N° d'ordre: 1943

THÈSE

Présentée devant

l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Eugen N. VOLANSCHI

Équipe d'accueil : IRISA
École Doctorale : Sciences Pour l'Ingénieur
Composante universitaire : IFSIC

Titre de la thèse :

Une approche automatique à la spécialisation de composants système

soutenue le 26 Février 1998 devant la commission d'examen

M. :	Jean-Pierre	BANÂTRE	Président
MM. :	Pierre	COINTE	Rapporteurs
	Santosh	SHRIVASTAVA	
MM. :	Gilles	MULLER	Examinateurs
	Charles	CONSEL	

Remerciements

Je tiens à remercier :

Jean-Pierre Banâtre, qui m'a fait l'honneur de présider ce jury.

Pierre Cointe et Santosh Shrivastava, d'avoir bien voulu accepter la charge de rapporteur.

Charles Consel, de m'avoir proposé ce sujet de pointe, et de m'avoir encadré pendant toute cette thèse avec une haute exigence, tout en me laissant beaucoup de liberté d'action. Sans son enthousiasme, sans son optimisme incorrigible ou sans sa receptivité, une bonne partie de cette thèse n'aurait jamais été écrite.

Gilles Muller, d'avoir co-dirigé cette thèse ; sans son solide soutien d'expert en systèmes je ne me serais jamais aventuré en dehors des pistes balisées, entre langages et systèmes.

Eric Pillevesse, pour la confiance qu'il m'a montrée a maintes reprises tout au long du contrat avec le Centre National d'Études en Télécommunications.

Je remercie également tous les membres des équipes Compose et Lande, pour de très nombreuses intéractions, pour leur permanente disponibilité, pour leurs corrections sur ce manuscrit, pour le développement et la maintenance de Tempo. Et plus généralement, pour avoir su rendre ces équipes si agréables à vivre.

Enfin, je tiens a remercier mes parents, qui m'ont donné des brillants exemples à suivre. J'espère seulement qu'ils me pardonneront d'avoir changé un peu de domaine...

Table des matières

1 Introduction 7
1.1 Besoins d'adaptabilité dans les systèmes d'exploitation . . . 7
1.2 Évaluation partielle . . . 8
1.3 Sujet de la thèse . . . 9
1.3.1 Génération de composants système spécialisés . . . 9
1.3.2 Composants système adaptatifs par spécialisation . . . 10
1.4 Plan . . . 10

i État de l'art 13

2 Optimisation de code système 15
2.1 Répercussions sur la structure des systèmes . . . 16
2.1.1 Systèmes configurables . . . 17
2.1.2 Systèmes extensibles . . . 17
2.1.3 Systèmes adaptatifs . . . 20
2.2 Conclusion . . . 23

3 Étude de cas : l'optimisation du RPC 25
3.1 Travaux antérieurs d'optimisation . . . 26
3.1.1 Optimisations dans le cadre traditionnel . . . 27
3.1.2 Nouvelles architectures de protocoles . . . 31
3.1.3 Approches langage . . . 32
3.2 Discussion . . . 34

4 L'évaluation partielle — une approche automatique à l'optimisation de programmes 37
4.1 Généralités sur l'évaluation partielle . . . 37
4.1.1 Un exemple concret . . . 37
4.1.2 Aspects extensionnels . . . 39
4.1.3 Stratégies d'évaluation partielle . . . 39
4.1.4 Évaluation partielle à l'exécution . . . 40
4.2 Tempo, un spécialiseur pour le langage C . . . 40
4.2.1 Adéquation aux applications système . . . 40
4.2.2 Spécialisation à l'exécution . . . 41

4.2.3 Interface . . . 42
4.3 Comparaison avec d'autres techniques d'optimisation . . . 42
4.4 Conclusion . . . 44

ii Spécialisation automatique de composants système 45

5 Spécialisation de l'implémentation RPC de Sun 49
5.1 Un exemple simple . . . 49
5.2 Opportunités de spécialisation dans le protocole RPC de Sun . . . 50
5.2.1 Sélection statique entre encodage et décodage . . . 51
5.2.2 Vérification statique du débordement des tampons . . . 52
5.2.3 Propagation des codes de retour . . . 53
5.2.4 Discussion . . . 54
5.3 Évaluation de performances . . . 54
5.4 Expérience acquise . . . 57
5.4.1 Utilisation de Tempo . . . 58
5.4.2 Optimisation d'un code existant . . . 58
5.4.3 Intégration du code spécialisé . . . 58
5.5 Conclusion . . . 59

6 Spécialisation des IPC du micro-noyau Chorus 61
6.1 Présentation de Chorus . . . 61
6.1.1 Les IPC de Chorus . . . 62
6.1.2 Architecture des IPC du micro-noyau Chorus . . . 62
6.2 Opportunités de spécialisation . . . 66
6.2.1 Choix du sous-cas à optimiser . . . 66
6.3 Spécialisation du code . . . 69
6.3.1 Spécialisation de code C++ avec Tempo . . . 69
6.3.2 Spécialisation à la compilation . . . 71
6.3.3 Spécialisation à l'exécution . . . 73
6.4 Intégration du code spécialisé . . . 73
6.5 Évaluation de performances . . . 75
6.6 Conclusion . . . 76

iii Composants adaptatifs par spécialisation 79

7 Une approche déclarative à la spécialisation 81
7.1 Définition d'une approche de haut niveau à la spécialisation . . . 82
7.1.1 Difficultés avec un langage tel que C . . . 83
7.1.2 Survol de notre approche . . . 83
7.1.3 Exemple . . . 84
7.1.4 Problématique de la spécialisation déclarative . . . 85
7.2 Classes de spécialisation . . . 86

7.2.1 Sémantique des classes de spécialisation 87
7.2.2 Construction d'un système de fichiers adaptatif par spécialisation . . 89
7.3 Compilation des classes de spécialisation 91
7.3.1 Extension de l'exemple 94
7.4 Problématique de la spécialisation à l'exécution 96
7.4.1 Techniques de *cache* pour les versions spécialisées 97
7.4.2 Quelques stratégies de cache 98
7.4.3 Surcoût du support à l'exécution 99
7.5 État courant 99
7.6 Extensions 101
7.6.1 Optimisations 101
7.6.2 Extensions de la puissance d'expression 102
7.7 Conclusion 104

8 Conclusion **105**
8.1 Contributions 105
8.2 Perspectives 106

Bibliographie **109**

A L'algorithme de compilation des classes de spécialisation **119**

Chapitre 1

Introduction

L'*adaptabilité* est en train de devenir une caractéristique incontournable des systèmes d'exploitation. Elle se doit de répondre à des besoins fondamentaux tels que :

- l'*évolution* et l'*hétérogénéité* des matériels informatiques, ainsi que la prise en compte de leurs caractéristiques de bas niveaux ;
- la *généralité* sans cesse croissante des services systèmes, qui doivent intégrer des caractéristiques nouvelles comme la qualité de service, la sécurité, etc., tout en maintenant un bon niveau d'*efficacité* ;
- le besoin d'*intégration* avec d'autres composants logiciels offrant des services *middleware* ou applicatifs, pour constituer des systèmes informatiques complets.

Malgré les besoins importants d'adaptabilité dans les systèmes d'exploitation, les techniques utilisées pour répondre à ces besoins sont généralement *ad hoc*.

Cette thèse étudie une approche systématique à l'adaptation de composants système, basée sur une technique automatique d'optimisation de programmes, à savoir l'*évaluation partielle*.

Nous présentons brièvement les besoins d'adaptation dans les systèmes d'exploitation d'une part, et l'évaluation partielle d'autre part, avant de détailler le sujet de cette thèse.

1.1 Besoins d'adaptabilité dans les systèmes d'exploitation

De manière traditionnelle, les systèmes d'exploitation ont suivi le modèle d'une machine virtuelle, en offrant aux applications un ensemble de services génériques, les mêmes pour toutes les applications. Pour beaucoup d'applications, une implémentation générique de ces services est parfaitement adaptée. Cependant, il existe des applications (ou des classes d'applications) avec des besoins spécifiques qui sont mal satisfaits par le cas général.

Ainsi, par exemple, les politiques génériques de gestion de fichiers sont souvent mal adaptées à des utilisations particulières comme les bases de données [108]. De même, les politiques génériques d'ordonnancement des tâches sont parfois mal adaptées aux transferts de contrôle entre des fils d'exécution concurrents communicant par appels de procédures à distance [110]. En fait, ce type de conflits a été identifié dans presque tous les sous-systèmes de gestion de ressources [8, 43, 54].

Le conflit entre généricité et performance démontre qu'un système d'exploitation efficace ne saurait ignorer les besoins spécifiques des applications qui utilisent ses services. Diverses formes de systèmes d'exploitation qui offrent la possibilité aux applications de *personnaliser* leurs services ont récemment vu le jour [15, 47, 122, 83, 22, 99, 103]. Dans les systèmes dits *extensibles*, les applications peuvent étendre l'éventail des services système avec des fonctionnalités nouvelles. Les systèmes dits *adaptatifs* peuvent modifier eux-mêmes les services offerts (en terme de stratégie, de politique, d'algorithme ou d'implémentation), en tenant compte des besoins réels constatés en cours d'utilisation. De nombreuses expériences ont démontré que l'incorporation de services optimisés pour les besoins des applications peut nettement améliorer leurs performances. Dans la pratique, pas seulement les systèmes extensibles, mais tous les systèmes d'exploitation incorporent des optimisations portant sur différents composants.

Une constante dans la plupart de ces optimisations est le caractère *ad hoc* de la démarche : le code générique est réécrit manuellement pour l'adapter à un contexte donné. Un défaut inhérent à cette approche est le risque d'introduire des erreurs dans les composants optimisés. Ces erreurs peuvent avoir deux conséquences négatives fondamentales : le changement dans la fonctionnalité du composant ou la corruption du noyau. Le dernier problème est connu sous le nom de problème de sûreté dans les systèmes extensibles [15]. Pour éliminer le risque d'introduction d'erreurs, il est impératif d'utiliser une approche automatique pour accomplir la tâche d'optimisation.

1.2 Évaluation partielle

Des formes d'évaluation partielle ont été depuis longtemps utilisées pour résoudre le conflit entre généricité et performance. L'évaluation partielle est une technique de spécialisation automatique de programmes. Elle a pour but d'optimiser un programme pour un contexte d'exécution particulier, pour lequel certains paramètres du programmes sont connus [62]. Le résultat de la spécialisation est un programme dans lequel tous les calculs dépendant des paramètres connus ont été effectués. Par conséquent, le programme spécialisé est souvent plus efficace que la version d'origine, selon l'importance relative des calculs éliminés.

La technique d'évaluation partielle a été étudiée initialement dans le contexte des langages de programmation fonctionnels. Depuis, des techniques et outils ont été développés pour une large variété de langages [70, 29, 11, 67] — logiques, impératifs, à objets —, avec des applications prometteuses [13, 52, 7]. Il existe aujourd'hui des outils qui traitent des langages d'utilisation industrielle, tels que le langage C [9, 6, 34]. Ces outils sont capables d'effectuer la spécialisation soit lors de la compilation (par rapport à des paramètres connus statiquement), soit pendant l'exécution même du programme [36].

1.3 Sujet de la thèse

La spécialisation automatique a atteint un degré de maturité qui semble indiquer qu'elle puisse résoudre le conflit entre généricité et performance sur des composants logiciels de taille réelle.

Cette thèse propose une approche automatique à l'adaptation de composants, reposant sur la spécialisation de programmes aussi bien lors de la conception de composants, que lors de leur exécution. Nous validerons cette approche dans le domaine des systèmes d'exploitation, en optimisant des composants système génériques existants. Notre approche comprend deux volets : la génération de composants système spécialisés à partir de composants existants et la génération de composants adaptatifs par spécialisation.

1.3.1 Génération de composants système spécialisés

Le premier volet de notre approche définit une méthode pour la conception des composants systèmes optimisés pour les besoins des applications. La méthode que nous proposons est d'implémenter une seule version générique d'un sous-système, et de dériver automatiquement des versions optimisées pour chaque application, par spécialisation de cette version générique à l'aide de l'évaluation partielle. Cette approche présente des avantages essentiels par rapport aux approches traditionnelles :

- La maintenabilité du système est préservée même en présence d'extensions multiples optimisant un même composant, car uniquement la version générique du composant doit être considérée pour toute mise à jour. En outre, du fait de la dérivation automatique des extensions on évite le risque d'introduire des erreurs ou des incohérences, inévitables lors d'une spécialisation manuelle.

- La spécialisation automatique offre également une solution au problème de sûreté des systèmes d'exploitation extensibles, car les extensions produites par spécialisation préservent la sémantique du module générique, qui est supposé fiable (*trusted*).

- L'application de l'évaluation partielle ouvre des possibilités d'optimisation difficiles à imaginer autrement:

  - Elle permet une spécialisation systématique, pour toutes les applications avec des fortes contraintes de performances, et — à l'intérieur d'une même application — une *spécialisation incrémentale* pour répondre à des cas critiques de plus en plus particuliers.
  - Elle permet de spécialiser du code à l'exécution et donc d'exploiter des opportunités d'optimisation dynamiques, qui ne sont pas aisément accessibles à une approche manuelle. La spécialisation à l'exécution permet ainsi de confectionner du “code jetable” — des versions de code extrêmement spécialisés et efficaces, utilisables pendant une période d'exécution (par exemple le temps d'une session entre un client et un serveur).

Pour spécialiser automatiquement des composants génériques, nous avons participé au développement d'un évaluateur partiel pour des programmes C, nommé Tempo, spécialement adapté aux applications système. Ainsi, notre spécialiseur a été conçu pour pouvoir traiter des composants de grande taille. De plus, Tempo incorpore plusieurs analyses nouvelles qui se sont avérées indispensables pour les applications système.

Nous démontrons l'adéquation de Tempo à des applications existantes à travers deux exemples complets de spécialisation sur des composants système commerciaux : le protocole d'appel de procédures à distance de Sun, et le module de communication inter-processus du micro-noyau CHORUS.

1.3.2 Composants système adaptatifs par spécialisation

Le second volet de notre approche vise l'intégration des différentes versions spécialisées dans un unique composant, *adaptatif par spécialisation*. Ce type particulier de comportement adaptatif assure une spécialisation du composant par rapport aux besoins instantanés en cours de l'utilisation. L'implémentation d'un comportement adaptatif par spécialisation concerne la gestion des versions spécialisées en cohérence avec les différents besoins des applications. Dans le cas de la spécialisation à l'exécution, cette forme d'adaptativité concerne aussi la production de versions spécialisées au vol.

Au lieu d'imposer au concepteur du système d'implémenter de manière explicite tous ces aspects opérationnels, nous introduisons une approche déclarative à la spécialisation de composants. Plus précisément, nous définissons un support langage pour exprimer d'une manière aisée le comportement adaptatif par spécialisation. À partir des déclarations de spécialisation associées à un composant générique, tous les aspects opérationnels impliqués dans la production et la gestion des versions spécialisées sont réalisés de manière automatique.

Notre approche est non-intrusive, dans le sens où la spécification de la spécialisation est séparée du composant lui-même, permettant de générer plusieurs instances différentes d'un même service système.

1.4 Plan

Le plan de la thèse est le suivant : le chapitre 2 analyse les besoins d'optimisation dans le code système, et montre les répercussions des services optimisés sur la structuration des systèmes. Le chapitre 3 réalise un inventaire des techniques d'optimisation traditionnelles sur l'exemple d'un service système représentatif : l'appel de procédures à distance (ou RPC). L'esprit de cette étude de cas est de séparer les techniques spécifiques au domaine considéré et les techniques plus généralement applicables, qui pourraient être factorisées par spécialisation automatique.

Le chapitre 4 introduit l'évaluation partielle et présente Tempo, notre outil de spécialisation de programmes, en insistant sur son adéquation aux applications système.

La deuxième partie de la thèse expose le premier volet de notre approche, consistant à dériver les extensions système spécialisées à partir de composants génériques existants. Le chapitre 5 présente l'application de cette méthode à la spécialisation de l'implémentation du protocole RPC de Sun. Le chapitre 6 considère un deuxième exemple de spécialisation, cette fois visant du code situé à l'intérieur du micro-noyau Chorus, et prenant en compte des opportunités de spécialisation éphémères apparaissant le temps d'une connexion client-serveur.

La troisième partie de la thèse expose le deuxième volet de notre approche, concernant la déclaration du comportement adaptatif par spécialisation. Nous montrons que les approches antérieures à la gestion de la spécialisation, reposant sur des annotations dans le programme source souffrent d'un manque de flexibilité et reflètent le manque de structure d'un langage tel que le C. Nous montrons en revanche que dans un langage structuré à base d'objets on peut intégrer de manière naturelle un support langage pour l'adaptation par spécialisation.

Nous concluons dans le chapitre 8 en esquissant les perspectives ouvertes par ce travail.

