale de manière significative. Les comparaisons de performance pour les deux plateformes et les deux expérimentations sont décrites à la figure 4.7. Les mesures résultent de la moyenne de 10000 itérations.

Ainsi que nous nous y attendions, la plateforme PC/Linux est toujours plus rapide que la plateforme IPX/SunOs. Ceci est en partie dû à un processeur plus rapide, mais aussi au fait que les cartes Fast-Ethernet ont une bande passante réelle plus élevée et une latence plus petite que nos cartes ATM. Une conséquence de l'élimination d'instructions par le processus de spécialisation est que le fossé entre les plateformes est plus réduit sur le code spécialisé (voir les comparaisons sur l'encodage, figures 4.7-1 et 4.7-2).

Micro-test d'encodage. Les résultats détaillés du micro-test d'encodage sont décrits dans le tableau 4.1. Le code du talon client spécialisé s'exécute jusqu'à 3,7 fois plus vite que le code non spécialisé sur IPX/SunOS, et jusqu'à 3,3 fois sur PC/Linux. Puisque le nombre d'instructions éliminées est linéaire avec la taille du tableau et que l'on a un coût fixe du à l'encodage de l'entête, on peut intuitivement s'attendre à voir l'accélération s'accroître avec la taille du tableau jusqu'à une asymptote. Cependant, sur le Sun IPX l'accélération décroît avec la taille du tableau d'entiers (voir figure 4.7-5). L'explication réside dans le fait que le temps d'exécution du programme est dominé par les accès mémoire et que le Sun

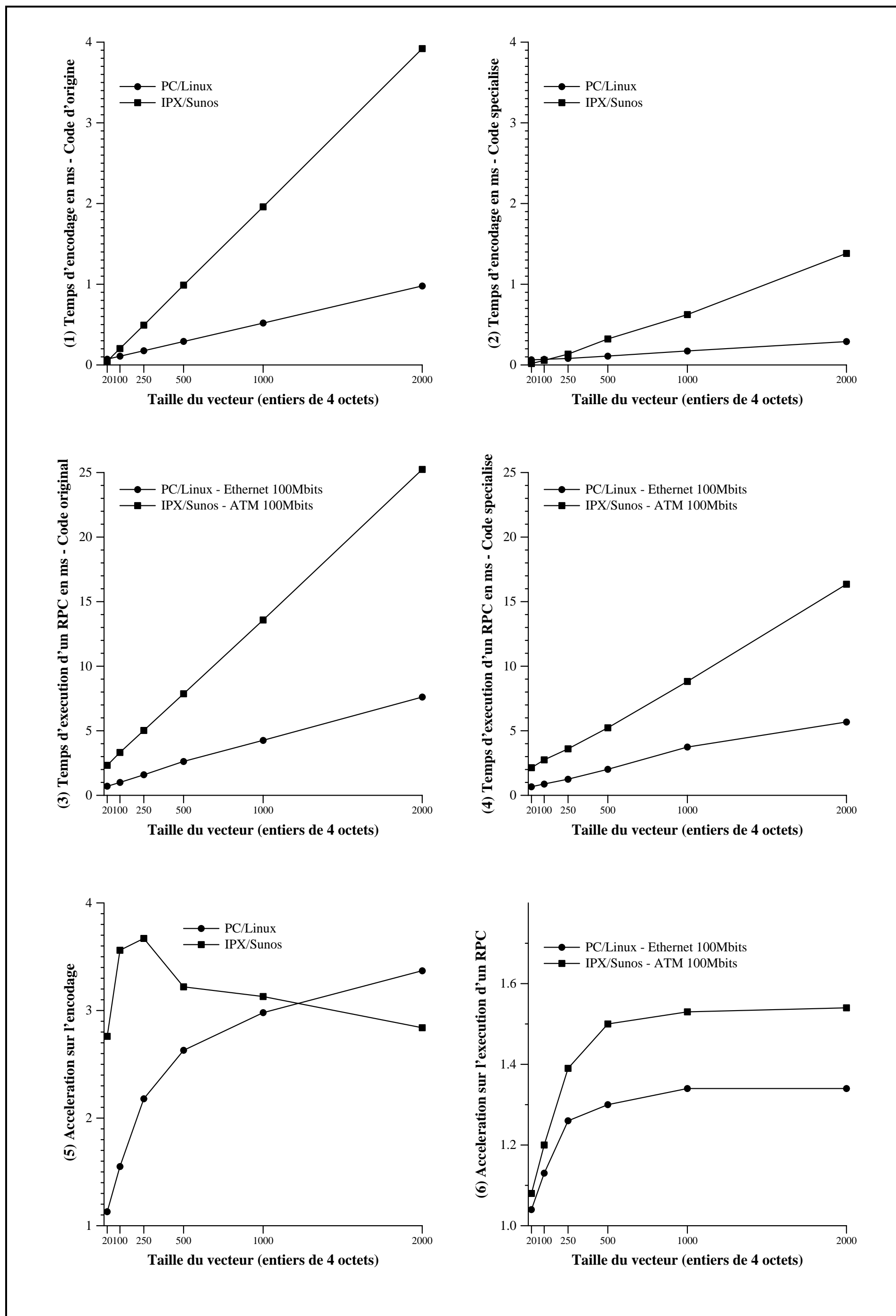

FIG. 4.7 – *Comparaison de performance entre IPX/SunOS et PC/Linux*

IPX possède un cache relativement petit : quand la taille du tableau croît, le temps d'encodage est dû en majeure partie à la copie du tableau d'entiers dans le tampon de sortie ; même si la spécialisation fait décroître le nombre d'instructions utilisées pour encoder un entier, le nombre de copies mémoire reste égal dans le code spécialisé et le code non-spécialisé.

Pour le PC, qui possède un cache de 512 Ko, ce comportement n'apparaît pas ; comme prévu, la courbe d'accélération s'approche d'une asymptote.

Taille du tableau	IPX/SunOs			PC/Linux		
	Original	Spécialisé	Accélération	Original	Spécialisé	Accélération
20	0,047	0,017	**2,75**	0,071	0,063	**1,20**
100	0,20	0,057	**3,50**	0,11	0,069	**1,60**
250	0,49	0,13	**3,75**	0,17	0,08	**2,10**
500	0,99	0,30	**3,30**	0,29	0,11	**2,60**
1000	1,96	0,62	**3,15**	0,51	0,17	**3,00**
2000	3,93	1,38	**2,85**	0,97	0,29	**3,35**

TAB. 4.1 – *Performance de l'encodage client en ms*

RPC complet. Les résultats de ce test de niveau application sont décrits dans le tableau 4.2. Le code spécialisé s'exécute jusqu'à 1,55 fois plus vite que le code non-spécialisé sur IPX/SunOS, et jusqu'à 1,35 fois plus vite sur le PC/Linux. Dans les deux cas on atteint une asymptote. Pour la plateforme IPX/SunOs, l'accélération décroît avec la taille des données à cause des accès mémoire. Pour la plateforme PC/Linux, la raison essentielle est due à la nécessité d'envoyer plusieurs packets Ethernet, ce qui augmente le temps de latence dû au réseau. Enfin, le RPC inclut un appel à `bzero` pour initialiser le tampon d'entrée à la fois du côté client et du côté serveur[2]. Ces initialisations augmentent davantage le surcoût de l'accès mémoire lorsque la taille des données croît.