Première partie

État de l'art

Chapitre 2

Optimisation de code système

L'évolution historique des systèmes d'exploitation témoigne d'une complexification continuelle des fonctionnalités offertes aux applications. Les systèmes primitifs, offrant uniquement une abstraction des différents périphériques, ont été suivis progressivement par des systèmes multi-tâches, multi-processeur ou distribués. D'autres fonctionnalités nouvelles telles que le multimédia ou la prise en compte des environnements mobiles sont en train d'être inclues dans la plupart des systèmes actuels.

Ces nouvelles fonctionnalités ont contribué à rendre l'écriture des applications plus facile, en ramenant une bonne partie de la complexité des applications à l'intérieur du système. En même temps, le code de chaque service système a tendance à être très générique par rapport à l'utilisation courante, étant donné qu'il doit répondre à un nombre important de situations.

Examinons les principaux types de généricité rencontrés dans des programmes système :

Variété du matériel. Les services du système d'exploitation doivent prendre en compte les particularités d'un nombre croissant de plates-formes et de périphériques. Selon toute probabilité, cette diversification des matériels ne va que s'accentuer dans l'avenir.

Minimalité de l'interface. Pour des raisons de portabilité des applications et de facilité d'utilisation, les systèmes essaient généralement d'offrir une interface reposant sur un nombre minimal de concepts. Ces concepts constituent des abstractions sur un ensemble de services ou objets de bas niveau, et donc regroupent habituellement une famille de services. Par exemple, le concept de fichier dans Unix regroupe un très grand nombre de cas autour de l'abstraction d'un flot de caractères (*stream*). Les cas particuliers sont aussi divers qu'un périphérique, un fichier sur disque ou une connexion par réseau.

Besoin de flexibilité. Les abstractions fournies par un système permettent d'unifier la plupart des utilisations courantes des entités de bas niveau. Néanmoins, pour permettre à certaines applications de tirer parti, par exemple, d'un matériel particulier, l'interface du système offre souvent différentes *options*, permettant d'ajuster le service sous-jacent. Ainsi, dans le cas d'une connexion réseau, l'interface permet de passer aux protocoles de plus bas niveau des paramètres tels que le délai d'attente ou le nombre de retransmissions d'un paquet.

Architecture modulaires. Afin de maîtriser la complexité du code, les développeurs de systèmes utilisent systématiquement deux formes de modularité : verticale (par exemple un empilement de couches réseau) et horizontale (par exemple entre les différents sous-systèmes). Un des effets bénéfiques de cette architecture est de permettre de construire des modules interchangeables. En revanche, les interfaces entre ces modules et les modules adjacents doivent garder un niveau d'abstraction suffisant pour toutes les combinaisons permises, ce qui se traduit par un niveau supplémentaire de généricité.

L'implémentation des modules système intégrant un tel niveau de généralité apporte des avantages évidents en termes de génie logiciel, tels qu'une meilleure configurabilité ou maintenabilité. En revanche, l'efficacité est compromise, car dans le code, cette généralité se traduit par la traversée d'un nombre accru de couches, et par une multitude de vérifications. Toutes ces opérations ne constituent pas du travail effectivement utile pour le service, mais sont pourtant effectuées à chaque invocation.

On comprend donc facilement pourquoi l'optimisation a toujours été une préoccupation majeure des concepteurs de systèmes. Traditionnellement, on procède à l'optimisation d'un cas particulier d'utilisation, considéré comme *cas courant*. Il est évident que ce cas courant — quoique choisi pour satisfaire la plupart des applications —, peut s'avérer inadapté pour d'autres applications. Idéalement, il faudrait que chaque application puisse optimiser le service selon ses propres besoins.

L'adaptation d'un service à un cas particulier d'utilisation peut apporter un gain en performances parce que la généralité du service (ou tout du moins une partie) n'est plus nécessaire. Ainsi, l'utilisation courante d'une application peut se restreindre à une configuration matérielle donnée ; une famille de services peut être utilisée toujours pour demander un même sous-service, et éventuellement avec un certain nombre d'options fixées ; la configuration du service en termes de composition de modules peut être fixée elle aussi de manière statique.

Par exemple, l'installation d'une application de type base de données sur une machine particulière définit un contexte d'utilisation dans lequel : la configuration matérielle est connue ; les services d'entrée-sortie se restreignent souvent à des fichiers sur disque, probablement sous accès indexé et/ou séquentiel ; la configuration des couches (par exemple si on utilise le niveau NFS ou pas) peut être fixée elle aussi. La connaissance de tous ces paramètres peut amener à la définition d'un chemin critique optimisé, dans lequel une partie importante de la généricité pourra être éliminée.

2.1 Répercussions sur la structure des systèmes

L'existence de plusieurs versions d'un même service introduit une nouvelle problématique, qui doit être prise en compte dans la structuration des systèmes personnalisables. Les nouveaux problèmes concernent principalement :

La génération des versions optimisées. Le plus souvent, les services sont réécrits manuellement, soit à partir de zéro, soit en réutilisant des parties du code existant à travers l'héritage des langages à objets. D'autres systèmes utilisent des générateurs de code spécifiques à un domaine.

L'intégration des versions optimisées dans le système. Le système doit permettre l'extension du noyau, de manière statique ou dynamique, tout en préservant un certain niveau de "sûreté".

La gestion des versions optimisées. En présence des extensions, le système doit décider quand utiliser une version optimisée plutôt qu'une autre, en fonction du contexte d'appel. Certains systèmes utilisent le *dispatch* des langage à objets pour cette sélection ; d'autres systèmes s'adaptent à un changement de contexte en installant (voire en générant) une nouvelle version.

Nous analysons par la suite comment ces points ont été traités dans différents systèmes existants.

L'idée de personnaliser les services système pour les besoins spécifiques d'une application a été étudiée depuis longtemps déjà dans différents systèmes d'exploitation. Gopal *et al.* [50] classifie les systèmes personnalisables en :

– *systèmes configurables*, dans lesquels on offre à l'application des options pour choisir entre plusieurs stratégies préexistantes, de sorte que des modules système appropriés soient utilisés pour effectuer ce service ;

– *systèmes extensibles*, dans lesquels on peut intégrer au sein du système de nouveaux services, qui n'étaient pas prévus lors de sa construction ;

– *systèmes adaptatifs*, dans lesquels le comportement des services peut évoluer de manière autonome pendant l'exécution du système, en réponse aux changements dans les besoins des applications.

2.1.1 Systèmes configurables

Des formes de services système configurables par les applications existent depuis longtemps. Par exemple, dans Berkeley UNIX, l'appel de système `madvice()` permet d'ajuster le fonctionnement de la mémoire virtuelle, et l'appel de système `socket()` permet de choisir une pile de protocoles à utiliser pour les communications en réseau.

La limitation des systèmes configurables est que toutes les implémentations d'un service doivent être prévues à l'avance, de manière statique. Ceci restreint l'utilisation de ce principe à des classes d'applications reconnues comme critiques lors de la construction du système.

2.1.2 Systèmes extensibles

Les systèmes extensibles éliminent cette limitation en offrant la possibilité d'incorporer dans le système des services complètement nouveaux, définis par chaque application. Avec cette possibilité, il apparaît un problème de sûreté de fonctionnement : les extensions chargées par les applications ne doivent pas corrompre les structures de données du noyau. Les différents systèmes extensibles se distinguent principalement par les solutions qu'ils emploient pour assurer en même temps l'extensibilité, la sûreté, et un surcoût minimal pour l'utilisation des extensions.

Un principe de base utilisé pour permettre l'extensibilité est de séparer les abstractions offertes par le système de leurs politiques de gestion. Souvent, les mauvaises performances de certaines applications proviennent d'une discordance entre les politiques implementées par le système et celles que nécessite l'application. Une solution à ce problème, initialement explorée dans le cadre du système Hydra [72], est de permettre aux applications d'implémenter leur propres politiques de gestion, à l'aide de composants système personnalisés. En général, cette approche demande aux développeurs d'exposer des interfaces de bas niveau qui étaient auparavant cachées à l'intérieur du système.

Les systèmes d'exploitation à base de micro-noyau [2, 101, 109] encapsulent une partie de la fonctionnalité système (par exemple le sous-système de pagination de la mémoire) dans des serveurs de niveau application. La communication entre le noyau et ces serveurs est réalisée par des envois de messages. Les services système peuvent être modifiés en remplaçant les serveurs existants ou en fournissant de nouveaux serveurs. Selon la possibilité d'inclure des nouveaux serveurs en cours d'exécution, sans regénérer et relancer le système, l'extensibilité peut être classifiée comme statique ou dynamique. Avec l'approche micro-noyau, la personnalisation est effectuée à gros grain, au niveau d'un serveur entier.

La sûreté est assurée par le fait que les serveurs utilisateur tournent dans des espaces d'adressage différentes de l'espace noyau. En revanche, cela impose un surcoût qui peut s'avérer important : des appels système qui s'effectuaient uniquement par un appel de procédure et des accès à des données partagées dans l'espace noyau, maintenant impliquent des changements de contextes et des emballages/déballages de données, tous associés avec le passage de messages à travers une barrière de protection.

Ce surcoût en termes de communications a conduit à une restructuration des micro-noyaux de la deuxième génération, qui a consisté à ramener certains serveurs critiques de nouveau dans l'espace du noyau. Cette tendance réalise en fait un compromis entre la sûreté et la performance, car le noyau est rendu vulnérable aux interférences avec les serveurs résidant dans le noyau.

Pour pallier à ce désavantage de sûreté, Bryce et Muller [21] proposent de placer les services personnalisés dans des domaines de protection superviseur, situés à l'intérieur de l'espace d'adressage de l'application. Un domaine de protection (DP) définit un sous-ensemble de l'espace mémoire d'une application. Le code se trouvant à l'intérieur d'un DP ne peut pas être accédé directement par l'application, mais uniquement en effectuant un appel de procédure protégé. Il s'agit en fait d'un *trap* système qui permet à l'application d'exécuter en mode superviseur le code contenu dans le DP à partir d'un point d'entrée préalablement défini. Au cours de l'exécution dans un DP, étant en mode superviseur, l'application a accès aux fonctions du noyau. Au retour du *trap*, l'application retourne en mode utilisateur. Du point de vue de l'application, un DP constitue donc un module protégé accessible uniquement au travers de son interface procédurale.

L'avantage principal des DP est de permettre une extensibilité à grain plus fin, en termes de code, car ce mécanisme impose un surcoût beaucoup plus léger qu'une communication par messages. Une limitation des DP est que le grain de protection, en termes de données, est celui d'une page de mémoire virtuelle, donc moins fin.

D'autres travaux utilisent des techniques inspirées des langages de programmation pour assurer la protection des données et du code, entre le noyau et les extensions. *SPIN* [15] est un système extensible dynamiquement, et à un grain très fin, en maintenant des bonnes performances. Les extensions — appelées *spindles* — sont chargées dans l'espace d'adressage du noyau, mais ne peuvent pas accéder à la mémoire de manière incontrôlée et ne peuvent pas exécuter des instructions privilégiées, qui pourraient compromettre l'intégrité du système. Ces propriétés sont garanties en exprimant les extensions dans un langage modulaire fortement typé, à savoir Modula-3. Avec cette approche, *SPIN* offre un modèle de protection efficace à base de capacités (*capability*). Une capacité en *SPIN* est simplement une référence opaque, protégée, à une structure de données cachée par une interface.

Les extensions sont placées dans des domaines de protection logiques, qui sont des espaces de noms contenant des interfaces. Les domaines sont référencés à leur tour par des capacités, et il existe un ensemble d'opérations d'édition de liens entre les domaines. Toutes ces opérations sont effectuées par le compilateur de Modula-3, et le code objet résultant est certifié avec une signature criptée. La prise en compte des extensions rajoutées dynamiquement est assurée par un système de communication basé sur des événements. Les extensions et le noyau génèrent des événements qui sont distribués aux routines de traitement déclarées lorsque chaque extension a été installée.

Dans Aegis [47], le principe de la séparation entre abstractions de base et politiques de gestion est poussé jusqu'à son extrême. L'idée est de construire un noyau vraiment minimal (offrant seulement 5 appels système !), appelé un “exokernel”, qui n'implémente aucune politique, mais uniquement quelques abstractions de très bas niveau telles que : les pages de mémoire physique, le processeur, les interruptions, le TLB, etc. Par exemple, le système de fichiers ou les pilotes de périphériques sont considérés comme faisant partie du niveau application. L'exercice consistant à remonter les politiques dans les applications a un triple but en Aegis :

– intégrer des extensions spécifiques à une application particulière,

– intégrer des extensions spécifiques à une configuration matériel particulière,

– augmenter la performance des services système, en diminuant le nombre de changements de contexte.

Les extensions peuvent être localisées soit dans une bibliothèque de l'application, soit incorporées au noyau si elles ont besoin de privilèges superviseur. Dans ce dernier cas, la sûreté est assurée par la conjonction des deux techniques suivantes :

– une analyse statique est effectué directement sur le code binaire; toutes les références mémoires directes et tous les branchements directs sont vérifiés;

– pour les accès mémoire et branchements qui ne sont pas vérifiables statiquement, des tests sont introduits à l'exécution, selon une technique nommé *sandboxing* [118].

Alternativement, les extensions peuvent être exprimées dans un langage fortement typé, et dans ce cas les mêmes techniques que dans *SPIN* sont utilisées.

D'autres systèmes implémentent l'extensibilité en termes de réflexion. Apertos [122, 113] a été conçu pour étudier un environnement ouvert et mobile, et pour expérimenter une nouvelle architectures orientée objet. Apertos est constitué de deux catégories d'objets : des objets qui contiennent de l'information, et des méta-objets qui définissent la sémantique et le comportement des objets. Le système Apertos peut changer dynamiquement en permettant aux objets de s'associer ou se dissocier librement des méta-objets correspondants. L'extensibilité en Apertos est assurée par la possibilité de charger dynamiquement du code protégé dans le noyau. La protection est réalisée principalement par l'encapsulation associée aux langages à objets.

Scout [83] est un système extensible destiné aux applications orientées communications. Scout est organisé autour d'un concept de *chemin de données* (*data path*), qui abstrait l'acheminement des données entre deux points du système. Toutes les ressources du système sont allouées à des flots de données, plutôt qu'à des processus ou fils d'exécution, comme dans les systèmes traditionnels. Un système Scout particulier est configuré pour fournir uniquement la fonctionnalité demandée pour une classe d'applications donnée. Cette technique présente l'avantage d'offrir un système souple et efficace pour des plates-formes dédiées à des tâches spécifiques, telles qu'un serveur de fichiers ou un serveur Internet. Le désavantage de cette approche est le manque de flexibilité et d'extensibilité à l'exécution, qui limite l'utilité de Scout comme système d'exploitation universel.

2.1.3 Systèmes adaptatifs

Dans les systèmes configurables et extensibles, les applications qui souhaitent modifier le système doivent soit rajouter leur propre composants personnalisés, soit faire en sorte que ces composants soient installés d'avance. Les étapes de la personnalisation sont ainsi contrôlées par une entité (processus ou humain) extérieure au système lui-même. Un système adaptable est par conséquent passif.

Typiquement, l'ajout d'un composant personnalisé consiste à charger le code approprié à l'intérieur du noyau. Cette méthode permet une personnalisation très flexible, mais présente un important désavantage : le fait d'exprimer les besoins de l'application directement en termes de code demande une forte expertise de programmation système au développeurs d'applications.

On considère qu'une application ou un système est adaptatif si celui-ci peut modifier lui-même 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.

Conceptuellement, 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 impliqués dans 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 [23].

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 décisions 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.

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 mis en œuvre au dessus d'un système sous-jacent possédant la propriété d'extensibilité.

2.1.3.1 Systèmes adaptatifs existants

Dans le système adaptatif Synthesis [99], l'introspection prend en compte uniquement l'information accessible à travers l'interface système standard. Typiquement, la lecture d'un fichier pourra être spécialisée par rapport à tous les paramètres fixés lors de l'ouverture du fichier. De cette manière, même des applications qui n'ont pas été réécrites pour l'adaptation peuvent en profiter. En revanche, l'information est implicite, donc approximative, ce qui veut dire que certaines opportunité d'optimisation sont perdues. Par exemple, le style d'accès au fichier (séquentiel ou discontinu, par caractère ou par bloc, ...) ne pourra pas être prévu avec précision.

Les composants personnalisés en Synthesis sont produits par des générateurs dynamiques de code spécialement prévus dans le noyau, un pour chaque service adaptable. Ces générateurs utilisent des fragments de code pré-compilés, qu'ils instancient et recopient très efficacement, pour produire une instance du service optimisé pour le cas en question. Le nouveau code est intégré dans le système par une technique de modules composables à l'exécution (appelés *quajects*). La sûreté du noyau est basée sur la correction des générateurs de code. Par exemple, si un fichier était inaccessible, sa procédure de lecture ne serait jamais générée. De même, quand un fichier est fermé, sa procédure de lecture est invalidée, et l'espace mémoire occupé par le code est recyclé.

Depuis Synthesis, de nombreux systèmes d'exploitation ont commencé à intégrer différentes formes d'adaptation par génération dynamique de code. Pour la plupart d'entre eux, l'adaptation se restreint à un service spécifique et utilise des optimisations spécifique à son domaine. Un exemple typique est la génération dynamique de filtres de paquets à partir d'une spécification fournie par une application, réalisé par Engler et Kaashoek [46]. Les mesures montrent une accélération d'un ordre de grandeur par rapport à une implémentation classique des filtres de paquets, interprétés à la volée. Un autre exemple de service adaptatif est le service d'appel de procédures à distance développé dans Mach par Ford *et al.* [48], que nous détaillerons dans la section 3.1.3.

Choices [22] est un système d'exploitation adaptatif orienté objet. Son architecture est basée sur des structures d'objets (*object frameworks*), organisées de manière hiérarchique selon leur fonction et leur applicabilité. Il existe des structures d'objets pour chaque sous-système, tels que la mémoire virtuelle, le système de fichiers, la communication par messages, la mémoire virtuellement distribuée, etc. Le système peut être personnalisé pour une application ou une plate-forme particulière, en substituant les objets d'une (sous-)structure. Au dessus de ce support pour l'extensibilité, Choices implémente des services adaptatifs tels qu'un système de fichiers auto-réglable en fonction de son contexte d'exécution [77].

μChoices est une re-implémentation de Choices à base d'un micro-noyau, qui emprunte beaucoup de principes de conception de son prédécesseur, notamment en ce qui concerne les structures d'objets permettant une personnalisation incrémentale. μChoices permet aussi une extensibilité dynamique par le chargement des agents interprétés [74]. Les agents sont écrits dans un langage similaire à Tcl ou en Java, et servent typiquement à construire des agrégats de plusieurs appels système, afin de minimiser le nombre de changements de contextes entre l'espace utilisateur et système. Un autre avantage de leur interprétation est de pouvoir contrôler leur consommation de ressources — par exemple on peut arrêter un agent qui dépasse son quota d'utilisation du CPU.

Dans le système VINO [103], l'étape d'introspection consiste à récolter à la fois des données statiques (telles que la trace d'une exécution passée représentative) et des données dynamiques, accumulées au cours de l'exécution courante. L'étape de décision inclue par conséquent deux types d'analyses. L'analyse des données statiques procède par une "simulation *in situ*", c'est-à-dire la réexécution d'un appel système avec des différentes politiques, et aboutit à des heuristiques pour le choix de la meilleure politique pour un cas donné. L'analyse des données dynamiques vise à détecter les cas d'anomalie au cours de l'exécution, afin de déclencher un changement de politique. Enfin, l'étape d'action consiste à remplacer des méthodes des objets système ou des gestionnaires d'événements internes au système. Ce remplacement est effectué de manière indépendante au niveau de chaque processus.

2.1.3.2 Adaptativité par spécialisation

Les systèmes extensibles ou adaptatifs fournissent un support pour la personnalisation des services. Différentes solutions existent pour assurer la sûreté du noyau et un surcoût minimal de l'extensibilité. Cependant, l'écriture des extensions (ou des générateurs d'extensions, pour Synthesis) reste manuelle, et nécessite une importante expertise soit au développeur d'application, soit au développeur système. De plus, si les extensions sont des versions différemment optimisées d'un service général, il se pose un grand problème de maintenabilité : lors de la modification d'un service, il faut assurer que toutes les versions restent correctes, et en cohérence avec la nouvelle sémantique du service. Ce problème est très délicat surtout parce que la sémantique d'un service système est difficile à définir formellement.

Le projet Synthetix [98] vise à proposer une alternative à cette approche manuelle à la construction des systèmes adaptatifs, afin de dépasser ses limitations. La démarche consiste à dériver les composants système personnalisés par spécialisation automatique de composants génériques, à l'aide de l'évaluation partielle. Les composants sont spécialisés pour des cas critiques en termes de performance. Ces cas sont définis par des propriétés sur l'état du système (ou d'un composant), appelées des *invariants*. On distingue entre des vrais invariants, qui sont des propriétés valables pendant toute l'exécution du système, et des *quasi-invariants*, qui deviennent valables à un certain moment pendant l'exécution mais peuvent être invalidées ultérieurement.

Pour implémenter l'étape d'introspection, le système est instrumenté par l'insertion de fragments de code nommées *gardes*, qui détectent tout changement d'un quasi-invariant. En cas d'invalidation d'un quasi-invariant, l'étape d'analyse choisit des composants plus génériques pour remplacer les composants concernés par ce quasi-invariant. De manière duale, lorsqu'un quasi-invariant est validé, des composants plus spécialisés sont choisis, assurant une forme de *spécialisation incrémentale*. L'étape d'action consiste à remplacer l'ancien composant avec le composant choisi, en tenant compte des contraintes dûes à la concurrence du système [40].

Une partie des travaux présentés dans cette thèse a été réalisée en collaboration avec le projet Synthetix. Nous décrirons dans les chapitres 5 et 6, à travers deux exemples complets, comment des composants système optimisés peuvent être obtenus de façon automatique à l'aide de l'évaluation partielle. Dans le chapitre 7, nous introduisons une approche déclarative à la construction des composants adaptatifs, à travers une extension langage pour la spécification des invariants et quasi-invariants. Cette approche rend automatique l'instrumentation du système avec le code nécessaire à l'auto-adaptation — et qui concerne l'introspection, la prise de décision et l'action.

2.2 Conclusion

Nous avons analysé dans ce chapitre les raisons qui font que la généricité est une caractéristique intrinsèque des composants système. Cette généricité apporte certains avantages en termes de génie logiciel, mais pénalise les performances de certaines applications. Ce conflit pousse les concepteurs de systèmes à introduire de nouveaux services système optimisés pour des besoins spécifiques. L'existence de plusieurs versions optimisées introduit trois nouveaux problèmes, concernant respectivement : la génération des versions optimisées, l'intégration de ces versions dans le système et la gestion de ces versions.

Pour les deux derniers problèmes, on peut constater que d'une part, l'intégration des versions optimisées a été résolue de manière satisfaisante dans le cadre des systèmes extensibles. D'autre part, la gestion des versions optimisées a été traitée dans le cadre des systèmes adaptatifs existants, mais cette solution demande une expertise système très importante.

En revanche, il n'existe actuellement pas de cadre général permettant la production des versions optimisées. Les générateurs de code à la Synthesis ont certainement exhibé des opportunités d'optimisation intéressantes, mais cette solution demande une expertise système et une intervention manuelle très importante. Le projet Synthetix vise à contourner cette difficulté en utilisant la spécialisation automatique comme facteur d'optimisation. Les solutions que nous développerons dans la suite de ce document s'inscrivent dans cette idée initiée par le projet Synthetix.

Avant de présenter ces solutions, quelques questions se posent naturellement à ce point dans notre réflexion. Si les services système sont si génériques, est-ce que la spécialisation (même manuelle) est déjà utilisée dans beaucoup de systèmes ? Est-ce que la spécialisation traite toutes les opportunités d'optimisation existant dans les services génériques ? Si ce n'est pas le cas, quelle est la proportion d'optimisations que traite la spécialisation, en comparaison des autres techniques d'optimisation utilisées ?

Le chapitre suivant traite ces questions en détail.

Chapitre 3

Étude de cas : l'optimisation du RPC

Ce chapitre présente l'éventail des techniques manuelles ou semi-automatiques employées habituellement pour optimiser des services système. Pour cette étude, nous considérons un exemple typique de service générique : l'appel de procédure à distance. Nous montrons pourquoi ce service est important à optimiser et les raisons principales de sa généricité. Nous montrons ensuite que différentes formes de spécialisation ont déjà été utilisées dans de nombreux travaux, *en complémentarité* avec d'autres techniques plus spécifiques au domaine considéré.

L'appel de procédure à distance (ou RPC [16], pour *remote procedure call*) est un protocole de communication communément utilisé dans les systèmes distribués. Ce protocole rend l'interaction avec une machine distante très ressemblante à un appel de procédure local. Dans un RPC, un processus appelant demande l'exécution d'un service à un autre processus, qui peut se trouver sur une autre machine. Le processus appelant est bloqué jusqu'à l'exécution complète du service, et reprend ensuite le calcul avec la valeur de retour récupérée. Le programmeur d'applications distribuées n'a pas à se soucier des détails de la communication entre les processus, ni de la représentation des données transmises et réceptionnées. Le RPC offre un moyen de communication transparent à la localisation des processus : la même primitive est appelée dans le cas local (processus appelant et processus appelé sur le même site) ou distant (processus sur de sites différents). Ceci permet de changer la topologie d'une application distribuée sans avoir pour autant à la modifier ou la recompiler. Le mécanisme du RPC est à la base des architectures client-serveur, et des services aussi répandus que le système de fichiers distribué NFS de Sun.

Bien que le mécanisme du RPC soit utilisé très largement, et qu'il offre beaucoup d'avantages, ses implémentations sont souvent inefficaces. Même l'apparition des réseaux hauts-débits, comme l'ATM ou FDDI, n'a pas changé cette situation de manière significative. Dans les cas où les contraintes d'efficacité sont particulièrement fortes, le surcoût induit par les RPC pousse les concepteurs d'applications réparties à concevoir leur propres protocoles de communications pour des usages ou pour des plate-formes spécifiques [14, 53, 61], ce qui complique considérablement le modèle de programmation, et augmente par conséquence le coût de développement et l'effort de maintenance.

L'inefficacité de beaucoup d'implémentations du mécanisme du RPC provient, en partie, de son important degré de généricité. En effet, toutes les causes potentielles de généricité d'un service système, énumérées dans le chapitre précédent (voir page 15), y trouvent une instanciation :

- **Variété du matériel.** Typiquement, le protocole est conçu pour pouvoir être utilisé sur une grande variété de matériels, ce qui fait que les hypothèses du cas le plus défavorable sont considérées. Par exemple, le média de communication est typiquement considéré non-fiable (perte ou duplication possible des messages). Si on utilise ce protocole générique sur un réseau fiable, une grande partie du code ne sera jamais utilisée (gestion des erreurs, retransmissions, ...). De même, le protocole intègre une couche pour la gestion des adresses Internet, pour le cas d'un réseaux géographique. Quand le protocole est utilisé sur un réseau local, cette couche devient superflue.

  Enfin, le protocole RPC doit prendre en compte le cas où les machines eméttrice et réceptrice utilisent des modèles mémoire différents (*big endian*, *little endian*), et donc recourt typiquement à une conversion des arguments dans un format indépendant de l'architecture. Dans le cas d'un réseau homogène, cette conversion devient inutile.

- **Minimalité de l'interface.** Comme tout service système, le RPC rassemble en fait une famille de services assez différents. Tout d'abord, le cas local et le cas distant impliquent des mécanismes de transfert des arguments distincts : copie de mémoire pour le cas local, ou envoi sur le réseau pour le cas distant. Par ailleurs, certaines systèmes incluent dans le service RPC des possibilités de broadcast ou de multicast (envoi d'une requête à un groupe de processus).

- **Flexibilité.** Le service RPC offre typiquement des options pour choisir : le protocole sous-jacent (datagramme ou par connexion), le comportement en cas de révocation (interruptible ou non), le délais d'attente (fini ou infini), etc. Selon la valeur de ces options, certaines fonctionnalités seront utilisées et d'autres ignorées. De fait, le chemin d'exécution critique sera défini, en partie, par ces options.

- **Composition des modules.** Comme tout protocole, le RPC est normalement implémenté par un empilement de couches logicielles. Pour assurer la flexibilité impliquée par le jeu d'options, certaines couches doivent pouvoir être remplacées par d'autres, sans changer le reste de l'implémentation. Par exemple, dans certaines implémentations la couche de niveau transport peut être soit UDP (de type datagramme) soit TCP (de type connexion), au dessus d'une même couche IP. Il en résulte que la couche IP ne peut faire aucune hypothèse sur la couche supérieure ; elle doit donc intégrer un certain degré de flexibilité, qui implique un niveau supplémentaire de généricité.

Du point de vue de notre démarche les RPC représentent donc un bon exemple de service système générique.

3.1 Travaux antérieurs d'optimisation

Il existe de nombreux travaux, très variés, sur l'optimisation des RPC dans les systèmes d'exploitation. Nous ne nous proposons pas de donner une liste exhaustive de tous les travaux existants, mais nous allons classer les travaux les plus représentatifs, en prenant en compte les critères suivants.

Mesure de performance à optimiser. Une première raison de la multiplicité des optimisations réside dans la diversité des mesures de performance visées par les optimisations :

– La *latence* est une mesure du surcoût introduit par le protocole. Elle est égale au temps aller-retour d'un RPC auquel on soustrait à la fois le temps passé à exécuter le service et le temps des communications (pour l'émission des arguments et la réception du résultat). La latence est plus particulièrement importante pour les transactions de petite taille. C'est un cas particulier assez courant, car plusieurs études statistiques [53, 14] montrent que la plupart des RPC invoqués dans des systèmes de type micro-noyau véhiculent des messages de petite taille.

– Le *débit* définit la vitesse de transfert des données, à travers le réseau, mais aussi à travers les couches de protocoles (qui peuvent impliquer notamment des copies de mémoire). Le débit est particulièrement important pour des messages de grande tailles. C'est le cas de tout serveur avec des arguments ou des résultats complexes. Pour citer un exemple, une mémoire virtuelle répartie [73, 123] repose de manière critique sur des RPC à haut débit.

Sous-cas à optimiser. La deuxième raison de la multiplicité des optimisations réside dans la diversité des sous-cas possibles à optimiser. C'est pour le cas considéré comme le plus fréquent que l'on crée un chemin optimisé. Certaines études considèrent le cas du RPC local comme la situation la plus commune (lorsque le processus appelant et le processus appelé se trouvent sur le même site), d'autres considèrent le cas du RPC distant (communication entre deux sites).

Techniques d'optimisation utilisées. Enfin, une troisième raison de diversité est la multitude d'angles d'attaque possibles. Dans la section 3.1.1 nous décrivons les travaux qui conservent l'architecture générale d'une implémentation traditionnelle mais visent à optimiser des aspects spécifiques perçus comme des goulots d'étranglement, tels que la copie des messages ou la conversion des arguments. Dans la section 3.1.2, nous décrivons les travaux qui recourent à une restructuration du protocole selon une architecture différente, notamment pour diminuer la latence de l'architecture classique, organisée en couches. Finalement, dans la section 3.1.3 nous avons regroupé les travaux utilisant des techniques provenant des langages de programmation, typiquement effectuant des optimisations spécifiques au domaine.

3.1.1 Optimisations dans le cadre traditionnel

De manière traditionnelle, les RPC sont mis en œuvre sous la forme d'une pile de protocoles, au dessus d'une interface réseau. Nous présentons dans cette section des travaux d'optimisation qui conservent ce cadre général.

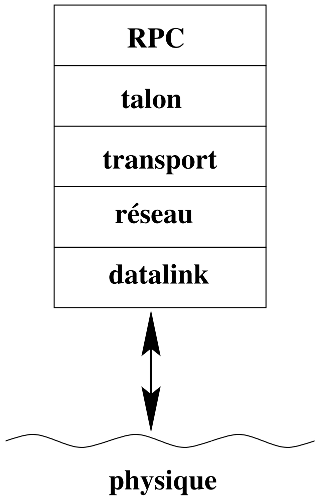


Fig. 3.1 – *Implémentation classique, avec les couches ISO*

3.1.1.1 Optimisations des protocoles en couches

La plupart des optimisations traditionnelles améliorent certaines couches de protocoles existantes, réimplémentent certaines couches, ou créent un chemin optimisé pour le cas le plus courant. Comme la pile de protocoles est complexe, les optimisations visent des couches différentes. Souvent, cette diversité dans la cible des optimisations vient de perceptions contradictoires de la source des goulots d'étranglement.

Dans une implémentation standard (non-optimisée), les RPC sont mis en œuvre par un empilement de couches comme celui de la figure 3.1. Le fonctionnement est le suivant : Le talon client alloue un message, emballe les arguments (tout en les convertissant dans un format indépendant de la machine), envoie le message et bloque, dans l'attente d'une réponse. Le message est traité successivement par toutes les couches. Chaque couche considère les données du niveau supérieur comme opaques, et rajoute son propre en-tête. La couche la plus basse envoie le message sur le médium physique, par l'intermédiaire du pilote de périphérique réseau. Quand le message arrive au site de destination, une interruption réseau est déclenchée par le matériel pour signaler l'arrivée d'un nouveau paquet. En général, l'interruption place le message dans une file d'attente, et réveille le fil de contrôle qui correspond à la couche la plus basse du protocole. Après avoir parcouru ainsi toutes les couches (ce qui a l'effet de démultiplexer le message), on réveille le processus serveur correspondant. Le talon serveur déballe les arguments et les convertit dans le format local ; puis, il invoque la procédure identifiée dans l'en-tête. Le retour est similaire à l'envoi et finalement le système réveille le processus client, qui reprend la procédure-talon bloquée en attente. Le talon dé-alloue le message et retourne le contrôle à l'application.