Taille du code. Ainsi qu'il est décrit dans le tableau 4.3, le code spécialisé est toujours plus gros que le code original. La raison est que le spécialiseur déroule par défaut les boucles d'encodage/décodage de tableau. Notons que le code spécialisé est également plus

2. Ce code n'apparaît pas dans le micro-test de l'encodage

Taille du tableau	IPX/SunOs			PC/Linux		
	Original	Spécialisé	Accélération	Original	Spécialisé	Accélération
20	2,32	2,13	**1,10**	0,69	0,66	**1,05**
100	3,32	2,74	**1,20**	0,99	0,87	**1,15**
250	5,02	3,60	**1,40**	1,58	1,25	**1,25**
500	7,86	5,23	**1,50**	2,62	2,01	**1,30**
1000	13,58	8,82	**1,55**	4,26	3,17	**1,35**
2000	25,24	16,35	**1,55**	7,61	5,68	**1,35**

TAB. 4.2 – *Performance du RPC aller-retour en ms*

Module	Générique	Spécialisé (taille du tableau)					
		20	100	250	500	1000	2000
code client	20004	-					
code client spécialisé		24340	27540	33540	43540	63540	111348

TAB. 4.3 – *Taille des binaires SunOS (en octets)*

Taille du Tableau	PC/Linux				
	Original	Spécialisé	Accélération	Déroulé 250x	Accélération
500	0,29	0,11	**2,65**	0,108	**2,70**
1000	0,51	0,17	**3,00**	0,15	**3,40**
2000	0,97	0,29	**3,35**	0,25	**3,90**

TAB. 4.4 – *Spécialisation avec des boucles d'entiers déroulées 250 fois (temps en ms)*

important pour une taille de tableau petite. Ceci est dû au fait que le code spécialisé contient aussi quelques fonctions génériques non-spécialisées de gestion des cas d'erreur.

Tandis que le déroulement de boucles accroît les tailles de code, il affecte également la localité du cache. Une expérimentation supplémentaire a été menée sur le PC pour mesurer cet effet. Comme dérouler complètement de grosses boucles peut excéder la capacité du cache d'instructions, nous avons déroulé seulement de façon partielle la boucle pour ajuster son corps à la taille du cache. Le code résultant (cf tableau 4.4) montre une détérioration plus faible de performance quand le nombre d'éléments croît.

Cette dernière transformation a été effectuée à la main. Elle montre qu'il est intéressant de permettre le contrôle du déroulement de boucles. Aussi, nous prévoyons d'introduire, dans le futur, cette stratégie dans Tempo.

4.5 Discussion

Dans cette section, nous relatons notre expérience d'utilisation de Tempo (cf section 4.5.1) et de spécialisation d'un logiciel commercial existant (cf section 4.5.2).

4.5.1 Expérience de l'utilisation de Tempo

Tempo est un spécialiseur de programmes en phase de développement active, qui par conséquent évolue en permanence au fur et à mesure des besoins rencontrés. La difficulté majeure de la spécialisation du RPC est que ce travail représente le résultat de multiples itérations sur : l'évaluation d'une version de Tempo, l'expression des améliorations à effectuer et le développement de la version suivante. En ce sens, notre travail n'est pas représentatif d'une utilisation réelle d'un spécialiseur en tant qu'outil figé de génie logiciel.

Plus précisément, les améliorations apportées à Tempo ont été effectuées suivant trois directions : (i) la précision des analyses, ainsi que nous l'avons décrit dans la section 4.3 (ii) l'étendue des constructions du langage C traitées, et (iii) la finesse de la paramétrisation de l'outil. Dans le cadre du RPC, l'enjeu est de spécialiser du code existant sans modifi-

cations ou presque. Cela suppose que Tempo est à même de traiter la plus grande partie des constructions du langage C, même si certaines d'entre elles ne sont pas intéressantes du point de vue recherche et ne posent que des problèmes d'ingénierie. Au cours de notre expérimentation, nous avons été amenés, dans un premier temps, à réécrire de manière systématique certaines parties du code, puis ultérieurement à ré-insérer le code original lorsque les constructions manquantes ont été ajoutées à Tempo.

Le contrôle du comportement de Tempo repose essentiellement sur un fichier de contexte décrivant le temps de liaison des variables d'entrée. Les possibilités initiales de déclaration permises par ce fichier se sont révélées trop limitées pour deux raisons. Premièrement, les données gérées par le RPC ont une représentation complexe reposant sur un ensemble de structures liées par des pointeurs, dont le temps de liaison ne pouvait être décrit. Deuxièmement, l'intégralité des fonctions utilisées par le programme (ex., appels système, malloc) n'est pas présente dans le code source analysé par Tempo. Bien que ces fonctions externes ne soient pas elles-même spécialisées, il est nécessaire de pouvoir décrire leur comportement vis-à-vis des alias et du temps de liaison. Ces deux limitations ont été levées ultérieurement par ajout d'un second fichier de contexte. Ce fichier contient une version simplifiée des fonctions externes qui abstrait leur comportement. Ces fonctions sont utilisées uniquement lors de l'analyse de temps de liaison ; leur contenu n'est toutefois pas résidualisé.

Bien que jusqu'à présent la priorité ait été donnée à la mise en oeuvre de notre "moteur" d'évaluation partielle, une interface utilisateur adaptée est indispensable pour travailler sur des logiciels réels. Tempo permet à l'utilisateur de visualiser les résultats de l'analyse avant la spécialisation. Différentes couleurs sont utilisées pour matérialiser les parties statiques et dynamiques d'un programme, aidant ainsi l'utilisateur à suivre la propagation des entrées déclarées comme connues et évaluer le degré de spécialisation qui peut être obtenu. Après spécialisation, l'utilisateur peut comparer le programme original avec le programme spécialisé, et décider si une spécialisation appropriée a été effectuée.

Comme pour d'autres optimiseurs performants, notamment dans le domaine du parallélisme, l'utilisateur a besoin d'avoir une certaine connaissance sur le processus de spécialisation lui-même. Dans le cas de l'évaluation partielle, la personne a besoin de comprendre des concepts fondamentaux tels que l'analyse de temps de liaison.

4.5.2 Spécialisation de programmes existants

Une importante constatation de notre expérimentation est que le code existant est un défi pour une technique d'optimisation telle que l'évaluation partielle. En effet, comme n'importe quelle autre technique d'optimisation, l'évaluation partielle est sensible à différentes caractéristiques du programme telles que sa structure et l'organisation de ses données. Dans l'état actuel de la technique, spécialiser un programme existant nécessite une certaine connaissance de sa structure et de ses algorithmes. Il requiert également que la personne qui spécialise estime quelles parties du programme doivent être évaluées de façon plus poussée. Ceci est en opposition avec une situation où le même programmeur à la fois écrit et spécialise du code : il peut alors le structurer en ayant à l'esprit la spécialisation.

Une inspection rigoureuse du code spécialisé résultant montre peu de perspectives pour de nouvelles optimisations sans restructuration majeure du code RPC. Cependant, Tempo n'est pas la panacée et nous avons eu occasionnellement à modifier légèrement le code

source original dans le but d'obtenir des spécialisations appropriées. La plupart des modifications consistaient à exposer des opportunités de spécialisation.

Plus précisément, du côté client, il existe une variable appelée `inlen` qui stocke la longueur des données déjà décodées. Lors du décodage du tampon d'entrée, la variable `inlen` est dynamique puisque l'appel de procédure distant peut échouer. Cependant, si aucune erreur ne se produit, `inlen` contient la taille des données résultat attendues. Dans ce cas, nous pouvons connaître la longueur attendue du message d'entrée, notée `expected_-inlen`. Dans la partie décodage, le squelette de code suivant :

```
inlen = <dyn>;
<statements>;
```

est réécrit à la main en

```
inlen = <dyn>;
if (inlen == expected_inlen) {
      inlen = expected_inlen;
      <statements>;
} else
      <statements>;
```

En résultat, dans la branche "then", la valeur connue de la variable `expected_inlen` est affectée à `inlen` ; les constructions suivantes peuvent maintenant exploiter cette valeur supplémentaire. Cependant, la branche "else" préserve la sémantique : elle gère le cas général.