Schroeder et Burrows présentent une implémentation optimisée[1] du RPC pour le multiprocesseur Firefly[102]. Ils considèrent comme cas courant le RPC distant, dans l'hypothèse où il existe un fil de contrôle serveur en attente, et il n'y a pas de conflit sur les verrous du

1. Tous les travaux présentés dans cette section optimisent l'implémentation du RPC par rapport au critère de latence.

noyau. Ils utilisent plusieurs techniques d'optimisation, dont :

– la compilation des fonctions d'emballage à partir de la description des arguments ;

– le démultiplexage du paquet effectuée directement par l'interruption Ethernet, évitant ainsi plusieurs changements de contextes ;

– l'optimisation de la gestion des tampons systèmes pour les messages ;

– la réécriture en assembleur de l'interruption Ethernet.

Globalement, Schroeder et Burrows obtiennent un facteur d'accélération de 3 sur le RPC distant, par rapport à l'implémentation d'origine.

Bershad *et al.* [14] optimisent la même implémentation du RPC sur Firefly, mais pour le cas local. Ils justifient l'importance d'optimiser ce cas par la récolte des statistiques d'appels RPC système dans un micro-noyau. Il en résulte que la plupart des appels RPC sont à la fois locaux et véhiculent des arguments de petite taille. Le protocole optimisé (appelé LRPC) conserve globalement la sémantique d'exécution et de protection des RPC, mais repose sur un mécanisme d'appel allégé. Le transfert de contrôle entre le client et le serveur contourne l'ordonnanceur général : le fil de contrôle client passe dans le contexte du serveur et continue son exécution directement. Les arguments sont passés sur une pile partagée dans une zone de mémoire commune [2].

Il est intéressant de noter que la sélection entre le cas optimisé des LRPC et le cas général repose sur une phase de liaison (*binding*) entre le client et le serveur, pendant laquelle un descripteur est initialisé. Ce descripteur contient un drapeau qui "pré-compile" le choix entre les deux cas. En même temps, il contient un certificat d'authentification qui permet d'éviter des vérifications à chaque appel.

Le transfert de contrôle des LRPC a été reformulé par la suite, par Draves *et al.* [44], en termes de *continuations*. Le but principal de cette étude est d'obtenir une expression plus uniforme de plusieurs optimisations *ad hoc*. Il est à noter qu'une légère amélioration des performances est également obtenue.

Johnson et Zwaenepoel [61] présentent une autre implémentation optimisée du RPC, appelée Peregrine. Leur contribution principale est d'avoir éliminé de manière systématique les copies des arguments, par plusieurs techniques combinées. Les améliorations concernent à la fois le cas distant et le cas local. Comme dans le LRPC, on retrouve dans Peregrine un descripteur créé lors de la liaison client-serveur, qui évite des authentifications répétées, et qui contient également un drapeau, indiquant si la conversion des arguments est nécessaire ou non. De toute façon, on exécute au plus une conversion, toujours chez le client, car les données sont toujours envoyées dans le format du serveur.

L'élimination éventuelle de la conversion rend possible la suppression de toute copie des arguments lors de l'envoi : les arguments sont doublement mappés dans l'espace utilisateur et noyau, ce qui permet à l'interface réseau de les copier directement dans le tampon d'émission, avec un matériel spécialisé (*scatter-gather* DMA). De même, à la réception sur le site serveur, il n'y a pas de copie (sauf par DMA), car les arguments sont déballés directement dans le

2. Cette optimisation est basée sur une particularité du langage Modula2+, dans lequel ce protocole est écrit.

format de la pile du processeur, et remappés dans le segment de pile du serveur. Certains arguments peuvent ne pas être copiés du tout, si ils sont annotés comme uniquement d'entrée ou uniquement de sortie. En fait, la seule copie qui est toujours exécutée est celle lors de la réception du paquet de retour sur le site client, et cela pour ne pas modifier l'interface des RPC.

Par opposition à d'autres travaux, Tekkath et Levy [110] essaient de trouver des optimisations des RPC adaptées à plusieurs types de systèmes, basés sur des réseaux aussi divers que l'Ethernet, l'ATM ou le FDDI. Globalement, ils constatent que les réseaux plus rapides ont augmenté le débit des RPC, mais n'ont pas diminué leur latence (parfois même au contraire !). Ils réimplémentent le protocole RPC directement au dessus de l'interface réseau (sans utiliser UDP/IP). Le processus client est gardé en attente active pour un certain temps, en espérant économiser un changement de contexte, si la réponse du serveur arrive suffisamment vite. La vérification de la cohérence du contenu (*checksum*) est effectuée uniquement si le matériel ne la fait pas lui-même.

Le cas courant considéré dans cette étude est le RPC distant, avec des arguments de petite taille[3]. De même que dans Peregrine, ils éliminent la conversion des arguments si possible, et évitent la copie de certains arguments, par des annotations *in/out*. Cependant, Tekkath et Levy montrent que les transferts des arguments par DMA ou par double mappage ne sont pas toujours sûrs, lorsqu'il y a des exigences de protection. La solution alternative qu'ils proposent pour supprimer des copies est de générer dynamiquement du code pour l'emballage des arguments, qui sera exécuté dans le noyau. De cette manière, la procédure personnalisée du noyau saura chercher les arguments directement dans le format de l'application ; la copie des arguments dans un tampon contigu n'est plus nécessaire. La génération du code est en partie manuelle, car elle repose sur un squelette (*template*) fourni par l'utilisateur, qui est instancié lors de la liaison client-serveur.

La leçon principale retirée par Tekkath et Levy est que chaque optimisation des RPC possède son domaine d'application et ses limites. Ils plaident alors pour un choix flexible et adaptatif parmi plusieurs versions optimisées, selon la taille du paquet, l'interface réseau, le niveau de sécurité requis et la fiabilité des transmissions.

3.1.1.2 Optimisation du support système

L'efficacité globale d'une implémentation RPC[4] repose d'un côté sur des protocoles optimisés, et d'un autre côté sur un support efficace de la part du système d'exploitation, pour gérer les tampons, les réveils (*timers*), les changements de contexte, les copies, etc. Certains travaux visent à optimiser justement ces primitives de support.

Clark *et al.* [27] considèrent que les protocoles en couches (en l'occurrence, TCP/IP) sont raisonnablement efficaces en termes de débit, même pour des réseaux rapides, tels que le FDDI. Selon eux, la lenteur constatée globalement ne vient pas du protocole lui-même, mais de la généralité des services système sous-jacents. Pour le prouver, ils écrivent leur

3. La condition est que la taille du message résultant soit inférieure à la taille du plus grand message accepté par le médium.

4. Certaines études parlent plus généralement de communications inter-processus (IPC) — dont les RPC sont le cas particulier le plus cité.

propre support système spécialisé, pour la gestion des tampons et des réveils. Même si leur critère à optimiser est le débit, ce principe d'un support spécialisé est à retenir aussi pour des optimisations de latence.

Druschel et Peterson [45] s'intéressent spécialement au support système pour les copies de données. Ils proposent un mécanisme efficace pour les remplacer (les *fbufs*), et montrent qu'en utilisant ce mécanisme comme protocole de transport local inter-domaine, on peut obtenir des RPC locaux efficaces, à la fois en termes de latence et de débit.

Plusieurs travaux proposent d'utiliser en remplacement des RPC locaux, l'appel de procédure protégé, en plaçant le client et le serveur dans un même espace d'adresses. L'intérêt de cette approche est à la fois d'économiser deux changements de contexte, et de simplifier le passage des paramètres. Bryce et Muller [21] implémentent l'appel protégé au sein de Mach avec des *domaines de protection*. Yarvin *et al.* [121] utilisent l'appel protégé pour implémenter des RPC locaux très efficaces sur un système à base de processeur Intel 486.

Hsieh *et al.* [58] proposent plusieurs solutions pour améliorer le support système des RPC. Ils montrent l'importance de l'optimisation notamment du changement de contexte, du transfert de contrôle, et de l'existence d'un support matériel pour les procédures d'authentification.

Liedtke [75] va encore plus loin : il ne se limite pas à la réécriture des morceaux du support système associé aux RPC. Il se propose de restructurer complètement le système d'exploitation, avec le but explicite d'obtenir des RPC locaux performants (en termes de latence). Il adopte une approche verticale, intégrant des solutions à tous les niveaux : structure du système (notamment l'organisation des espaces d'adressage), interface des RPC, politiques de gestion (nouvelles stratégies d'ordonnancement), choix d'implémentation (utilisation de registres). Il démontre qu'il existe un effet de synergie entre des optimisations génériques et celles dépendantes de la machine, et il obtient des performances très convaincantes sur les aller-retours RPC locaux. Dans une étude postérieure [76], il présente une autre amélioration du cas courant des RPC, en réduisant de façon drastique le temps d'un changement de contexte, sur des systèmes à base d'Intel 486.

3.1.2 Nouvelles architectures de protocoles

Dans un article prospectif, Clark et Tennenhouse [28], anticipent les limitations (en termes de latence) de l'architecture traditionnelle, en couches. Ils introduisent une technique dite d'intégration des couches (*Integrated Layer Processing*, ou ILP), — qui consiste à fusionner les traitements de plusieurs couches adjacentes, afin d'éviter des lectures/écritures mémoire répétées —, et énoncent le principe de la séparation *in-band*/*out-of-band* — qui consiste à enlever du chemin critique toute opération qui pourrait être effectuée en post-traitement. Par ailleurs, ils identifient la couche présentation comme goulot d'étranglement, et proposent la technique de Application Level Framing (ALF) pour enlever cette couche du chemin critique[5].

5. La technique d'ALF consiste à garder, même dans les messages de plus bas niveau, une structure correspondant à l'usage de l'application (par opposition à une structuration opaque, en paquets Ethernet par exemple). Cela permet à l'application de poursuivre parfois son traitement en dépit de la perte ou de la permutation de paquets.

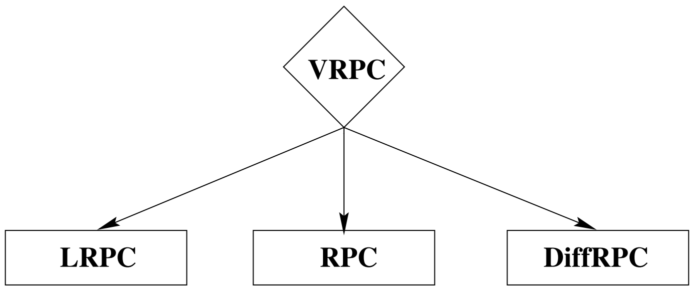


FIG. 3.2 – *Le protocole virtuel VRPC choisit le protocole sous-jacent selon la destination du message (serveur local, serveur distant, ou groupe de serveurs)*

Les “accélérateurs de protocoles” de van Rennese [114] constituent une application exemplaire des principes du ILP et de la séparation *in-band*/*out-of-band*. Il montre que pour diminuer la latence il faut optimiser le traitement *in-band* ; pour augmenter le débit il devient important d'optimiser aussi le traitement *out-of-band*. Les gains de performance obtenus par l'application systématique de ces principes sont considérables (un ordre de grandeur, le protocole étant mis en œuvre dans un langage fonctionnel).

Le système x-Kernel [91] est un système d'exploitation conçu pour simplifier l'implémentation de protocoles. Pour faciliter la conception et la réutilisation, les protocoles sont décomposés en des composants relativement simples, les *micro-protocoles*. Les protocoles sont obtenus par composition de micro-protocoles (tels un fragmenteur, un multiplexeur, etc) dans un *graphe de protocoles*. Pour assurer la composabilité, tous les protocoles respectent une interface générique, reposant sur des en-têtes uniformes. Pour éliminer la prolifération des options dans les protocoles, qui rendent les implémentations classiques aussi complexes, le système x-Kernel utilise les *protocoles virtuels*. Un protocole virtuel n'a pas d'état propre, mais réalise uniquement la sélection entre plusieurs protocoles de niveau inférieur, selon un critère dynamique. Par exemple, dans la figure 3.2 le protocole virtuel VRPC dirige les messages vers le protocole RPC optimisé adapté à sa destination.

Hutchinson *et al.* [59] présentent une expérience de décomposition en micro-protocoles d'un protocole RPC existant (Sprite RPC). Ils montrent que le surcoût d'un nombre accru de couches est bien compensé par l'accélération due aux protocoles virtuels, et à des optimisations spécifiques à cette architecture. Ils exposent les avantages du cadre x-Kernel en termes de génie logiciel (conception, maintenance, réutilisation).

3.1.3 Approches langage

Nous décrivons dans cette section des systèmes de conception de protocoles utilisant des méthodes de langages de programmation. La tendance extrême est d'écrire les protocoles dans un langage de spécification formelle (comme Estelle ou Lotos [18]), et de générer automatiquement un code exécutable. Nous ne présentons pas ces travaux, car ils présupposent une réimplémentation complète des couches réseaux, tandis que nous nous intéressons à l'optimisation d'une implémentation existante. Par ailleurs, la motivation de ce genre de travaux relève surtout d'autres critères que la performance.

Nous présentons dans cette section des systèmes dans lesquels on repose sur des méthodes langages uniquement pour décrire des aspects ponctuels (l'interface, le format des données, etc.). Ces travaux sont basés sur des compilateurs qui intègrent des optimisations spécifiques aux aspects correspondants. Souvent, ces compilateurs font appel à la génération dynamique de code [64], pour étendre la portée des optimisations.

Le système Morpheus [1] offre une encapsulation langage au support système de x-Kernel(qui a été décrit dans la section précédente). Avec Morpheus, la spécification des protocoles se fait dans un langage de programmation universel, enrichi de quelques abstractions permettant la construction des protocoles. Le langage est un dialecte de C++ avec quelques conventions et restrictions supplémentaires. Les abstractions offertes sont des squelettes pré-définis de protocoles (*shapes*) : multiplexeur, routeur, filtre. Tous les protocoles doivent spécialiser un de ces squelettes. Le compilateur profite pleinement de la connaissance du domaine, pour faire des optimisations agressives. Par exemple, afin de réduire le surcoût des couches, il réserve un registre pour pointer sur l'en-tête, et "court-circuite" des couches par une technique à base de continuations.

D'autres optimisations réalisées par Morpheus sont particulièrement intéressantes pour notre étude, car elles constituent un cas particulier de spécialisation dynamique de code. Plus précisément, l'utilisateur peut rajouter des annotations spécifiant des champs invariants dans les objets sessions. Guidé par ces annotations, le compilateur génère un modèle (*template*) de la routine d'envoi `send()`, qu'il instancie à chaque création d'un objet session. Toutefois, cette spécialisation est appliquée à une échelle restreinte : on ne crée qu'un seul *template* par routine instanciée. Par ailleurs, la propagation des constantes correspondants aux champs invariants est partiellement guidée par des annotations manuelles.

Mosberger *et al.* [85, 84] étudient des optimisations de latence selon une mesure particulière : le nombre de cycles d'attente mémoire par instruction du processeur. Tous leurs efforts sont ciblés sur l'amélioration du comportement de cache du chemin critique. Ils utilisent trois techniques langage pour essentiellement ré-aligner les morceaux de code dans le cache d'instructions, à l'aide d'un optimiseur automatique :

1. *outlining :* séparation du chemin critique par rapport au reste du code, et alignement dans une séquence contiguë d'instructions ; le chemin critique correspond juste à l'envoi de petits messages, sans le traitement des exceptions, et sans le code d'initialisation ;

2. *cloning :* duplication du chemin critique, avec un ré-alignement censé éviter les défauts de cache, couplée avec une forme très restreinte de spécialisation dynamique ;

3. *inlining :* dépliage des appels de fonctions, appliquée de manière sélective, avec une heuristique simple et efficace.

Classiquement, le chemin critique est défini uniquement à l'envoi de messages, car dans ce cas la séquence d'opérations est définie statiquement. Pour pouvoir également appliquer les mêmes techniques à la réception, Mosberger *et al.* créent plusieurs versions du chemin critique (correspondant à chaque processus destinataire), et utilisent un filtre de paquets [82] pour faire la sélection parmi ceux-ci.

Ford *et al.* [48] révèlent toute une série d'optimisations qui deviennent possibles si l'on spécifie séparément l'interface d'un service RPC et sa présentation. Par présentation, ils comprennent tout ce qui reste normalement non-spécifié dans un langage de description d'interfaces (IDL). Ces aspects non-spécifiés, qui sont habituellement traités d'une manière implicite (donc inflexible), concernent typiquement : la copie des paramètres-pointeurs, l'allocation et la libération des tampons, etc. Ils offrent un moyen simple de spécifier une présentation flexible, par des annotations dans la spécification IDL. On peut utiliser cette flexibilité pour fournir une présentation personnalisée, optimisée pour une application.

Par ailleurs, ces annotations peuvent être utilisées pour spécialiser dynamiquement le chemin critique par rapport à chaque paire client-serveur, en fonction des attributs de présentation du client et du serveur. Ford *et al.* ont employé cette technique dans une mise en œuvre du protocole de transport de Mach. Pour effectuer la génération dynamique de code, ils mettent ensemble des fragments de code, correspondant à des opérations de base du RPC. Par exemple, une classe d'opérations de base concerne la sécurité de la transaction RPC. Le niveau de sécurité est paramétrable à travers un attribut de présentation spécifique ; si l'on exprime par ce paramètre une certaine confiance dans le client, quelques sauvegardes ou mises à zéro de registres ne sont plus inclues dans le code spécialisé. C'est un cas très intéressant de spécialisation, mais appliquée elle aussi à une échelle restreinte : les seuls invariants considérés sont les attributs de présentation, et la forme de spécialisation implémentée est assez rudimentaire.