La véritable valeur de `expected_inlen` peut être calculée au moment de la spécialisation par un faux appel encodé à la fonction générique d'encodage/décodage. Nous sommes ainsi capables de spécialiser le décodage client. Cette situation ne s'est réellement produite que deux fois ; elle ne contredit donc pas notre prétention à traiter de façon automatique le code système existant. On peut donc dire que la version courante de Tempo a permis de spécialiser avec succès le RPC Sun.

4.6 Conclusion

On peut considérer que le RPC Sun est représentatif du code système existant, non seulement parce qu'il est mûr, commercial, standard, mais aussi parce que sa structure reflète un souci de qualité de production ainsi qu'un emploi non restreint du langage de programmation C.

Les exemples que nous avons présentés dans la section 4.2 (c.-à-d., la sélection de protocole, le contrôle de débordement du tampon, la propagation du résultat) constituent des exemples typiques des constructions trouvées dans le code système. Le fait que Tempo soit capable de les traiter automatiquement renforce notre conviction qu'un outil de spécialisation a naturellement sa place dans la palette des outils d'ingénierie destinés à la production de systèmes d'exploitation.

Chapitre 5

Une approche déclarative à la spécialisation de programmes

Lors de l'utilisation d'un outil de spécialisation de programmes tel que Tempo, une des difficultés rencontrées par l'utilisateur réside dans la manière de contrôler le processus de spécialisation. Dans le cas du RPC, la gestion de la spécialisation est relativement simple puisque celle-ci est effectuée uniquement à la compilation et qu'il n'existe qu'une version spécialisée du programme. Dans un cadre plus général, lorsque la spécialisation à la compilation et à l'exécution sont mélangées, de nouveaux problèmes apparaissent : comment détecter les opportunités de spécialisation ? sur quelle partie du programme faire porter la spécialisation ? quand activer et désactiver une instance spécialisée ? etc.

À ce jour, les solutions proposées sont ad-hoc. Par exemple, dans [6] des annotations sont introduites dans le source du programme, pour délimiter les fragments à spécialiser, ou pour gérer les versions spécialisées. Mark Leone et Peter Lee proposent de modifier la sémantique de certaines constructions syntaxiques pour déclencher la spécialisation [61]. Ces stratégies visent des aspects très particuliers du processus de spécialisation et beaucoup de problèmes restent inexplorés.

Dans ce chapitre, nous décrivons une approche déclarative à la spécialisation de programmes, dans le contexte des langages orientés objet [102]. Les avantages de notre approche sont les suivants :

– *Un support déclaratif intégré au langage.* Pour ne pas générer d'erreurs, la déclaration de la spécialisation ne doit pas être intrusive. Dans notre approche, elle peut être effectuée sans modification du source du programme par association de déclarations à une classe. On peut considérer que ces déclarations ajoutent de l'information au programme, dans la mesure où elles décrivent un *aspect* du programme [57] orthogonal à sa fonctionnalité.

– *Une approche uniforme à la spécialisation.* Ainsi que nous l'avons décrit précédemment, il existe différents types de spécialisation. Notre approche déclarative permet d'exploiter ces différentes stratégies dans un même cadre.

– *Un support d'exécution flexible.* L'intégration de la spécialisation dans une application requiert un support lors de l'exécution, pour : déclencher la spécialisation (dans le cas de la spécialisation à l'exécution) ; détecter les moments où une version spécialisée n'est plus valide ; décider quelles versions doivent être conservées et pour combien de temps, etc. Dans notre approche, certains de ces aspects sont inférés à partir des déclarations ; d'autres aspects peuvent être déclarés explicitement.

Le travail décrit dans ce chapitre a été réalisé en collaboration avec Eugène-Nicolae Volanski, Charles Consel et Crispin Cowan de l'Oregon Graduate Institue. Dans la section 5.1, nous introduisons le paradigme de classe de spécialisation et nous décrivons de manière informelle leur sémantique. Nous décrivons dans la section 5.2, un exemple d'application des classes de spécialisation à la spécialisation d'un système de fichier. Dans la section 5.3, nous présentons un schéma de compilation appliqué au langage Java. La section 5.4 est consacrée aux aspects relatifs au support d'exécution. Nous concluons dans la section 5.5.

5.1 Les classes de spécialisation

Nous proposons une extension aux langages orientés objet permettant d'exprimer la spécialisation d'une façon déclarative. Dans notre approche, l'utilisateur déclare quels composants du programme sont spécialisables et pour quels contextes. Dans le cadre d'un langage orienté objet, cela revient à indiquer quelles méthodes sont spécialisables, et par rapport à quelles variables. À partir de ces déclarations, un compilateur dédié détermine comment les versions spécialisées sont produites et gérées. Le résultat de la compilation est une version étendue du programme initial, capable de déclencher la spécialisation quand nécessaire, et de remplacer les versions spécialisées d'une manière transparente.

L'unité de déclaration est la *classe de spécialisation*. Elle enrichit l'information concernant une classe existante. Le rapport entre les classes normales et les classes de spécialisation est défini par une forme d'héritage reposant sur les *classes de prédicats* introduites par Chambers [27]. Une conséquence importante de cette technique est la possibilité d'exprimer la spécialisation incrémentale [33] par l'héritage entre les classes. Dans cette optique, la spécialisation d'une classe n'est pas fixée — elle évolue au fur et à mesure que les valeurs de spécialisation deviennent disponibles.

Pour configurer une application à un contexte d'utilisation donné, il suffit de déclarer séparément une classe de spécialisation. Le comportement dynamique concernant la spécialisation est dérivé de ces déclarations par le compilateur de classe de spécialisation. Dans la suite de ce chapitre, nous présentons un processus de compilation pour le langage Java. Cependant, le paradigme des classes de spécialisation est relativement indépendant d'un langage donné et peut être appliqué à d'autres langages objets.

La syntaxe des classes de spécialisation est décrite dans la figure 5.1. Cette syntaxe est définie comme une extension à Java : les non-terminaux qui ne sont pas définis (*identifier, integer, class_name, variable_name, method_name, method_prototype* et *method_definition*) sont ceux de Java.

```
sc_decl = [runtime] spec sc_name parent_decl [cache_decl] { (pred_decl ;)+ (method_decl ;)* }
parent_decl = specializes [class] class_name |
              extends [spec] sc_name
pred_decl = variable_name == value |
            variable_name |
            sc_name variable_name
method_decl = method_prototype |
              method_definition
cache_decl = cached cache_strategy [[ integer ]]
cache_strategy = LRU |
                 Amortization |
                 Priority |
                 BestFit |
                 ...
sc_name = identifier
```

FIG. 5.1 – *Syntaxe des classes de spécialisation*

Sémantique des classes de spécialisation

Dans cette section, nous décrivons de manière informelle la sémantique des classes de spécialisation.

Dans le langage Java, il existe deux formes de relations entre les classes et les interfaces. Premièrement, une classe (ou une interface) peut étendre (*extend*) une autre classe (ou respectivement une interface). Deuxièmement, une classe peut également implémenter (*implement*) certaines interfaces. Dans ce même esprit, nous avons défini une troisième forme de relation, *specialize*, qui permet d'attacher une classe de spécialisation à une classe standard.

L'ensemble des classes de spécialisation attachées à une classe C représente une hiérarchie de l'ensemble des états de spécialisation atteignables. La classe C est appelée la classe racine. Chaque classe de spécialisation spécialise soit une classe standard, soit étend une autre classe de spécialisation.

Tous les champs (variables et méthodes) apparaissant dans une classe de spécialisation doivent être définis dans la classe racine correspondante. En d'autres termes, une classe de spécialisation ne peut pas ajouter de nouveaux champs à une classe. Ceci s'explique par le fait que les classes de spécialisation ne sont pas un moyen d'ajouter de nouvelles fonctionnalités, mais permettent au contraire d'adapter une classe à un contexte précis. Pour définir le contexte de spécialisation, une classe de spécialisation S impose des contraintes sur les variables d'une classe C en définissant une liste de prédicats sur les états de l'objet. Un objet de la classe C est considéré comme étant dans un état de spécialisation S lorsque tous les prédicats de la classe de spécialisation S sont valides. Lorsque c'est le cas, les méthodes spécialisées définies par S remplacent les versions génériques de C.

Un prédicat peut référer à un état local de l'objet, mais également à des états non lo-

caux lorsqu'ils expriment des contraintes sur des objets inclus. Dans notre implémentation courante, nous ne considérons pas les variables de type tableau ou classe [1].

Un prédicat sur une variable non objet peut être valide à la compilation ou à l'exécution, suivant que celui-ci peut être complètement évalué ou non lors de la compilation. La syntaxe pour un prédicat valide à la compilation est "*variable_name*==*value*;", dans laquelle *value* est une constante spécifiée lors de la compilation. La syntaxe pour un prédicat valide à l'exécution est simplement "*variable_name*;". Ce prédicat est valide pour toute valeur appartenant au type de *variable_name*.

Par ailleurs, un prédicat peut être également classé en fonction de sa durée de vie, c.-à-d., s'il peut ou non changer de valeur lors de l'exécution. Si sa valeur ne peut pas changer, le prédicat est qualifié de *stable*. Si sa valeur peut changer, le prédicat est qualifié d'*instable*. Dans ce dernier cas, le support d'exécution (généré par le compilateur) est chargé de détecter les changements d'état de spécialisation. Il n'y a pas de distinction, au niveau syntaxique, entre les prédicats stables et instables. En fait, dans un langage tel que Java, il est simple de déterminer statiquement quels prédicats ne sont pas susceptibles d'être invalidés; il suffit de vérifier que les variables référencées ne sont jamais affectées en dehors des constructeurs.

Un prédicat sur une variable de type objet d'une classe *C1* requiert que le (sous-)objet soit dans un état de spécialisation spécifique *S1*, où *S1* est un état de spécialisation pour la classe *C1*. Ce prédicat se déclare "*S1 variable_name*;", de manière similaire à une déclaration de type en Java. En fait, on peut considérer que la variable est redéfinie en un type plus précis.

Nous avons fait le choix délibéré de restreindre les prédicats à ces formes simples, parce qu'elles sont directement exploitables par un spécialiseur de programmes. Ces restrictions font que le spécialiseur peut automatiquement générer une version spécialisée d'une méthode à partir de sa version générique. Toutefois, il est à noter que la syntaxe permet également à un utilisateur de spécifier l'utilisation d'une méthode Java pour le cas où la spécialisation serait faite manuellement.

Les constructions que nous avons décrites précédemment permettent la déclaration du comportement du programme vis-à-vis de la spécialisation. Il est également possible d'influencer le comportement de la spécialisation à l'exécution via les mots-clé `runtime` et `cached`. Leur signification est détaillée dans la section 5.4.

5.2 Spécialisation d'un système de fichier

Pour illustrer l'utilisation des classes de spécialisation, nous prenons comme exemple l'optimisation d'un système de fichiers par spécialisation incrémentale telle quelle l'a été décrite dans [84]. Cette expérimentation a été réalisée sur le système de fichiers de HP-UX. Dans cette étude, un certain nombre de prédicats stables et instables [2] ont été recensés et exploités par spécialisation manuelle, avec de très bons résultats. La validité des versions spécialisées était assurée par des fragments de code appelés *gardes*, insérés manuellement

1. Le programme Java peut contenir des tableaux ou des variables de type classe. Toutefois, celles-ci ne peuvent pas être référencées au sein d'un prédicat.

2. Dans cette étude, les prédicats stables et instables étaient appelés respectivement invariants et quasi-invariants.

dans le source du système. Les gardes étaient responsables à la fois de l'installation des versions spécialisées et de la restauration des versions génériques.

L'exemple présenté ci-dessous est un fragment de programme Java, directement inspiré du système de fichier de HP-UX. Nous définissons une classe de spécialisation pour l'objet principal, et nous montrons comment la gestion des versions spécialisées est dérivée des déclarations.

Les paradigmes clés d'un système de fichiers de type Unix sont le fichier et l'inode[3]. Deux structures de données implémentent ces entités : le descripteur de fichier et le descripteur d'inode. En fait, le concept de fichier Unix est une abstraction très générale pour des flots de caractères (*character streams*). Il couvre beaucoup plus que les fichiers habituels sur le disque, car il inclut : des sockets, des périphériques, des tuyaux (*pipes*), etc. Le type précis du fichier n'est pas mémorisé dans le descripteur de fichier, mais dans le descripteur de l'inode correspondant, qui est référencé dans chaque descripteur de fichier.

Un descripteur d'inode (voir figure 5.2) contient, entre autres éléments, l'information de type et des droit d'accès. En fait, l'information de type (qui est mémorisée à la fois dans le drapeau `pipe` et dans le masque `mode`) ne change jamais après la création de l'inode. Les droits d'accès (qui font aussi partie de `mode`) peuvent être changés par la méthode `chmod()`. Les fonctions les plus importantes de l'inode sont les méthodes de lecture et d'écriture (la méthode `write()` est omise dans la figure). La méthode de lecture est générique dans le sens où elle contient des instructions pour les différents type d'inodes.

La figure 5.3 présente la définition d'une classe de spécialisation qui instancie la définition générique de l'inode pour le cas d'un fichier sur disque, accessible uniquement en lecture (*read-only*). Ainsi qu'il l'a été montré dans [84], cette spécialisation permet d'obtenir un facteur d'accélaration de 3, par élimination des contrôles portant sur le type de l'inode et sur l'accès en exclusion mutuelle. La classe de spécialisation spécifie qu'à chaque fois que les variables de l'inode contiennent les valeurs précisées, toute opération de lecture doit invoquer une version spécialisée par rapport à ces valeurs. Si l'état change à nouveau, la version générique doit être réinstallée.

5.3 Compilation des classes de spécialisation

Le compilateur prend en entrée une classe Java et un ensemble de classes de spécialisation. Il produit une nouvelle définition de classe qui encapsule le comportement vis-à-vis de la spécialisation. Nous détaillons maintenant le processus de compilation sur l'exemple précédent de spécialisation de l'inode.

Le compilateur scinde les fonctions de l'objet inode original en plusieurs objets (voir figure 5.4). Un *objet encapsulant* (défini en figure 5.5) reproduit les fonctions d'origine de l'inode, à la différence près qu'il délègue les appels aux méthodes spécialisables vers un *objet implémentation*. En fait, deux objets implémentations sont générés : un objet générique (voir figure 5.6) et un objet spécialisé (voir figure 5.7), qui correspond à la classe de spécialisation `ReadFileInode` (voir figure 5.3).