Blackwell [17] optimise le déballage des arguments dans des protocoles qui supportent un format variable pour les messages, tels que Q.39B [49] ou ASN.1 [60]. Dans ces deux cas, on ne peut pas utiliser un compilateur statique de talons, tel que USC [92], puisque le nombre de formats n'est pas borné. Blackwell utilise alors de la génération dynamique de code pour créer une fonction de déballage spécialisée par rapport à un format de message déjà rencontré.

3.2 Discussion

En examinant les travaux antérieurs d'optimisation de RPC, on peut en extraire plusieurs leçons.

La première leçon est qu'il existe des optimisations qui sont spécifiques aux domaine considéré (c'est-à-dire soit applicables uniquement aux protocoles de communication soit spécifiques vraiment aux RPC), et d'autres optimisations qui ont une applicabilité plus générale. Parmi les optimisations spécifiques, on retrouve souvent par exemple l'élimination des copies de messages entre les différentes couches de protocoles, ou l'optimisation de l'ordonnancement (*scheduling*) en tirant profit du caractère synchrone des RPC.

Les techniques plus généralement applicables consistent à caractériser un chemin critique dans le code, et y appliquer des transformations telles que la fusion de couches, la propagation de constantes et l'élimination du code non-utilisé. L'application manuelle de ces techniques est une tâche pénible et peut être une importante source d'erreurs. Nous montrons dans le chapitre suivant comment ces transformations peuvent être effectuées automatiquement par évaluation partielle.

Une deuxième leçon intéressante est relative au choix même du chemin critique. Une partie des travaux définissent le chemin critique comme la trace d'exécution d'un appel RPC en général. D'autres travaux se focalisent sur un cas plus précis, en rajoutant des contraintes sur la localisation des processus, ou sur les architectures des machines client et serveur.

Enfin, quelques travaux [1, 17] utilisent pour l'optimisation des informations spécifiques à une unique paire client-serveur, pour générer dynamiquement du "code jetable" — extrêmement spécialisé et utilisable pendant une seule "session". Les RPC constituent ainsi un bon exemple de spécialisation incrémentale : plus on rajoute des contraintes pour la définition du chemin critique, plus on obtient un gain de performance.

La spécialisation automatique de code par évaluation partielle ouvre la voie à une généralisation massive de ces deux principes — la spécialisation incrémentale et la génération de code jetable.

Un autre point à retenir est l'importance de la couche de conversion de données (ou présentation) pour la performance globale des RPC, fait démontré par plusieurs travaux [28, 48]. C'est pour cette raison que nous avons choisi comme première cible de spécialisation automatique cette couche de présentation, comme décrit dans le chapitre 5.

Chapitre 4

L'évaluation partielle — une approche automatique à l'optimisation de programmes

Les deux chapitres précédents ont mis en évidence l'existence d'un important degré de généricité dans les services système (et en particulier dans le service d'appel de procédures à distance) et le conflit entre cette généricité et les performances des services. Ce chapitre présente l'évaluation partielle, une technique capable d'optimiser des programmes génériques, en les adaptant à un contexte particulier.

4.1 Généralités sur l'évaluation partielle

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

Un évaluateur partiel consiste en un ensemble réduit de règles de transformation de programmes visant d'une part à évaluer les expressions qui manipulent des données disponibles et d'autre part à reconstruire les expressions dépendantes des données manquantes.

La notion de spécialisation s'applique à une vaste classe de problèmes. En effet, l'évaluation partielle a été utilisée pour des applications aussi variées que la génération de compilateurs à partir d'interprètes [63], l'optimisation de programmes numériques [12] ou graphiques [7], le filtrage [30] et l'instrumentation de programmes [66].

4.1.1 Un exemple concret

Prenons comme exemple la fonction `printf()` — la fonction de formatage de texte couramment employée en C. Étant donné une chaîne de contrôle (le format d'affichage) et un ensemble de valeurs, cette fonction interprète la chaîne afin de déterminer comment

```
void mini_printf( char *fmt, int *value )
{
  int i = 0;
  while( *fmt != '\0' )
  {
    if( *fmt != '%' )
      putchar( *fmt );
    else
      switch(*++fmt) {
        case 'd': putint(value[i++]); break;
        case '%': putchar('%'); break;
        default : abort();  /* error */
      }
    fmt++;
  }
}
```

FIG. 4.1 – Définition de `mini_printf()`

```
void mini_printf_spec( int *value )
{
  putchar('n');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putint(value[0]);
}
```

FIG. 4.2 – Spécialisation de `mini_printf()` avec `fmt` égal à `"n = %d"`

formater et afficher les valeurs. On peut observer que, la plupart du temps, `printf()` est appelée avec un format constant. Cette situation est une occasion typique d'application de l'évaluation partielle, en particulier si `printf()` est appelée de manière répétée avec un même format constant.

Considérons `mini_printf()` (figure 4.1), une version simplifiée de la fonction `printf()` ordinaire où l'ensemble des valeurs est réduit à un tableau d'entiers et où la chaîne de contrôle ne comporte que deux directives de formatage : `%d` qui spécifie l'affichage d'un entier, et `%%` qui permet d'afficher le caractère `%` ; tous les autres caractères sont affichés tels quels.

La figure 4.2 présente la spécialisation de `mini_printf()` en fonction d'un format statique égal à la chaîne `"n = %d"`, alors que le tableau des valeurs reste inconnu, dynamique. C'est ce type de sortie que produit un évaluateur partiel. Toutes les opérations de manipulation du format ont été effectuées : le travail d'interprétation de la chaîne de contrôle a été totalement supprimé et il ne subsiste aucune trace de cette chaîne dans le programme résiduel. Les seules opérations restantes sont celles qui réalisent des effets de bord (comme `putchar()`), et celles qui dépendent de valeurs encore inconnues (tableau `value`).

4.1.2 Aspects extensionnels

Pour présenter l'évaluation partielle, il est important d'établir une distinction entre un programme (sa définition sous forme de texte) et la fonction (c'est-à-dire l'objet mathématique) que ce programme dénote. Pour ce faire nous utilisons la convention suivante : lorsqu'un nom apparaît en majuscule, il dénote un programme, sinon il dénote une fonction (ou une valeur). Nous ne définissons pas la correspondance entre un programme P et la fonction p qu'il dénote ; nous supposons que cette correspondance est donnée par la sémantique formelle du langage. Nous supposons de plus qu'il existe un évaluateur *eval* tel que :

$$eval(P, d) = p(d)$$

où d désigne des données d'entrée de P.

Étant donné un programme P à deux entrées et une valeur v, un évaluateur partiel est un programme, noté EP, calculant le *programme résiduel* P_v,

$$P_v = eval(EP, (P, v))$$

tel que, lorsque le programme résiduel est appliqué à la donnée manquante w,

$$eval(P_v, w) \equiv eval(P, (v, w))$$

Autrement dit, le programme original et le programme spécialisé produisent le même résultat. D'un point de vue fonctionnel, ceci peut être écrit comme suit :

$$p_v(w) \equiv p(v, w) \quad \text{où} \quad P_v = ep(P, v)$$

Plus généralement, v et w forment une partition des entrées de P. Les données disponibles pendant l'évaluation partielle sont dites *statiques* (les données v). Les données manquantes sont dites *dynamiques* (les données w) [63]. On dit que P_v est la *version spécialisée* de P en fonction de v.

4.1.3 Stratégies d'évaluation partielle

Concernant l'implémentation d'un évaluateur partiel, on distingue communément deux stratégies : *en ligne* (ou *online*) et *hors ligne* (ou *offline*) [62, chapitre 7]. La première stratégie consiste à déterminer le traitement du programme au fur et à mesure de la phase d'évaluation partielle. Les évaluateurs partiels reposant sur ce principe ont l'avantage de manipuler des valeurs concrètes, et donc, peuvent déterminer précisément le traitement de chaque expression d'un programme. Toutefois, ce processus est coûteux : l'évaluateur partiel doit analyser le contexte du calcul (c'est-à-dire les données disponibles) pour sélectionner la transformation de programme appropriée. Cette opération est effectuée de façon répétitive dans le cas d'une boucle, par exemple. Il n'est donc pas surprenant de constater que les performances d'un évaluateur partiel en ligne se dégradent rapidement lorsque de nouvelles transformations de programmes sont introduites.

La deuxième stratégie d'évaluation partielle est dite hors ligne. Elle comporte deux phases : une *analyse de temps de liaison* et une phase de *spécialisation*. Étant donné un programme et une description de ses entrées (lesquelles sont connues ou inconnues), l'analyse de temps de liaison détermine les expressions qui peuvent être évaluées lors de la phase d'évaluation partielle et celles qui doivent être reconstruites : les premières sont dites statiques, les autres dynamiques. Les informations de temps de liaison sont valides tant que la description des entrées du programme reste inchangée. Les expressions statiques et dynamiques étant connues à l'avance, la phase de spécialisation est plus efficace, car les transformations appliquées peuvent être calculées avant de disposer des vraies valeurs. Toutefois, l'analyse de temps de liaison, manipulant des valeurs abstraites, doit effectuer certaines approximations. Ainsi, le degré de spécialisation d'une stratégie hors ligne peut être moindre que celui d'une stratégie en ligne.

L'activité importante qui s'est développée dans le domaine de l'évaluation partielle a conduit à la réalisation de nombreux prototypes pour une variété de langages de programmation tels que Scheme [19, 29], C [6], Pascal [80] et Fortran [11].

4.1.4 Évaluation partielle à l'exécution

Traditionnellement, l'évaluation partielle est une transformation source-vers-source, c'est-à-dire qui opère sur le texte d'un programme. De ce fait, elle intervient à la compilation et ne peut exploiter que les valeurs disponibles à ce stade. Or, il est des cas où des invariants ne sont connus qu'au lancement du programme, et d'autres encore où le statut d'invariant n'est vrai que pendant une certaine période d'exécution du programme — on parle alors de *quasi-invariants*.

L'évaluation partielle à l'exécution permet d'effectuer une spécialisation durant l'exécution même du programme, au moment où les quasi-invariants deviennent connus. Pour des raisons évidentes d'efficacité du processus de spécialisation, cette transformation ne peut plus être une transformation source-vers-source, car ceci impliquerait d'appeler un compilateur au vol. L'évaluation partielle à l'exécution doit donc produire directement du code exécutable.

4.2 Tempo, un spécialiseur pour le langage C

Dans le projet Compose, nous avons développé un évaluateur partiel pour le langage C, nommé Tempo [34]. Tempo est un spécialiseur hors ligne. Étant donné qu'il traite le langage C, l'étape d'analyse inclue une phase qui détermine les alias des pointeurs et les effets de bord, avant de calculer les temps de liaison.

4.2.1 Adéquation aux applications système

Tempo a été spécialement conçu pour pouvoir traiter des applications système de taille réelle. En fait, toute la puissance d'un évaluateur partiel hors ligne tient dans la finesse de ses analyses. Pour s'assurer que les analyses implémentées par Tempo sont adéquates, nous avons d'abord recensé les besoins de spécialisation existants dans des échantillons de code système. Le résultat de cette étude nous a conduit à définir des analyses dans Tempo (et notamment l'analyse de temps de liaison) pouvant satisfaire les propriétés suivantes :

- *structures partiellement statiques :* un temps de liaison individuel est donné à chaque champ d'une structure, au lieu d'un temps de liaison global pour toute la structure. Cette information est propagée interprocéduralement [57].
- *sensibilité au flot de contrôle* : le temps de liaison d'une variable n'est pas global ; il peut varier en fonction du point d'exécution du programme [56].
- *sensibilité au contexte d'appel* : plusieurs instances d'une même fonction avec différents temps de liaison pour les arguments peuvent coexister (on parle de *polyvariance de temps de liaison*) [56].
- *sensibilité au contexte d'utilisation* : il est courant que l'état du système soit représenté par de multiples structures de données, complexes et interconnectées par des pointeurs. Très souvent, l'état est partiellement statique, et donc utilisé à la fois dans des calculs statiques et dynamiques. La sensibilité au contexte d'utilisation admet un temps de liaison dit *statique et dynamique*, qui permet d'exploiter les valeurs d'expressions statiques tout en forçant leur résidualisation pour les autres calculs [57].
- *sensibilité aux valeurs de retour* : exploitation du résultat statique d'une fonction qui doit malgré tout être résidualisée parce qu'elle effectue des effets de bord dynamiques [56].

L'implémentation de l'analyse de temps de liaison est décrite en détail par Hornof dans sa thèse [55].

Cette précision des analyses, développée initialement pour des applications système, a permis par ailleurs l'application avec succès de Tempo dans d'autres domaines comme le calcul scientifique [90] ou les langages dédiés [111].

4.2.2 Spécialisation à l'exécution

Avec Tempo, il est possible de spécialiser un programme non seulement à la compilation mais aussi à l'exécution [36]. Cette capacité de Tempo ouvre la porte à de nombreuses applications [90], en particulier dans les programmes systèmes où l'on trouve plus souvent des quasi-invariants (représentant notamment un état temporaire d'un sous-système) que de véritables invariants.

Techniquement, la spécialisation à la compilation et à l'exécution partagent la même analyse. Dans le second cas, l'analyse est suivie de la production automatique de fragments de code source qui vont nous servir à assembler au vol le programme spécialisé. Ces fragments de code source sont incomplets dans la mesure où les invariants ne sont connus qu'à l'exécution, mais ils peuvent déjà être compilés. Durant l'exécution, il faut sélectionner et copier certains de ces fragments compilés, y insérer les valeurs connues et reloger quelques sauts afin d'obtenir la spécialisation souhaitée.

Ces opérations sont relativement simples et permettent donc un processus de spécialisation très efficace qui ne nécessite qu'un nombre limité d'exécutions du programme spécialisé avant d'être amorti.

4.2.3 Interface

Un spécialiseur hors ligne, comme Tempo, offre l'avantage d'un comportement prévisible. En effet, les transformations de programme sont décidées au moment de l'analyse, avant que la spécialisation effective n'ait lieu. Toutefois, il faut un moyen de rendre exploitables les résultats de cette analyse. À cet effet, Tempo dispose de fichiers de sortie colorés au format MIME ou HTML et qui peuvent être visualisés sous `Emacs` ou avec un navigateur Web. L'information fournie comprend : les temps de liaison, la polyvariance, la liste des alias des pointeurs dé-référencés, l'usage des variables globales. Sont affichées également, pour chaque construction syntaxique, les transformations de programme décidées par l'étape d'analyse.

L'utilisateur dispose également de plusieurs fichiers qui lui permettent de décrire le contexte d'analyse initial (alias et temps de liaison) et les valeurs effectives de spécialisation.

4.3 Comparaison avec d'autres techniques d'optimisation

Il est intéressant de comparer l'évaluation partielle avec d'autres techniques d'optimisation de programmes, aussi bien automatiques que manuelles, afin de mieux comprendre son applicabilité, ses avantages, et ses limites.

Un premier parallèle peut être dressé entre l'évaluation partielle et les optimisations automatiques incorporées couramment dans des compilateurs commerciaux (on peut en trouver un tour d'horizon assez complet dans Bacon *et. al* [10]). Il existe en effet une grande similarité entre les effets de l'évaluation partielle et les effets combinés de quelques optimisations classiques, à savoir la propagation de constantes, le dépliage de fonctions, l'élimination du code non-utilisé. Nous décrivons par la suite l'évolution historique de la propagation de constantes, comme étant la plus proche de l'évaluation partielle.

La propagation de constantes a été initialement définie pour des langages impératifs, comme une optimisation intra-procédurale et portant uniquement sur des variables scalaires [3]. Depuis, d'importantes améliorations y ont été apporté, comme par exemple la propagation inter-procédurale [120], et couplée ultérieurement avec le dépliage de fonctions ou le clonage des procédures [39], la prise en compte de certains types de données comme les nombres complexes et les tableaux [79]. En revanche, les propagateurs de constantes existants ne traitent pas le cas des structures de données définies par l'utilisateur, et ni leur manipulation à travers des pointeurs.

Les effets de l'évaluation partielle peuvent être vus aussi comme une forme inter-procédurale de factorisation de code (*code hoisting*, ou encore *loop-invariant code motion*) [10], qui consiste à déplacer des instructions dans le programme afin de réduire leur fréquence d'utilisation (par exemple de déplacer les instructions invariantes hors d'une boucle). Ce même effet est obtenu en évaluation partielle — mais d'une manière bien plus agressive (interprocédurale, polyvariante, ...) —, par l'exécution statique de ces instructions lors de la spécialisation.

Malgré la similarité des effets, l'évaluation partielle et les optimisations intégrées dans les compilateurs visent des cibles bien différentes. Un compilateur optimisant doit obéir à une contrainte d'universalité, en garantissant des bonnes performances pour tout programme compilé. En revanche, l'évaluation partielle s'adresse à la classe bien spécifique des programmes génériques. Sur cette classe de programmes, on obtient souvent des amélioration conséquentes, dépassant parfois un ordre de grandeur ; le gain n'est toutefois pas garanti, car dans certains cas les transformations effectuées par l'évaluation partielle peuvent même ralentir le programme initial.