Les objets implémentation sont abstraits par une interface nommée `Mutable` (voir figure 5.8). L'objet inode est étendu par le compilateur de façon à intégrer ce niveau d'abs-

3. Dans un souci de simplification, nous faisons abstraction des v-nodes.

```
public class Inode extends Object {
   short mode; // rwx info + fichier standard/pilote d'E/S
   boolean pipe; // vrai si inode est un "pipe"
   // ...

   public Inode(short mod, boolean pip) { // constructeur
      mode = mod;
      pipe = pip;
   }

   public void chmod(short mod) { /* change le mode d'accès
                                        (rwx bits) */
      // ... vérifications diverses
      mode = (mode & ~RWX_BIT) | mod;
   }

   public int read() { // lecture d'un octet
        // Version générique: fichier disque, pipe, pilote
        // ... vérification du droit de lecture
   }
};
```

FIG. 5.2 – *La définition d'origine de l'inode*

```
spec ReadOnlyInode specializes class Inode {
   mode == I_REGULAR | R_BIT;
   pipe == false;

   read(); /* produit une version simplifiee:
                  fichier disque, lecture */
}
```

FIG. 5.3 – *Une classe de spécialisation pour l'inode*

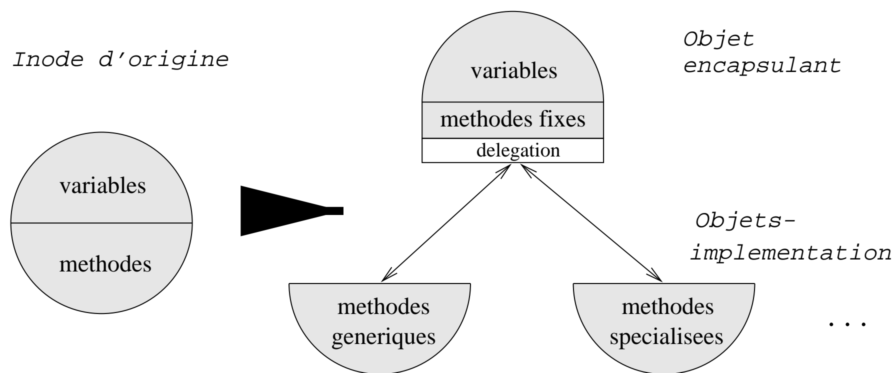

FIG. 5.4 – *L'inode compilé*

traction. Il inclut une nouvelle variable, `scImpl`[4], qui réfère à l'objet implémentation courant. La méthode spécialisable `read()` de l'objet encapsulant est remplacée par une méthode de délégation qui invoque tout simplement l'objet implémentation courant. Pour que les méthodes spécialisables puissent accéder aux champs d'origine de l'inode, une référence arrière vers l'objet encapsulant (la variable `scEncl`) est introduite ; toute référence à ces champs (via `this`) dans les méthodes spécialisables est substituée par une référence à l'objet encapsulant (via `scEncl`).

Afin de détecter des changements d'état qui peuvent changer l'implémentation courante, les affectations de la variable `mode` sont *gardées* : toute affectation à cette variable est substituée par un appel à une nouvelle méthode, nommée `scSet_mode()`. En plus de l'affectation proprement dite, cette méthode vérifie si la nouvelle valeur invalide les méthode spécialisées. En cas d'invalidation de l'état de spécialisation, la méthode `scNotify()` est appelée pour déterminer une nouvelle classe de spécialisation. Plus précisément, elle procède par évaluation de chaque classe de spécialisation associée à l'inode, et vérifie si les prédicats sont valides. Quant une classe de spécialisation est trouvée (en dernier recours, on sélectionne la version générique), l'implémentation courante est mise à jour via la méthode privée `scSwitchToImpl()`.

Lorsque l'implémentation est changée, `scSwitchToImpl()` produit une nouvelle instance de `InodeImpl` ou `ReadFileInodeImpl`. Dans l'exemple de l'inode, toutes les méthodes spécialisées sont produites lors de la compilation. Dans le cas d'une spécialisation à l'exécution, le constructeur d'une nouvelle implémentation invoquerait le spécialiseur à la volée.

On peut remarquer que les affectations à la variable `pipe` ne sont pas gardées, même si celle-ci est impliquée dans un invariant ; une simple analyse statique a déterminé que cette variable n'est jamais affectée en dehors des constructeurs.

Les affectations dans les constructeurs ne sont gardées non plus, car l'état n'est pas encore complètement initialisé. En revanche, les constructeurs sont étendus pour appeler la méthode `scNotify()` juste avant le retour. En effet à ce stade, l'initialisation de l'état

4. Comme convention de nommage, tous les champs (variables ou méthodes) introduits par le compilateur commencent avec le préfixe "`sc`".

```
public class Inode extends Object implements Mutable {
   // copie les variables d'origine:
   short mode; // bits 'rwx' + periph./fichier
   boolean pipe; // Vrai pour un pipe

   // rajoute quelques nouvelles variables:
   InodeImpl scImpl; // implémentation courante
   List scClients; // liste des clients 'Mutable'

   // reecrit les constructeurs:
   public Inode(short mod, boolean pip) {
      // copie le corps d'origine:
      mode = mod;
      pipe = pip;

      // rajoute un code d'initialisation:
      scClients = null;
      scNotify();
   }

   // copie les méthodes d'origine, tout en
   // réécrivant les affectations à 'mode'
   void chmod(short mod) {
      // . . . différents tests
      scImpl.scSet_mode((mode & ~RWX_BIT) |
                              mod);
   }

   // implémente l'interface Mutable:
   public void scNotify() {
      // Détermine une nouvelle classe de
      // spécialisation
      if (mode == (I_REGULAR | R_BIT) &&
            !pipe)
         scSwitchToImpl("ReadOnlyInode");
      else
         scSwitchToImpl("Inode");
      // propage la notification vers les clients:
      scClients.scNotify();
   }
   protected void scSwitchToImpl(String spec)
      {. . .}
   public void scAttach(Mutable m)
      {. . .} // rajoute un client
   public void scDetach(Mutable m)
      {. . .} // supprime un client
   public boolean scIsA(String spec)
      {. . .} // inspecte l'état

   // délègue les méthodes spécialisables:
   public int read() { return scImpl.read(); }
};
```

FIG. 5.5 – *L'objet encapsulant de l'inode*

```
import sclib.*;

class InodeImpl extends Impl {
   // partie fixe pour implémentations génériques:
   protected Inode scEncl; // l'objet encapsulant
   public InodeImpl(Inode i) { // constructeur
      scEncl = i;
   }

   // gardes:
   short scSet_mode(short new_mode) {
      scEncl.mode = new_mode; // l'affectation
      if(new_mode ≠ I_REGULAR | R_BIT)
         scEncl.scNotify(); // informe l'Inode
      return new_mode;
   }

   // méthodes spécialisables:
   int read() {
        // Version générique:
        // avec 'scEncl' substitué pour 'this'
   }
};
```

FIG. 5.6 – *L'implémentation générique de l'inode*

```
import sclib.*;

class ReadFileInodeImpl extends InodeImpl {
   // redéfinition des gardes:
   // (ici aucune)

   // redéfininition des méthodes specialisables:
   int read() {
      // . . . version simplifiée:
      // fichier disque, read-only
   }
};
```

FIG. 5.7 – *L'implémentation spécialisée de l'inode*

```
package sclib;

public interface Mutable {
   public void scNotify();
   private void scSwitchToImpl(String sc);
   public void scAttach(Mutable m);
   public void scDetach(Mutable m);
   public boolean scIsA(String sc);
};
```

FIG. 5.8 – *L'interface* `Mutable`

est achevée, et donc l'implémentation courante peut être choisie de manière appropriée.

5.4 Aspects relatifs au support d'exécution

Toute instance de l'objet spécialisable (le fichier, dans notre exemple) commence son exécution avec l'implémentation la plus générique. Les affectations aux champs des prédicats instables, effectuées par l'intermédiaire des gardes, change éventuellement l'implantation de cette instance.

Il est à noter que différentes instances de l'objet spécialisable changent d'implantation de manière indépendante. Le seul surcoût introduit dans le chemin critique par la gestion de plusieurs implémentations spécialisées est un appel de méthode virtuelle. Le surcoût des gardes est payé uniquement lors du changement des prédicats instables — une opération censée être peu fréquente.

5.5 Conclusion

Les classes de spécialisation constituent une approche déclarative à la spécialisation de composants logiciels. Elles permettent au programmeur de spécifier d'une manière simple et élégante quels composants d'une application sont à spécialiser et pour quels contextes d'utilisation. Cette approche englobe dans un même cadre différentes techniques de spécialisation existantes (à la compilation, à l'exécution, manuelle ou automatique, incrémentale, . . .). Enfin, le processus de spécialisation est décrit de manière séparée au programme d'origine, ce qui évite l'introduction d'erreurs à la suite d'une modification.