Notons aussi qu'il existe une complémentarité entre les optimisations apportées par l'évaluation partielle et celles effectuées par les compilateurs. En effet, l'évaluation partielle à la compilation produit un programme source, qui sera après compilé et optimisé par un compilateur standard. Dans le cas de l'évaluation partielle à l'exécution, on utilise toujours un compilateur pour produire les fragments de code binaire à assembler pendant l'exécution.

Récemment, une autre forme de spécialisation de programmes a été exploré dans des langage impératifs : la spécialisation orientée données (*data specialization*) [68]. Cette technique est basée essentiellement sur la même étape d'analyse que l'évaluation partielle, mais recourt à des transformations de programmes différentes. Notamment, aucune transformation n'est appliquée au flot de contrôle du programme. Les transformations introduisent uniquement des structures de données (appelées *caches*) qui accumulent les résultats des calculs statiques et évitent de les calculer de manière répétée. Avec cette technique, la spécialisation orientée données peut obtenir des effets très similaires à l'évaluation partielle à l'exécution (car les valeurs statiques ne sont pas utilisées avant l'exécution par l'étape de transformation), et ce avec un moindre surcoût (car n'impliquant pas de génération dynamique de code). Cependant, la qualité des optimisation est moindre, justement parce qu'elle ne vise pas à restructurer le flot de contrôle. Selon l'application considérée, les avantages ou les inconvénients de chaque approche peuvent primer ; de fait, une future version de Tempo offrira les deux stratégies.

En comparaison avec d'autres formes de spécialisation, manuelles ou automatiques, on peut citer quelques exemples de transformations qui ne sont pas effectuées par l'évaluation partielle. Typiquement, aucune transformation n'est appliquée à des données dynamiques. En particulier, des optimisations classiques comme les simplifications algébriques, la propagation de variables (*copy propagation*) ou la propagation d'expressions [10] ne sont effectuées que sur des données complètement statiques. Par exemple, dans une suite d'affectations $x_2 = x_1; x_3 = x_2$, la première affectation n'est réduite que si tout le flot de données est statique (dans notre cas, il suffirait que x_1 soit statique). En fait, dans ce cas simple de propagation de variable, l'optimisation pourra être effectuée automatiquement par le compilateur sous-jacent à l'évaluateur partiel, ce qui illustre bien la complémentarité des deux outils. Dans des cas plus compliqués, où les données dynamiques à propager concernent des structures de données complexes (par exemple des tampons de messages), leur optimisation reste pour l'instant l'apanage des optimisations manuelles.

Une autre limitation de principe de l'évaluation partielle est qu'elle ne peut jamais modifier la complexité algorithmique du programme. Il a été prouvé en effet, sur un langage impératif simple, que cette transformation apporte au plus un facteur d'accélération constant (c'est-à-dire, qui ne dépend pas de la taille des données dynamiques) [62].

4.4 Conclusion

L'évaluation partielle offre une alternative viable à l'optimisation manuelle du code système, car elle incorpore des transformations de code correspondant aux optimisations manuelles les plus courantes : dépliage de fonctions, propagation des valeur connues, réduction des conditionnelles avec un test décidable statiquement, etc. L'intérêt premier de cette approche est d'éviter l'introduction d'erreurs lors de la spécialisation.

L'évaluation partielle peut aller encore plus loin que les optimisations manuelles ou même les optimisation automatiques classiques, de plusieurs manières. Premièrement, l'automatisation du traitement permet de s'attaquer à un volume de code en principe illimité, et surtout d'appliquer ce traitement de manière systématique. Deuxièmement, la possibilité d'optimiser le code au vol pendant l'exécution permet à l'évaluation partielle d'exploiter des constantes connues en cours d'exécution ou des invariants éphémères.

Deuxième partie

Spécialisation automatique de composants système

Les chapitres 2 et 3 ont montré le besoin d'extensions système spécifiques aux applications, pour des raisons de performance. Pour généraliser l'utilisation de telles extensions personnalisées, nous proposons une approche consistant à spécialiser automatiquement des composants génériques. Notre approche repose sur l'utilisation de Tempo, notre évaluateur partiel de programmes C, spécialement conçu pour la spécialisation de programmes système.

Cette solution apporte une meilleure maintenabilité, car les modifications ultérieures concerneront uniquement la version générique. Les modifications peuvent être repercutées ensuite sur toutes les versions optimisées par une simple re-spécialisation.

Un autre avantage de cette approche automatique est de résoudre les problèmes traditionnels de sûreté d'un système extensible présentés dans la section 2.1, en garantissant que les extensions rajoutées statiquement ou dynamiquement conservent la sémantique du composant générique, tout en l'accélérant pour un contexte d'utilisation particulier.

Enfin, l'utilisation systématique de la spécialisation automatique ouvre la voie à la spécialisation incrémentale, et à la production d'instances de code "jetables", ultra-spécialisées.

Notre approche peut être appliquée à deux classes de composants génériques : soit des composants conçus et développés spécifiquement dans un but de spécialisation, soit des composants systèmes existants qui recèlent des opportunités de spécialisation. Ce dernier cas présente évidemment un plus grand défi, mais en revanche, la spécialisation permet de réutiliser et d'optimiser du code existant.

Les deux chapitres suivants montrent la faisabilité de la spécialisation des composant système existants, à travers deux exemples complets de spécialisation de code industriel, en l'occurrence l'implémentation du protocole RPC de Sun et le module d'IPC du micro-noyau Chorus.

Chapitre 5

Spécialisation de l'implémentation RPC de Sun

Le protocole RPC de Sun a été proposé en 1984 comme support pour le développement d'applications distribuées. Depuis, ce protocole est devenu un standard *de facto* dans la conception et l'implémentation des services distribués, comme par exemple NFS [81] et NIS [100].

L'implémentation RPC de Sun est structurée en deux parties (voir figure 5.1) :

1. Le *talon* est la partie du protocole spécifique à chaque application. Le talon est produit par un compilateur de talons appelé `rpcgen`, à partir de la spécification d'interface (IDL) d'un service RPC. Cette spécification décrit l'interface du service dans un langage spécialisé. Le talon implémente la couche de conversion des arguments entre le format local et un format indépendant de la machine, appelé XDR (pour eXternal Data Representation).

2. Le reste du protocole est indépendant de l'application, et assure l'envoi et la réception des messages (entre le client et le serveur) nécessaires au protocole RPC. Cette couche utilise à son tour un protocole sous-jacent tel que la couche des *sockets* de BSD Unix.

À un niveau plus fin, le protocole RPC de Sun consiste en plusieurs micro-couches, chacune dédiée à une tâche simple. Par exemple, il y a des micro-couches pour la conversion d'un type de base, pour la conversion d'une structure de données complexe, pour sélectionner les paramètres appropriés des protocoles sous-jacents, etc. De ce fait, la structuration de Sun RPC en micro-couches est représentative d'une architecture modulaire utilisée dans un logiciel du commerce.

5.1 Un exemple simple

Considérons un exemple simple pour illustrer le fonctionnement de cette architecture en micro-couches. Le service `rmin()` est utilisé pour calculer le minimum de deux entiers, sur un serveur distant.

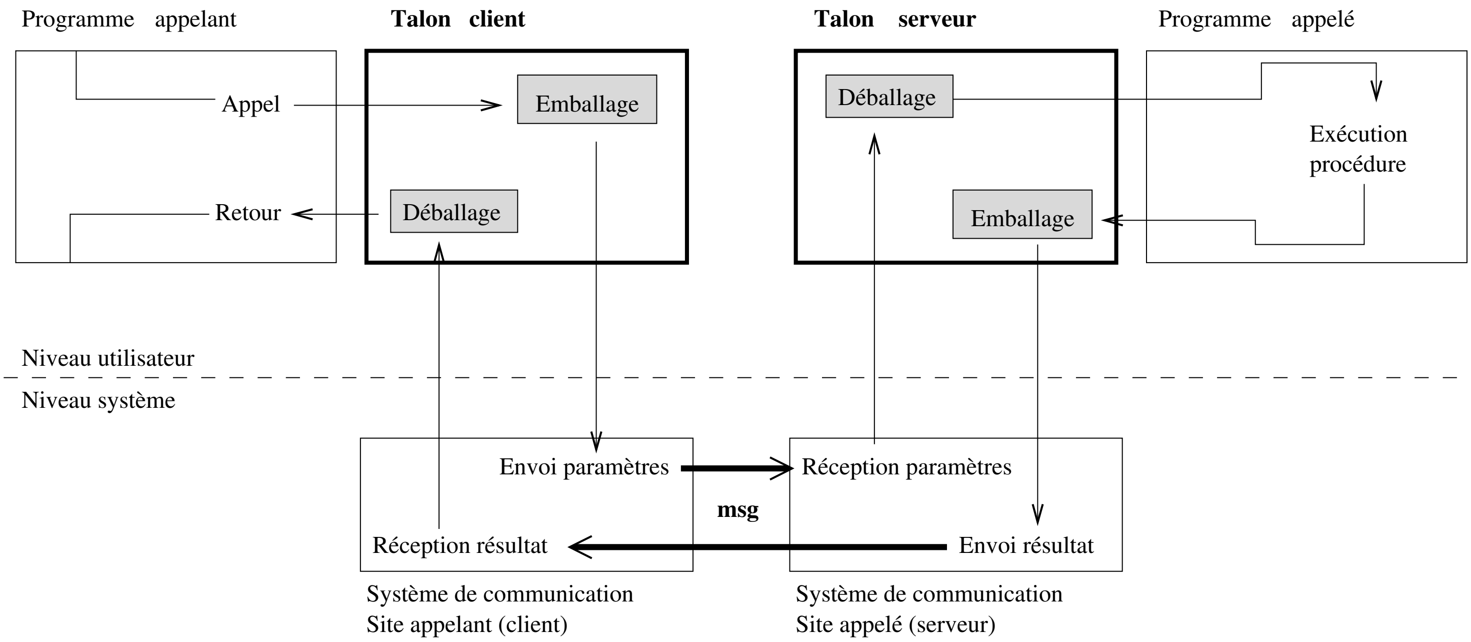


FIG. 5.1 – *Le protocole RPC de Sun*

Le client utilise le compilateur de talons `rpcgen` pour compiler la spécification d'interface correspondant à `rmin()` en un ensemble de fichiers C. Ces fichiers implémentent d'une part l'appel sur le site client, et d'autre part la distribution des requêtes vers les services sur le site serveur. Plutôt que d'inclure tous ces fichiers, la figure 5.2 montre une trace d'exécution simplifiée d'un appel à `rmin()`.

5.2 Opportunités de spécialisation dans le protocole RPC de Sun

Dans le protocole RPC de Sun, la généricité intervient sous la forme d'une interprétation répétée de quelques descripteurs (c'est-à-dire structures de données). Ces descripteurs sont examinés dans plusieurs couches pour diriger la gestion des tampons, le choix des protocoles à utiliser (TCP ou UDP) ou de l'action à effectuer (encodage ou décodage), etc. La plupart de ces paramètres sont fixés pour un RPC donné. Par conséquent, cette information peut être exploitée pour éliminer l'interprétation et produire du code spécialisé pour un RPC particulier. Considérons ces opportunités de spécialisation plus en détail, pour voir comment elles peuvent être exploitées par évaluation partielle.

Les structures de données les plus importantes impliquées dans le processus d'interprétation sont le descripteur d'un client et le descripteur d'un tampon d'encodage. Certains champs de ces descripteurs ont des valeurs statiques, c'est-à-dire qu'elles ne dépendent pas des arguments d'un appel RPC. Pourtant, les valeurs de ces champs sont interprétées et propagées à travers les couches à chaque envoi de message. Il s'ensuit qu'une source d'optimisation serait d'effectuer pendant la spécialisation tous les calculs qui en dépendent. En

```
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)  //       Écriture 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. 5.2 – *Trace abstraite de la partie encodage d'un appel distant à* `rmin`

voici quelques exemples.

5.2.1 Sélection statique entre encodage et décodage

Une telle forme d'interprétation apparaît dans la fonction `xdr_long()` détaillée dans la figure 5.3 Cette fonction peut effectuer plusieurs opérations sur un entier, dont l'encodage et le décodage. L'opération appropriée est choisie selon la valeur du champ `x_op` de l'argument `xdrs`. Cette forme d'interprétation est utilisée de manière similaire pour tous les autres types de données.

En fait, la valeur du champ `x_op` est fixée par le contexte d'exécution (processus d'encodage ou de décodage). Cette information peut être propagée de manière inter-procédurale jusqu'à la fonction `xdr_long()` à travers la structure `xdrs` (voir figure 5.3). Ainsi, la sélection selon `xdrs->x_op` est complètement éliminée ; la version spécialisée de cette fonction est réduite à l'instruction de retour. En fait, dans notre cas, la version spécialisée sera complètement éliminé par dépliage, en raison de sa petite taille.

```
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);  //    Écrire 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. 5.3 – *Lecture ou écriture d'un entier long :* `xdr_long()`