Nous appliquons actuellement les classes de spécialisation à l'optimisation des bibliothèques standard Java, notamment en ce qui concerne le graphique et le réseau. Notre approche consiste à traduire vers du C, le code Java produit par le compilateur de classes de spécialisation. Le code C est ensuite spécialisé par Tempo. Dans ce but, nous avons conçu un environnement Java, appelé Harissa [70]. Harissa offre un translateur de code intermédiaire vers C et un support d'exécution qui permet de mélanger au sein d'un même programe du code interprété et compilé.

Les classes de spécialisation ont été conçues afin d'optimiser des composants système génériques [35]. Nous étudions leur application sur un sous-ensemble du langage C++ avec pour objectif de spécialiser les IPC de CHORUS. L'intérêt de cette approche est de pouvoir exploiter des opportunités d'optimisation relatives à des prédicats instables et qui requièrent une spécialisation à l'exécution. Par conséquent, les classes de spécialisation représentent une approche complémentaire aux stratégies d'optimisation par hiérarchie de classes utilisées jusqu'à présent dans les systèmes d'exploitation reposant sur une technologie objet (ex., [48, 25, 90, 44]).

Chapitre 6

Conclusion et perspectives

Dans ce document, nous avons suivi une démarche verticale pour décrire notre contribution. À partir d'un problème précis, à savoir la conception d'un système tolérant les fautes, nous avons abordé la problématique de la conception de systèmes extensibles, puis nous avons proposé des solutions à la construction de systèmes adaptatifs via la conception de composants systèmes spécialisables.

Notre expérience montre que la conception de systèmes extensibles et adaptatifs ne relève pas d'un domaine unique, ou plutôt que les solutions sont à rechercher dans la fertilisation croisée de plusieurs domaines de recherche qui vont de la conception de noyaux jusqu'à la transformation de programmes.

Nous concluons la présentation de notre travail en décrivant les perspectives offertes par la conception de systèmes extensibles et adaptatifs.

Systèmes extensibles

Le concept de micro-noyau a représenté une première proposition à la construction de systèmes modulaire et extensibles. Bien que ce concept ait prouvé son intérêt, sa viabilité d'un point de vue industriel est plus discutable. Dans le domaine des systèmes embarqués ou spécifiques, deux micro-noyaux, Chorus [44] et Qnx [49], sont commercialisés. Leur succès est entre autres dû à leur degré élevé de modularité interne, ce qui permet de configurer le noyau aux besoins précis des applications.

Dans le domaine des systèmes d'usage général, l'utilisation des micro-noyaux se solde par un échec relatif. À notre connaissance, plusieurs systèmes industriels de type Unix ont été implémentés sur des micro-noyaux : Mix [14] (une version de SVR4) et Chorus/Fusion (SCO Unix) au-dessus du micro-noyau Chorus, et OSF/1 au dessus d'une version de Mach 3.0. Toutefois, ces Unix ont le désavantage majeur d'offrir des performances sensiblement inférieures à celles d'un système monolithique [37]. On peut citer également l'expérience du projet Workplace chez IBM dont l'objectif était de remplacer l'ensemble des systèmes maison (OS/400, OS/2, AIX). Ce système devait être multi-personnalité et apte à s'exécuter sur les différentes machines du constructeur depuis le portable jusqu'au mini-ordinateur. Ce projet n'était pas un projet de recherche, mais un réel projet de développement reposant sur

les travaux de D. Julin sur la version multi-serveur de Mach/BSD [97]. Le projet a été arrêté principalement pour des raisons de performances, notamment en raison du coût du changement d'espace d'adressage entre tâches [86]. Une deuxième raison de leur échec réside dans les conflits entre les différentes sémantiques des personnalités système ; ceci rend complexe la conception d'une structure de système qui préserve la performance [41].

Pour autant, il existe un réel besoin d'extensibilité dans les systèmes d'usage général. Certains mécanismes tels que la tolérance aux fautes, la migration de processus ou le support du multimedia ne sont pas intégrables simplement dans un système d'usage général [73]. D'une part, ces mécanismes sont très spécifiques à une classe d'application donnée ; ils ne sont par conséquent utiles qu'à peu d'utilisateurs. D'autre part, les choix d'implémentation d'un mécanisme donné sont également très dépendants des besoins de l'application. Il en résulte qu'il est très difficile de choisir et d'intégrer une unique implémentation dans le noyau.

Pour les raisons citées plus haut, notre travail sur DP-Mach et d'autres propositions récentes dans le domaine des noyaux de systèmes extensibles [16, 40] visent à offrir des extensions de granularité fine, relatives à une application ou un groupe d'applications données. Une deuxième tendance vise à ce que les interfaces exportées soient de plus en plus proche du coeur du noyau, voire de la machine physique. Il en découle des problèmes de sûreté des extensions qui ne peuvent pas toujours être résolus par les techniques classiques de protection reposant sur le matériel. C'est un des problèmes que résoud la spécialisation automatique de composants système.

Spécialisation de composants système

La spécialisation de composants génériques est une technique qui reconcilie les objectifs contradictoires de généralité et de performance. Elle permet d'adapter un composant logiciel à un contexte d'utilisation précis. Le premier bénéfice est un gain de performance, puisque les opérations génériques sont éliminées dans le programme résiduel. En outre, lorsque la spécialisation est automatique, elle préserve la sémantique du programme générique et son degré de sûreté. Cette propriété permet la réutilisation de composants système existants, donc matures et d'un degré de sûreté satisfaisant.

La problématique de l'utilisation d'un spécialiseur (tel que l'évaluateur partiel Tempo), réside dans l'aptitude de cette technologie à traiter des composants système réels. Le problème est relatif d'une part à la taille des logiciels, et d'autre part à la complexité des constructions du langage C généralement utilisées par les programmeurs système.

À ce titre, notre étude sur la spécialisation du RPC de Sun apporte une contribution importante. Nous montrons que l'évaluation partielle peut être appliquée à un langage aussi complexe que C, et qu'il est possible de traiter du code système réel non-spécifiquement écrit en vue de la spécialisation. Ceci montre que l'évaluation partielle entre dans une phase de relative maturité. On peut donc penser que notre technologie va sortir des laboratoires de recherche pour pouvoir être appliquée dans l'industrie. D'ores et déjà, plusieurs sociétés se sont portées candidates pour évaluer l'utilisation de Tempo dans le développement de leur produits.

À court terme, notre étude sur le RPC peut être utilisée pour optimiser des applications distribuées réelles. Des candidats intéressants sont par exemple PVM ou le système

de gestion de fichiers NFS. En complément, il est également envisageable de spécialiser les protocoles réseau faisant partie du noyau (ex., TCP/IP, les pilotes d'E/S). Ces protocoles possèdent également une structure en couche et sont analogues aux protocoles que nous avons spécialisés. Notre expérience est par conséquent généralisable à la totalité d'un sous-système réseau, ce qui permettrait de générer des composants réseaux spécifiques aux besoins d'une application donnée.

À plus long terme, un objectif ambitieux est de construire un système complet reposant sur des composants spécialisables. Au moins deux voies s'ouvrent actuellement à notre réflexion. Une première approche est de considérer l'étude d'une bibliothèque de composants système existants telle que la bibliothèque Flux [42]. Cette approche serait assez similaire à notre étude sur le RPC, puisqu'elle consisterait à exhiber les opportunités de spécialisation sur des composants écrits en C. Une seconde approche réside dans la spécialisation de composants système écrits dans le langage Java. En raison de sa propreté, ce langage remporte un succès dans de nombreux domaines d'applications, notamment dans l'implémentation de systèmes d'exploitation (ex., Java OS). De part le caractère orienté-objet du langage, et la rapidité de développement imposée par ce domaine en pleine expansion, les composants logiciels Java ne sont pas encore optimisés et recèlent de multiples possibilités de spécialisation. Cette seconde approche est par conséquent particulièrement prometteuse. Pour ce faire, nous étudions la spécialisation de programmes C générés par le traducteur de code intermédiaire Java vers C que nous avons développé dans Harissa [70]. Plus précisément, nous enrichissons les analyses implémentées dans Tempo afin que celui-ci traite efficacement les constructions spécifiques au support des objets.