5.2.2 Vérification statique du débordement des tampons

La vérification du débordement des tampons lors de l'encodage ou décodage constitue une autre forme d'interprétation. Cette situation intervient dans la fonction `xdrmem_putlong()` détaillée dans la figure 5.4. Concrètement, pendant le processus d'encodage, l'espace restant disponible dans le tampon est comptabilisé par le champ `xdrs->x_handy`. Ce champ est initialisé avec une valeur statique, et par la suite testé et décrémenté avec des valeurs statiques, à chaque appel à `xdrmem_putlong()` et aux procédures similaires. Puisque ce processus implique uniquement des valeurs statiques, toutes les vérifications de débordement peuvent être effectuées lors de la spécialisation. Les seules opérations restant à être effectuées dans le programme spécialisé sont les copies dans le tampon (si toutefois aucun débordement n'a été constaté).

```
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. 5.4 – *Écriture d'un entier long :* `xdrmem_putlong()`

Ce deuxième exemple est important non seulement à cause du gain de performance qu'il apporte, mais aussi parce qu'il montre la différence avec une optimisation manuelle consistant à éliminer indifféremment toutes les vérifications de débordement des tampons. Dans notre cas, les vérifications sont éliminées de manière sûre et systématique, en préservant strictement la sémantique du programme d'origine.

5.2.3 Propagation des codes de retour

Le troisième exemple utilise les résultats des exemples précédents. La valeur de retour de la procédure `xdr_pair()` (détaillée dans la figure 5.5) dépend de la valeur de retour de `xdr_int()`, qui dépend à son tour de la valeur de retour de `xdr_putlong()`. Nous avons vu que `xdr_int()` et `xdr_putlong()` ont des valeurs de retour statiques. Il en résulte que la valeur de retour de `xdr_pair()` est statique aussi. Si on spécialise aussi l'appelant de `xdr_pair()` (c'est-à-dire `clntudp_call()`) par rapport à cette valeur de retour, `xdr_pair()` n'a plus besoin de retourner une valeur du tout ; son type de retour peut être transformé dans `void`. La version spécialisée de `xdr_pair()`, avec les appels à `xdr_int()` et `xdr_putlong()` dépliés, est montré en figure 5.6. Le résultat effectif est toujours `TRUE` indépendamment de la valeur du paramètre dynamique `objp`, car l'encodage de deux entiers ne peut jamais provoquer un débordement. Lors du dépliage, ce résultat est utilisé dans la fonction appelante `clntudp_call()` (omise dans la figure) pour réduire un autre test.

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

FIG. 5.5 – *Routine d'encodage `xdr_pair()` utilisée dans `rmin()`*

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

5.2.4 Discussion

Le but du processus d'encodage est de copier les arguments dans le tampon de sortie. Pour réaliser cette tâche, le code minimal qu'on peut espérer est probablement celui de la figure 5.6. En fait, en appliquant Tempo au code d'origine on obtient exactement ce code spécialisé. Les propriétés de l'analyse de temps de liaison intégrée dans Tempo, qui ont été énumérées dans le chapitre 4, sont essentielles dans cet exemple :

- Le traitement inter-procédural de structures partiellement statiques a permis de propager les champs statiques des descripteurs (le descripteur de tampon par exemple) à travers les micro-couches.
- La sensibilité au contexte d'appel a permis de créer plusieurs versions des procédures génériques, telles que `xdr_long()` — une pour l'encodage, une autre pour le décodage.
- La sensibilité au contexte d'utilisation a été indispensable pour la propagation des pointeurs vers les structures partiellement statiques, tels que le paramètre `xdrs` de la procédure `xdrmem_putlong()` par exemple. Dans le corps de la procédure, les utilisations dynamiques de ce pointeur coexistent avec des utilisations statiques, sans provoquer d'interférences.
- La sensibilité aux valeurs de retour a permis d'exploiter de manière statique les codes d'erreur propagés entre les couches, pour optimiser le cas courant (dans lequel aucune exception n'est déclenchée).

5.3 Évaluation de performances

Nous avons spécialisé à la fois le côté client et le côté serveur de l'implémentation RPC de Sun datant de 1984 [86, 89, 88]. Le code original était de 1500 lignes pour le client (commentaires exclus) et 1700 lignes pour le serveur. Nous avons écrit un petit programme de test qui utilise des appels de procédures à distance, simulant le comportement typique des programmes parallèles impliquant des échanges importants de données structurées. Ceci est un cas représentatif pour une large classe d'applications qui utilisent un réseau de stations comme une machine massivement parallèle.

Les mesures ont été effectuées sur deux types de matériel :

- Deux stations de travail Sun IPX 4/50 tournant sous SunOS 4.1.4, avec 32MB de mémoire et connectées par un lien ATM à 100Mbits/s. Les cartes ATM sont de type Fore Systems, modèle ESA-200. Cette configuration est relativement inefficace, à la fois en termes d'unité centrale, de latence et de débit réseau. À titre de comparaison, le débit réseau d'une configuration actuelle varie typiquement entre 155Mbits/s et 622Mbits/s.
- Deux machines récentes compatibles PC, à base de Pentium, tournant sous Linux, avec 96MB de mémoire et connectées par un réseau Fast-Ethernet à 100 Mbits/s. Le réseau n'était pas partagé avec d'autres machines pendant les tests.

Pour évaluer l'impact de la spécialisation, nous avons effectué deux séries de mesures : (1) des micro-mesures sur le talon du client, et (2) des mesures d'un aller-retour de RPC, tel qu'il était perçu au niveau de notre application de test. Cette application exécute une boucle invoquant un simple RPC qui envoie et récupère un tableau d'entiers. Le but de cette deuxième expérience est de prendre en compte les paramètres propres aux architectures, tels que le cache, la bande passante de la mémoire et du réseau, qui ont un impact significatif sur la performance globale. Les résultats des évaluations de performances sont donnés en figure 5.7. Les chiffres correspondants aux micro-mesures et au mesures d'aller-retour représentent une moyenne sur 10.000 itérations.

Il n'est pas surprenant que la configuration PC/Linux soit toujours plus rapide que la configuration IPX/SunOS. Ceci est dû en partie aux vitesses des unités centrales, mais aussi au fait que les cartes Fast-Ethernet aient une latence moindre et une bande passante supérieure aux cartes ATM utilisées. Une conséquence de la spécialisation est que le décalage entre les deux configurations diminue sensiblement (voir les mesures comparatives dans la figure 5.7-1/5.7-2 et 5.7-3/5.7-4).

Micro-mesures. Les résultats détaillés des micro-mesures sur la couche d'encodage sont donnés dans le tableau 5.1. Le code du talon client spécialisé est jusqu'à 3,7 fois plus rapide sur les IPX/SunOS, et jusqu'à 3,3 fois plus rapide sur les PC/Linux.

Taille	IPX/SunOs			PC/Linux		
du tableau	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. 5.1 – *Performance de l'encodage client en ms*

Intuitivement, on s'attendrait à ce que le facteur d'accélération s'accroisse avec la taille du tableau, car le poids de la boucle d'encodage relatif au temps total devient plus grand. Pourtant, sur le Sun IPX, le facteur d'accélération diminue avec la taille du tableau (voir figure 5.7-5). La raison en est que dans le code spécialisé la boucle d'encodage a été déroulé complètement, devenant essentiellement une séquence de copies mémoire. Ceci a l'effet positif d'éliminer toutes les opérations statiques (vérifications de débordement par exemple), mais a aussi l'effet négatif d'accroître la taille du code. À partir d'une certaine taille du tableau, le code spécialisé ne rentre plus dans le cache d'instructions, d'où le fait que les performances du code spécialisé diminuent brusquement. Cet effet n'est pas visible dans les mesures sur PC parce que son cache d'instructions est plus grand, et les tailles de tableaux considérées n'ont pas atteint le seuil de débordement.

Il est à noter que pendant les mesures, le cache n'est pas partagé avec d'autres applications, ce qui favorise la version non-optimisée, plus courte, qui sera donc toujours dans le cache d'instructions. Dans un cas réel, le cache d'instructions sera utilisé en concurrence par

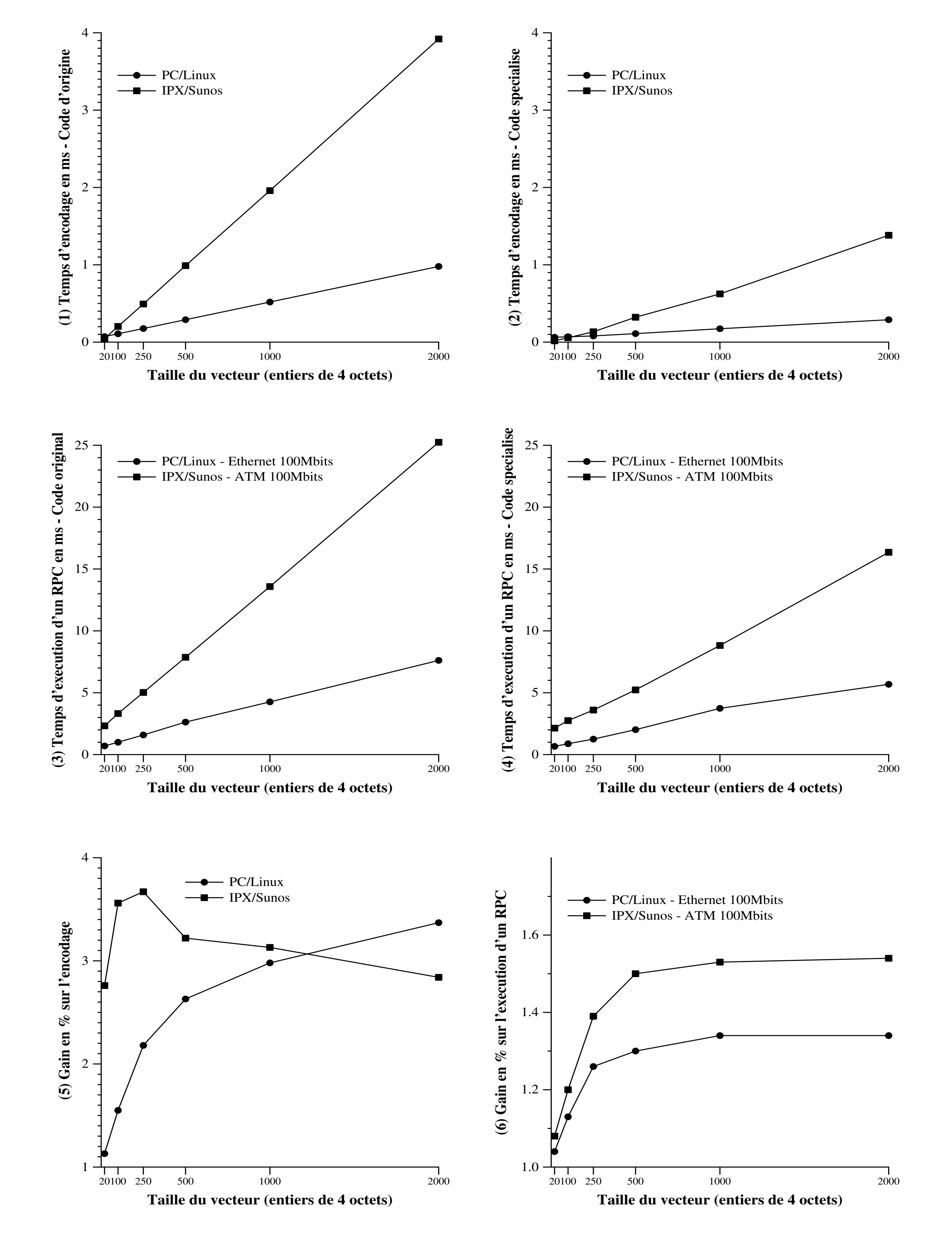


FIG. 5.7 – *Comparaison de performance entre IPX/SunOS et PC/Linux*

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. 5.2 – *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. 5.3 – *Taille des binaires SunOS (en octets)*

plusieurs tâches ou fils d'exécution différents, et très probablement aucune des versions ne sera contenue complètement dans le cache d'instructions. Par conséquent, la différence de vitesse même sur des tableaux de grande taille tendra à s'estomper. On peut conclure que du point de vue de cet effet de cache, notre expérience considère le cas le plus défavorable.

RPC aller-retour. Les temps d'un aller-retour RPC mesurés au niveau application sont détaillées dans le tableau 5.2. Le code spécialisé est jusqu'à 1,55 fois plus rapide sur les IPX/SunOS, et jusqu'à 1,35 fois plus rapide sur les PC/Linux. Les facteurs d'accélération sont plus modestes que ceux des micro-mesures à cause du temps passé sur le réseau, qui, lui, est incompressible.

On remarque le fait que l'effet de cache sur les IPX a disparu, parce que le chemin d'exécution est beaucoup plus long, même dans la version non-optimisée — il traverse toutes les couches de la pile de protocoles sous-jacente (sockets, UDP, ...). Par conséquent, l'impact du cache diminue, et la diminution du nombre d'instructions par spécialisation reste toujours visible.

Taille du code. Comme on peut le constater dans le tableau 5.3, le code spécialisé est toujours plus grand que le code d'origine. La raison en est bien-sûr le déroulage de la boucle d'encodage. De plus, le code spécialisé se rajoute toujours au code générique plutôt que de le remplacer, car en cas de réception d'un paquet d'erreur on doit encore utiliser le code générique qui inclue le traitement d'erreurs.

5.4 Expérience acquise

L'application de Tempo au protocole RPC de Sun nous a permis d'apprendre plusieurs leçons.

5.4.1 Utilisation de Tempo

Le succès de la spécialisation sur cet exemple démontre que la précision des analyses intégrées dans Tempo est bien adaptée aux besoins courants du code système.

Un autre aspect qui s'est révélé important pour pouvoir spécialiser des composants de cette taille est l'interface avec l'utilisateur. Comme décrit dans la section 4.2.3, Tempo offre la possibilité d'afficher une version coloriée du code selon les attributs de temps de liaison qui ont été calculés pour chaque construction syntaxique. De cette manière, l'utilisateur peut vérifier si les parties statiques (flot de contrôle, variables, expressions, ...) correspondent bien à l'intuition préalable. Avec cette méthode, on peut détecter des inconsistances dans le contexte de spécialisation (et les corriger, le cas échéant) avant de passer à la phase de spécialisation proprement-dite.

5.4.2 Optimisation d'un code existant

Une leçon importante tiré de cette expérience est que la spécialisation d'un code existant nécessite une forme particulière d'expertise. Comme toute technique d'optimisation, l'évaluation partielle est sensible à différentes caractéristiques d'un programme, telles que la structure du contrôle et l'organisation des données. En conséquence, la spécialisation d'un programme existant demande une connaissance de l'algorithme et des structures des données suffisante pour estimer les calculs qui devraient être statiques. Cette méthodologie est opposée à la plupart des expériences antérieures d'évaluation partielle, dans lesquelles le programme était écrit dans le but spécifique d'être spécialisé.

5.4.3 Intégration du code spécialisé

À travers cet exemple, nous avons pris conscience qu'après la production automatique du code spécialisé, il reste encore une étape à effectuer manuellement. Cette étape consiste à intégrer les fragments de code spécialisés dans le composant d'origine.

Dans notre cas, la spécialisation implique uniquement les talons RPC, qui étaient placés dans des bibliothèques utilisateur. On peut choisir donc comme solution d'extensibilité de rajouter le code spécialisé aux mêmes bibliothèques, qui seront liées au client, et respectivement au serveur.

La prise en compte du nouveau code au niveau du flot de contrôle est différente sur le chemin d'envoi et de réception du client. Pour le chemin de l'envoi de la requête RPC, le code spécialisé pour l'emballage remplace simplement le code d'origine, car les prémisses de la spécialisation (c'est-à-dire les invariants utilisés) sont valides à chaque appel RPC de ce service. En revanche, pour le chemin du retour, le code spécialisé correspond uniquement au cas d'une réponse normale à la requête (c'est-à-dire quand aucune erreur ne s'est produite). En termes d'invariants, le contexte de spécialisation stipulait que la taille du paquet reçu serait celle d'une réponse normale. Pour assurer la cohérence entre le contexte d'exécution dynamique et la version du code utilisée, nous avons rajouté le test suivant dans le protocole :

```
taille_recue = <dynamique>;
if (taille_recue == taille_attendue) {
      <code_specialise>;
} else {
      <code_generique>;
}
```

La valeur de `expected_inlen` peut être déduite en principe directement de la spécification d'interface du service RPC. En pratique, elle peut être obtenue par un appel fictif de la fonction d'encodage du serveur.

5.5 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 5.2 (c'est-à-dire, 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 composants système.

Chapitre 6

Spécialisation des IPC du micro-noyau Chorus

Ce chapitre présente un deuxième exemple complet de génération automatique d'une extension système par spécialisation. Il s'agit de la spécialisation du module IPC du micro-noyau Chorus. Par rapport au chapitre précédent, cet exemple illustre en plus :

- la comparaison entre une spécialisation à la compilation et une spécialisation à l'exécution d'un même service;
- l'utilisation des instances de code jetables, dépendant d'un ensemble de quasi-invariants.

Le chapitre commence par présenter la famille de systèmes d'exploitation Chorus/OS, en montrant l'intérêt d'optimiser les IPC du micro-noyau sous-jacent. Ensuite, nous décrivons l'architecture et l'implémentation des IPC de Chorus, afin de présenter les opportunités de spécialisation dans la section 6.2. La suite du chapitre présente les résultats de la spécialisation, et la dernière section présente quelques conclusions.

6.1 Présentation de Chorus

La famille Chorus/OS couvre une gamme de systèmes d'exploitation allant de très petits systèmes temps réel en passant par des systèmes entièrement compatibles Posix/RT, et jusqu'à des systèmes distribués comprenant des serveurs Unix. La famille est composée de :

- Chorus/Micro : typiquement utilisé pour des environnements embarqués disposant d'une mémoire très limitée (de l'ordre de 10K), tels que les téléphones mobiles, les lecteurs de cartes, etc.
- Chorus/ClassiX : typiquement utilisé dans des environnements mono-site avec plusieurs cartes ou bien dans des configurations distribuées demandant un plus haut niveau de fonctionnalités. Chaque système offre une interface compatible Posix/RT et est capable d'exécuter des multiples applications temps-réel, et des multiples personnalités système.

– Chorus/JaZZ : un système d'exploitation fournissant un environnement complet d'exécution Java, comprenant une machine virtuelle et l'ensemble de services associés nécessaires.

Toute la famille Chorus/OS repose sur la technologie micro-noyau, et utilise donc de manière intensive des primitives de communication inter-processus (IPC). Nous nous intéressons ici à l'optimisation des IPC de la version 3 du micro-noyau Chorus.

6.1.1 Les IPC de Chorus

Dans les systèmes d'exploitation traditionnels (non-distribués), le terme IPC réunit les moyens d'interaction entre des processus ou des fils d'exécution différents, situés éventuellement dans des espaces d'adressage distincts. Ceci inclut des mécanismes de communication et de synchronisation tels que : la mémoire commune, les signaux, les sémaphores, les tubes (*pipes*) et les *sockets* BSD.

Dans les systèmes d'exploitation distribués (dont la famille Chorus/OS), on désigne par IPC les primitives de *communication par messages* entre des processus ou des fils d'exécution différents, situés éventuellement sur des sites différents. Dans la suite de ce chapitre, nous considérons pour les IPC cette dernière définition.

Les IPC offrent un moyen de communication transparent à la localisation des processus : les mêmes primitives sont utilisées dans le cas local (processus sur le même site) ou distant (processus sur de sites différents).

Dans Chorus les messages d'IPC peuvent être envoyés de deux manières différentes :

1. *asynchrone :* par une primitive d'envoi non-bloquante d'un message `ipcSend()`. Ce type de message peut être reçu par une primitive de réception bloquante `ipcReceive()`. Les IPC asynchrones ne sont pas fiables : les messages peuvent être perdus.

2. *synchrone :* par un protocole d'appel de procédure à distance (RPC), qui bloque l'appelant jusqu'à ce qu'il reçoive une réponse à sa requête. Les primitives correspondantes sont `ipcCall()` pour lancer une requête et `ipcReply()` pour répondre à une requête. Le protocole RPC est fiable, reposant sur une réservation stricte des ressources sous-jacentes.

Beaucoup d'études considèrent les RPC comme un cas représentatif pour la performance des IPC, car ils ont une influence décisive sur la performance de toute application client-serveur, et également sur la crédibilité des micro-noyaux eux-mêmes [75]. En conséquence, nous avons choisi d'optimiser les IPC synchrones (ou RPC) de Chorus.

6.1.2 Architecture des IPC du micro-noyau Chorus

Cette section décrit l'architecture générale de l'implémentation des IPC dans la version 3 du micro-noyau Chorus. Nous présentons uniquement les détails nécessaires pour présenter les opportunités d'optimisation discutées dans la section suivante. Une description plus complète de cette architecture IPC peut être trouvée dans [25] et une présentation plus détaillée de l'implémentation peut être trouvée dans [26].

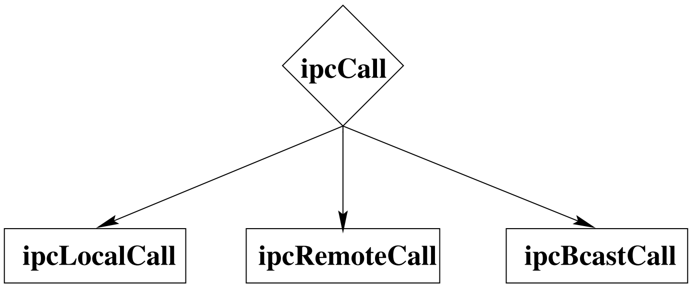


FIG. 6.1 – *Le protocole IPC de* CHORUS

Les IPC de CHORUS permettent d'envoyer des messages à des *ports*. Chaque port est associé à un *acteur*. L'acteur est l'unité d'allocation des ressources en CHORUS. Il définit principalement un espace d'adressage. Les ressources d'un acteur sont partagées par plusieurs fils d'exécution, concurrents ou parallèles. En l'occurrence, les messages adressés à un port peuvent être lus par tous les fils d'exécution de l'acteur associé au port. Il existe plusieurs types de ports : (i) les *ports fixes*, qui restent associés en permanence à un unique acteur ; (ii) les *ports mobiles*, qui peuvent migrer d'un acteur à un autre en cours de leur vie ; et (iii) les *groupes de ports*, qui définissent des ensembles de ports auxquels on peut envoyer un message de manière atomique. Chaque acteur possède un port privilégié, appelé le *port par défaut* de l'acteur.

Le micro-noyau CHORUS r3 est un micro-noyau de troisième génération, qui intègre de nombreux résultats de recherche, et tire les leçons des implémentations antérieures. Ainsi, l'implémentation des IPC dans CHORUS r3 repose sur un cadre dérivé du modèle x-Kernel, que nous avons décrit dans la section 3.1. Cependant, seules quelques-unes des caractéristiques de ce modèle ont été reprises. Par exemple, la technique de décomposition des protocoles dans des micro-protocoles est relativement peu appliquée : il n'existe qu'un nombre restreint de protocoles de base, chacun relativement complexe.

Néanmoins, on retrouve une architecture réseau configurable, par composition de protocoles. On retrouve également la technique de protocoles virtuels permettant d'acheminer les messages de manière dynamique dans le graphe de protocoles.

Par exemple, plusieurs protocoles d'IPC sont fournis : il existe une version générique, `ipcRemoteCall`, qui repose sur des protocoles réseau sous-jacents et peut être utilisée pour le cas local ou distant ; en fait, pour des raisons de performance, les communications locales utilisent une version optimisée, `ipcLocalCall`, qui ne fait plus appel aux protocoles réseau sous-jacents, mais rajoute directement le message dans la file d'attente du port local ; une troisième version est utilisée pour envoyer des messages à un groupe de ports (en *multicast*).

Le choix entre ces versions se fait par le protocole virtuel `ipcCall` (voir figure 6.1) qui choisit le protocole sous-jacent à chaque envoi de message, en fonction du port (ou groupe de ports) de destination.

6.1.2.1 Le protocole virtuel `ipcCall`

La figure 6.2 donne une représentation synthétique et simplifiée du protocole virtuel `ipcCall`. La définition des fonctions auxiliaires les plus importantes est détaillée sous la