Vers une généralisation des classes de spécialisation

Dans l'état actuel de notre technologie, Tempo peut être considéré comme un moteur de spécialisation : son utilisation peut être parfois difficile, lorsque différents types de spécialisation (à la compilation et à l'exécution) sont utilisés conjointement au sein d'une même application. Pour simplifier le contrôle de la spécialisation, nous avons introduit un formalisme, les classes de spécialisation, qui permet à un utilisateur de déclarer le comportement de la spécialisation de manière indépendante du programme.

En fait, les classes de spécialisation constituent un cas particulier de langage de description du processus d'adaptation. On retrouve dans ce langage trois facettes, expression du contexte, analyse des opportunités de spécialisation, gestion/guidage du spécialiseur, qui permettent de décrire les trois phases du processus d'adaptation. Certaines facettes de ce langage sont relativement simples, notamment celle relative à l'observation du contexte. En fait, ce langage est dédié à la description du processus d'adaptation pour une classe d'applications ne nécessitant que des actions de spécialisation.

Dans le futur, nous envisageons d'enrichir ce langage pour décrire des possibilités d'adaptation plus complexes. Le domaine d'application visé est notamment celui des télécommunications. Dans ce contexte, les applications ont une durée de vie assez longue et doivent être reconfigurées pour suivre les besoins de l'exploitation. Ceci demande de pouvoir décrire des actions d'adaptation relatives par exemple au redimensionnement de l'application ou à la migration de processus. L'intérêt de cette approche est de ne pas modifier le programme original ; le processus d'adaptation est déclaré indépendamment et peut

être aisément modifié au cours de l'évolution du système. Nous espérons ainsi faciliter le développement de systèmes complexes.

Bibliographie

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proc. of Usenix 1986 Summer Conference*, pages 93–112, juillet 1986.

[2] M. Ahamad and L. Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proc. of 8th Symposium on Reliable Distributed Systems*, pages 2–11, Seattle (WA), octobre 1989.

[3] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, San Francisco, CA, USA, juin 1992. Yale University, New Haven, CT, USA. Technical Report YALEU/DCS/RR-909.

[4] L.O. Andersen. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 47–58. New York: ACM, 1993.

[5] D.P Anderson. Metascheduling for continuous media. *ACM Transactions on Computer Systems*, 3(11):226–252, août 1993.

[6] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [83], pages 149–159.

[7] J.P. Banâtre, M. Banâtre, G. Lapalme, and Fl. Ployette. The design and building of ENCHERE, a distributed electronic marketing system. *Communications of the ACM*, 29(1):19–29, janvier 1986.

[8] J.P Banâtre, M. Banâtre, and G. Muller. Main aspects of the Gothic distributed system. In *European Teleinformatics Conference on Research into Networks and Distributed Applications*, pages 747–760, Vienna, Austria, avril 1988.

[9] J.P. Banâtre, M. Banâtre, and G. Muller. Architecture of fault-tolerant multiprocessor workstations. In *Second Workshop on Workstation Operating Systems*, pages 20–24, Pacific Groove (CA), septembre 1989. IEEE Computer Society.

[10] M. Banâtre, P. Heng, G. Muller, and B. Rochat. How to design reliable servers using fault tolerant micro-kernel mechanisms. In *USENIX Mach Symposium*, pages 223–231, Monterey, California, novembre 1991.

[11] M. Banâtre, P. Joubert, C. Morin, G. Muller, B. Rochat, and P. Sanchez. Stable transactional memories and fault tolerant architectures. *Operating System Review*, 25:68–72, janvier 1991.

[12] M. Banâtre and G. Muller. Étude et réalisation d'une architecture multiprocesseur tolérante aux pannes et du système opératoire associé. Rapport de fin de contrat DRET no 90 346, INRIA, 1995.

[13] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. Design decisions for the FTM: A general purpose fault tolerant machine. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems*, pages 71–78, Montréal (Canada), juin 1991.

[14] N. Batlivala, B. Gleeson, J. Hamrick, S. Lurndal, D. Price, J. Soddy, and V. Abrossimov. Experience with SVR4 over Chorus. In USENIX-μKERNELS92 [101], pages 223–241.

[15] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[16] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP95 [96], pages 267–283.

[17] B. Bhargava and S.R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach. In *Proc. of 7th Symposium on Reliable Distributed Systems*, pages 3–12, Columbus (OH), octobre 1988.

[18] D.L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 25(3):35–43, mai 1990.

[19] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[20] C. Bryce. *Étude et mise en œuvre de propriétés de sécurité dans les systèmes informatiques*. PhD thesis, Université de Rennes I, octobre 1994.

[21] C. Bryce, V. Issarny, G. Muller, and I. Puaut. Towards safe and efficient customization in distributed systems. In *Proceedings of 6th ACM SIGOPS European Workshop "Matching Operating Systems to Applications needs"*, pages 57–61, Warden, Allemagne, septembre 1994.

[22] C. Bryce and G. Muller. Matching micro-kernels to modern applications using fine-grained memory protection. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 272–279, San Antonio, TX, USA, octobre 1995. IEEE Computer Society Press.

[23] G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory systems. In *Proceedings of the Fourteenth Symposium on Reliable Distributed Systems*, pages 96–105, Bad Neuenahr, Germany, septembre 1995. IEEE Computer Society Press.

[24] G. Cabillic and I. Puaut. Stardust: an environment for parallel programming on networks of heterogeneous workstations. *Journal of Parallel and Distributed Computing*, 40:65–80, février 1997.

[25] R. Campbell, N. Islam, P. Madany, and D. Raila. Designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, septembre 1993.

[26] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real-time mpeg video audio player. In *Proceedings of the 1995 International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'95)*, pages 151–162, New Hampshire, avril 1995.

[27] C. Chambers. Predicate classes. In *Proceedings of the ECOOP'93 European Conference on Object-oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserstautern, Germany, juillet 1993.

[28] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, février 1985.

[29] J.S. Chase, H.M. Levy, M.J. Feely, and E.D. Lazowska. Sharing and addressing in a single address space system. *ACM Transactions on Computer Systems*, novembre 1994.

[30] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, septembre 1990. ACM Press.

[31] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, janvier 1993. ACM Press.

[32] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, février 1996.

[33] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, Copenhagen, Denmark, juin 1993. ACM Press. Invited paper.

[34] G. Coulson, G.S. Blair, and P. Robin. Micro-kernel support for continuous media in distributed systems. *Computer Networks and ISDN Systems*, 26:1323–1341, 1994.

[35] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel, and E.N. Volanschi. Specialization classes: An object framework for specialization. In *Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, Seattle, Washington, octobre 1996.

[36] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proc. of 10th Symposium on Reliable Distributed Systems*, pages 12–20, Pise (Italie), septembre 1991.

[37] R.W. Dean and F. Armand. Data movement in kernelized systems. In *Proc. of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, pages 27–28, Seattle, avril 1992.

[38] E.N. Elnozahy, D.B. Johnson, and Y.M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical report, School of Computer Science, Carnegie Mellon University, septembre 1996.

[39] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. of 11th Symposium on Reliable Distributed Systems*, pages 39–47, Houston (TX), octobre 1992.

[40] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [96], pages 251–266.

[41] B.D. Fleisch. The failure of personalities to generalize. In HOTOS'97 [52], pages 8–13.

[42] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, pages 38–51, St-Malo, France, octobre 1997.

[43] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunde. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[44] M. Gien. Evolution of the CHORUS open microkernel architecture: The STREAM project. In *Proceedings of Fifth IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS'95)*, Cheju Island, Korea, août 1995. IEEE Computer Society Press. Also Technical Report CS/TR-95-107, Chorus Systemes.

[45] B.J. Gleeson. Fault tolerance: Why should I pay for it. In M. Banâtre and P.A. Lee, editors, *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, volume 774 of *LNCS*, pages 66–77, Le Mont Saint-Michel (France), juin 1993.

[46] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0-201-63451-1.

[47] J. Gray. *Notes on Database Operating Systems.*, volume 60 of *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[48] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. Technical Report SMLI TR-93-14, Sun Microsystems Laboratories, Inc., avril 1993.

[49] D. Hildebrand. An architectural overview of QNX. In USENIX-μKERNELS92 [101], pages 113–126.

[50] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM'97 [81], pages 63–73.

[51] P. Hoschka and C. Huitema. Control flow graph analysis for automatic fast path implementation. In *Second IEEE workshop on the architecture and Implementation of high performance communication subsystems*, Williamsburg, VA, septembre 1993.

[52] *6th Workshop on Hot Topics in Operating Systems*, Cape Cod, Ma, mai 1997. IEEE Computer Society.

[53] W.C. Hsieh, Fiuczynski M.E., Garrett C., Savage S., Becker D., and Bershad B. N. Language support for extensible operating systems. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 127–133, Tucson, AZ, USA, février 1996.

[54] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software - Practice And Experience*, 23(2):201–221, février 1993.

[55] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, juin 1993.

[56] T.T-Y Juang and S. Venkatesan. Crash recovery with little overhead. In *Proc. of 13th International Conference on Distributed Computing Systems*, pages 454–461, Arlington (TX), mai 1991.

[57] G. Kiczales. Aspect-oriented programming. http://www.parc.xerox.com/spl/-projects/aop/, 1996.

[58] E.J. Koldinger, J.S. Chase, and S.J. Eggers. Architectural support for single address space operating systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, Boston, MA, USA, octobre 1992.

[59] R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. In *Proc. of Fall Joint Computer Conference*, pages 1150–1158, Dallas, 1986.

[60] B. Lampson. Atomic transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.

[61] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [83], pages 137–148.

[62] P.A. Lee and T. Anderson. Dependable computing and fault-tolerant systems, vol. 3. In J.C. Laprie A. Avizienis, H. Kopetz, editor, *Fault Tolerance: Principles and Practice*. Springer Verlag, 1990.

[63] P. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proc. of 4th International Conference on Data Engineering*, pages 154–163, Los Angeles (CA), février 1988.

[64] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proc. of 10th Symposium on Reliable Distributed Systems*, pages 1–10, Pise (Italie), septembre 1991.

[65] Sun Microsystem. NFS: Network file system protocol specification. RFC 1094, Sun Microsystem, mars 1989. ftp://ds.internic.net/rfc/1094.txt.

[66] S.J. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, and H. Van Staveren. A distributed operating system for the 1990s. *IEEE Computer*, pages 44–53, mai 1990.

[67] G. Muller, M. Banâtre, N. Peyrouze, and B. Rochat. Lessons from FTM: an experiment in the design & implementation of a low cost fault tolerant system. *IEEE Transaction on Reliability*, pages 332–340, juin 1996. Extended version available as IRISA research report 913.

[68] G. Muller, M. Hue, and N. Peyrouze. Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment. In K. Echtle, D. Hammer, and D. Powell, editors, *Dependable Computing - EDCC1*, volume 852 of *LNCS*, pages 491–508, Berlin (Allemagne), octobre 1994. Springer Verlag.

[69] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, mai 1998. IEEE Computer Society Press. To appear.

[70] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Portland (Oregon), USA, juin 1997. Usenix.

[71] G. Muller and N. Peyrouze. Providing fault-tolerance through inheritance facility. In *ECOOP'93 Workshop on Object-Based Distributed Programming*, Kaiserslautern, juillet 1993.

[72] G. Muller, B. Rochat, and P. Sanchez. A stable transactional memory for building robust object oriented programs. In *EuroMicro 91*, pages 359–364, Vienne (Autriche), septembre 1991.

[73] G. Muller and P. Sens. *Placement Dynamique et Répartition de Charge : application aux systèmes parallèles et répartis*, chapter Migration et points de reprise dans les

systèmes distribués : Techniques de mise en œuvre et mécanismes communs, pages 57–80. INRIA, 1997.

[74] G. Muller, E.N. Volanschi, and R. Marlet. Automatic optimization of the Sun RPC protocol implementation via partial evaluation. In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, pages 105–110, mars 1997.

[75] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM'97 [81], pages 116–125.

[76] G. Muller N. Peyrouze. FT-NFS: an efficient fault tolerant NFS server designed for off-the-shelf workstations. In *Proc. of 26th International Symposium on Fault-Tolerant Computing Systems*, pages 64–73, Sendai (Japan), juin 1996.

[77] G. Necula. Proof-carrying code. In *Conference Record of the* 24th *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 106–116, Paris, France, janvier 1997. ACM Press.

[78] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Usenix Symposium on Operating System Design and Implementation (OSDI)*, pages 229–243, Seattle, Wa, USA, octobre 1996.

[79] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, mai 1981.

[80] S. O'Malley, T. Proebsting, and A.B. Montz. USC: A universal stub compiler. In *Proceedings of Conference on Communication Architectures, Protocols and Applications*, London (UK), septembre 1994.

[81] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, juin 1997. ACM Press.

[82] N. Peyrouze. *Conception et réalisation d'un système de gestion de fichier NFS efficace et sûr de fonctionnement*. Thèse de doctorat, université de Rennes I, septembre 1995.

[83] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, mai 1996. ACM SIGPLAN Notices, 31(5).

[84] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In SOSP95 [96], pages 314–324.

[85] R. Ramsey. *All about administering NIS+*. SunSoft, 1993.

[86] F.L. Rawson. Experience with the development of a microkernel-based, multi-server operating system. In HOTOS'97 [52], pages 2–7.

[87] M. Raynal and J.M. Helary. *Synchronization and Control of Distributed Systems and Programs*. Wiley series in parallel computing, 1990.

[88] B. Rochat. *Une approche à la construction de services fiables dans les systèmes distribués*. Thèse de doctorat, université de Rennes I, février 1992.

[89] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[90] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In USENIX-μKERNELS92 [101], pages 39–70.

[91] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. of Usenix 1985 Summer Conference*, pages 119–130, Portland, juin 1985.

[92] F. Schmuck and J. Wyllie. Experience with transactions in quicksilver. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 239–253, octobre 1991.

[93] M.D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[94] L.M. Silva and J.G. Silva. Global checkpointing for distributed programs. In *Proc. of 11th Symposium on Reliable Distributed Systems*, pages 155–162, Houston (TX), octobre 1992.

[95] J.P. Singh, W.D. Weber, and A. Gupta. Splash : Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, avril 1991.

[96] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, décembre 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[97] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In USENIX Association, editor, *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA*, pages 119–130, Berkeley, CA, USA, janvier 1995. USENIX.

[98] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, août 1985.

[99] Y. Tamir and C. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. of 1984 International Conference on Parallel Processing*, pages 32–41, août 1984.

[100] C.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, mai 1993.

[101] *USENIX - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, Seattle, WA, USA, avril 1992.

[102] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, octobre 1997. ACM Press.

[103] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, décembre 1993. ACM Operating Systems Reviews, 27(5), ACM Press.

[104] W.G. Wood. A decentralised recovery control protocol. In *Proc. of 11th International Symposium on Fault-Tolerant Computing Systems*, pages 159–164, Portland (OR), juin 1981.