```
    ── ipcCall(request, srcPort, dest, reply, delay) ──
I     if(delay) ...                                   // interprète quelques options
      if(reply->flags)...

II    result = ipcSetPathSource()                     // estampille le message
        ── ipcSetPathSource() ──                      // avec le port source

          p = portLiTable.getElement(srcPort)         // recherche le descripteur de port
          p->getUi()                                  // récupère son nom unique
          p->getPi()                                  // et sa clé de protection

      if(result != 0)...                              // vérifie le code de retour

III   ipcSetPatDest()                                 // établie la destination
        ── ipcSetPathDest() ──

          destType = dest->target->uiType()           // établie le type du port (fixe/mobile/groupe)
          localizeFun = ipcLocalizationSwitch[destType]  // recupère la fonction de localisation corresp.

      do {
        pathType = (*localizeFun)()                   // invoque cette fonction de localisation
          ── absPortLocalize() / migPortLocalize() / groupPortLocalize() ──
                                                      // recupère le type de chemin a suivre:
            ...                                       // local/distant/diffusion
IV      ipcProto = ipcCallSwitch[pathType]            // récupère le protocole sous-jacent
        result = (*ipcProto)()                        // et invoque ce protocole
          ── ipcLocalCall() / ipcRemoteCall() / ipcBcastCall() ──

            ...
V     } while(result<0 && isLocalizationError(result))   // vérifie le résultat du protocole
      return result                                   // retourne le résultat
```

FIG. 6.2 – *Le protocole virtuel* `ipcCall`

forme d'une boîte imbriquée, placée à côté de l'appel. Dans le cas des appels indirects de fonctions, la boîte imbriquée détaille la liste des fonctions possiblement appelées. Notons que les définitions des fonctions ne sont pas complètes — elles contiennent principalement les opérations impliquées dans le processus de spécialisation.

L'exécution du protocole virtuel `ipcCall` se compose de cinq étapes. La première étape concerne l'interprétation d'un certain nombre d'options passées en paramètres, tels que le délai maximum utilisé pour attendre une réponse à la requête courante, ou un drapeau qui indique si la requête est interruptible.

Lors de la deuxième étape, la fonction `ipcSetPathSource()` estampille la structure représentant la requête avec des paramètres propres au port expéditeur (identification, protection). Pour retrouver ces informations, le descripteur du port source est recherché dans une table du système nommé `portLiTable`, à partir de l'identificateur local du port `srcPort`. Cet identificateur, local au site courant, est reçu en paramètre par l'appel de système `ipcCall`. Une valeur spéciale de l'identificateur local est réservée pour spécifier comme source le port par défaut de l'acteur courant.

La troisième étape invoque une fonction de localisation pour identifier le site où le port de destination réside. Il existe plusieurs fonctions de localisation selon les différents types de ports : fixes, mobiles et groupes de ports. La fonction de localisation correspondante au port de destination est déterminée dans la fonction `ipcSetPathDest()`, en utilisant une table de fonctions de localisation (`ipcLocalizationSwitch[]`). Ensuite, cette fonction de localisation est invoquée.

Les fonctions de localisation retournent le site de destination (si le port demandé existe) et le chemin à suivre pour atteindre ce site. Quatre types de chemins existent :

`NO_PATH` signifie qu'aucun site accessible ne contient le port de destination ;

`LOCAL_PATH` signifie que le port de destination se trouve sur le site local ;

`REMOTE_PATH` signifie que le port de destination se trouve sur un site distant ;

`BCAST_PATH` signifie que la destination est un groupe de ports en mode diffusion.

Lors de la quatrième étape, le protocole IPC correspondant au type de chemin retourné par la fonction de localisation est appelé, par l'intermédiaire d'une table de protocoles (`ipcCallSwitch[]`).

Finalement, la cinquième étape consiste à vérifier le code de retour du protocole invoqué. En cas d'une erreur de localisation, on revient à la troisième étape (localisation). Cette boucle est exécutée lorsque le port de destination a été localisé sur un site mais a migré entre temps.

Considérons brièvement les protocoles sous-jacents au protocole virtuel `ipcCall`, qui sont appelés lors de la quatrième étape.

6.1.2.2 Le cas local

Le protocole IPC invoqué dans le cas local est optimisé pour le cas où une routine de service est attachée au port de destination. Si aucune routine n'est attachée, on exécute une version générique du protocole, qui commence par créer un descripteur de transaction RPC. Le descripteur est inséré dans la queue de messages du port destination, en attendant que le fil d'exécution serveur soit activé. Si, par contre, une routine de service est présente, on emprunte un chemin optimisé, qui invoque la routine de service de manière immédiate. Pour ce faire, la fonction `msgHandlerCall()` réalise un changement de contexte vers le fil d'exécution serveur, en contournant le processus habituel d'ordonnancement, et en omettant de créer un descripteur de transaction.

6.1.2.3 Le cas distant

Le protocole IPC invoqué dans le cas d'un RPC distant recourt à une pile de protocoles sous-jacents relativement complexe. La complexité est due principalement à la fiabilité du protocole, qui garantit que les messages arrivent à destination indépendamment de la charge du réseau. Lors de l'ouverture d'une *session*, on utilise un sous-protocole de réservation de ressources (notamment des tampons mémoire pour les paquets) sur le site distant, ainsi que sur tous les sites intermédiaires. Une réservation réussie retourne un crédit correspondant au nombre de messages qu'il est possible d'envoyer. La gestion du crédit et les demandes de réservation ultérieures sont gérées de manière transparente par un protocole séparé.

6.1.2.4 Le cas de diffusion

Dans le cas où le port de destination est un groupe de ports, on invoque le protocole de diffusion `ipcBcastCall`. Dans la version r3 du micro-noyau, le RPC en mode diffusion n'est pas permis — il retourne simplement une erreur.

6.2 Opportunités de spécialisation

Comparé à notre expérience antérieure de spécialisation du protocole RPC de Sun décrite dans le chapitre 5, les IPC de CHORUS constituent un plus grand défi, puisque l'implémentation des IPC dans CHORUS est déjà bien optimisée. Un bon nombre d'invariants "classiques" ont déjà été exploités. Il s'agit donc d'aller plus loin que l'approche manuelle, pour obtenir une spécialisation beaucoup plus pointue. Typiquement, les spécialisations manuelles déjà effectuées ont considéré des propriétés "globales" (local ou distant, etc.), et non pas des invariants spécifiques à un client particulier et à son serveur. En fait, ces invariants n'ont pas été considérés dans l'approche manuelle, soit parce qu'ils auraient été trop fastidieux à exploiter, soit parce que certains concernaient des valeurs connues uniquement à l'exécution.

6.2.1 Choix du sous-cas à optimiser

Notre objectif est d'identifier un sous-cas critique en termes de performances, et d'y localiser les opérations "redondantes". Par opération redondante, nous entendons une opération qui est effectuée à chaque RPC, et dont les paramètres sont fixés soit par l'environnement d'exécution soit par un client et/ou un serveur particulier.

L'envoi de messages à un port local est un très bon candidat. Bien que le cas local soit optimisé sous la forme d'un protocole distinct du cas distant, il contient encore des opérations redondantes. Par exemple, le service de localisation n'a pas lieu d'être dans le chemin critique du cas local. Même pour des ports qui peuvent migrer, une approche optimiste peut considérer la localité du port comme un quasi-invariant, et reposer sur une re-spécialisation déclenchée par l'opération de migration, pour préserver la transparence de la localisation.

La performance du cas local est critique pour le comportement de tout système d'exploitation construit au dessus du micro-noyau CHORUS. En effet, la philosophie de modularité des micro-noyaux, consistant à construire un système complet par l'assemblage entre plusieurs modules serveurs, repose directement sur des RPC locaux très rapides. Habituellement dans ce cas, le port du serveur possède une routine de service attachée, pour traiter les requêtes de manière immédiate (ou synchrone).

Par ailleurs, une optimisation même modeste du cas local peut, potentiellement, influer de manière significative sur la performance de beaucoup d'applications. Notamment, devraient en profiter toutes les applications qui utilisent fréquemment des services système fournis par un serveur local.

Nous avons donc décidé de choisir comme chemin critique à optimiser le sous-cas d'un RPC vers un serveur local, avec une routine de service attachée.

Niveau	Invariants	Type
application	port de destination	inv
	port expéditeur	(q)inv
	options	qinv
noyau	descripteurs de ports : • type (fixe/mobile/groupe) • localisation • routine de service	 inv (q)inv qinv
	tables de fonctions	inv

TAB. 6.1 – *Invariants d'un IPC local*

6.2.1.1 Invariants

Les invariants et quasi-invariants correspondants à ce cas de spécialisation sont énumérés dans le tableau 6.1. Ces invariants se partagent en deux classes.

Une première classe d'invariants proviennent de l'application utilisant les IPC. Ainsi, le fait que l'acteur client utilise toujours un même service d'un même serveur implique qu'on peut considérer le port de destination comme un invariant. Le port expéditeur peut être aussi considéré comme invariant pour toutes les applications utilisant le port par défaut. Si le port source n'est pas le port par défaut, on peut toutefois le considérer comme quasi-invariant dans la plupart des applications. De même, la plupart des applications courantes choisissent des valeurs fixées pour les options relatives au protocole, qui correspondent à autant d'invariants.

Une deuxième classe d'invariants portent sur des structures de données du micro-noyau. Tel est le cas par exemple pour les descripteurs de système associés aux ports expéditeur et destinataire : certains champs de ces structures, comme le type du port (fixe/mobile/-groupe), sa localisation, et sa routine de service attachée, sont invariants. Le type d'un port est un invariant car il ne peut pas varier après la création du port. Si le port destinataire est fixe, ce qui correspond à un placement statique du service fourni, sa localisation peut être exploitée comme un invariant; si le port de destination est de type mobile, sa localisation peut toutefois être considéré comme un quasi-invariant, car l'opération de migration est coûteuse en pratique est donc utilisés rarement. Enfin, la routine de service attachée au port destinataire peut être considérée comme un quasi-invariant. En effet, cette routine peut être changée par l'appel de système `svMsgHandler()`, mais la durée de vie d'une routine de service est normalement suffisante pour rentabiliser la spécialisation.

D'autres invariants portant sur des structures de données du noyau sont la table de fonctions de localisation (`ipcLocalizationSwitch[]`) et la table de protocoles IPC sous-jacents (`ipcCallSwitch[]`). Ces tables sont initialisées au moment du lancement du système, et ne sont jamais modifiées ultérieurement. Les informations locales à un site (le numéro du site par exemple) sont aussi initialisées à ce stade et ne changent plus après.

Par ailleurs, dans CHORUS, un fil d'exécution est toujours attaché à un acteur. Cette association ne peut jamais changer pendant l'existence du fil d'exécution, donc son acteur courant peut être considéré comme un invariant. Par conséquent, le port par défaut correspondant à ce fil d'exécution est aussi un invariant.

6.2.1.2 Impact des invariants

Tous ces invariants offrent des opportunités d'optimisation, car ils rendent certains opérations redondantes, opérations qui peuvent être enlevées du chemin critique pour accélérer son exécution. Les opérations redondantes identifiées dans notre cas peuvent être groupées en quatre catégories.

Appels indirects. Les appels indirects (à travers des tables de fonctions invariantes) aux services de localisation et de recherche des ports peuvent être transformés en des appels directs, du fait de la connaissance du type des ports et de la table de services. De manière similaire, l'appel indirect des protocoles IPC sous-jacents peut être transformé en appel direct.

Bien au delà d'économiser un accès à une table de fonctions, la transformation des appels indirects en des appels directs permet de déplier la fonction appelée, ce qui à son tour permet d'effectuer d'autres optimisations. Cet effet de catalyseur est d'ailleurs bien connu dans les travaux d'optimisation des langages à objets visant à éliminer les appels de fonctions virtuelles [42].

Appels de bibliothèque. Les opérations le plus profitable à éliminer par spécialisation sont sans doute les appels à des fonctions de bibliothèque coûteuses du noyau, comme par exemple la recherche des différents descripteurs. Pour qu'un tel appel de bibliothèque puisse être évalué statiquement, deux conditions doivent être satisfaites : (1) tous les arguments de l'appel doivent être statiques, et (2) la fonction appelée ne doit pas effectuer des effets de bord dynamiques. Ainsi, par exemple, la recherche du descripteur du port destinataire peut être évalué complètement pendant la spécialisation, car elle n'implique que des accès en lecture dans les tables du systèmes considérées comme quasi-invariantes. Par contre, les fonctions de bibliothèque visant à acquérir un verrou du noyau ne peuvent pas être évaluées statiquement même si elle ne prennent aucun argument dynamique, car elles sont censées assurer la synchronisation pendant l'exécution. En fait, l'opération de synchronisation est un cas particulier d'effet de bord dynamique. Pour différencier entre les deux types de fonctions de bibliothèque, nous avons utilisé un support spécifique dans Tempo, qui permet de décrire les effets de bord des fonctions externes.

Interprétation d'options. L'interprétation des options invariantes, telles que la permission d'interrompre une requête ou l'option d'attente infinie de la réponse, peut être effectuée lors de la spécialisation, au lieu d'être effectuée à chaque IPC.

Propagation d'erreurs. Une partie de la gestion d'erreurs peut être aussi effectuée pendant la spécialisation, quand le code d'erreur provient d'une opération statique. Par exemple, la recherche statique du port de destination rend possible la vérification statique du descripteur retourné et de la présence d'une routine attachée. De même, le code de retour de différentes étapes d'un `ipcCall()` peut être vérifié statiquement.

Lorsque la condition d'erreur est dynamique, comme c'est le cas de la boucle de relocalisation de la figure 6.2, il faut recourir à une technique spéciale, qui consiste à séparer le chemin critique et le code de gestion des erreurs. Considérons l'exemple simplifié suivant :

```
i=0;
do {
  f(i, ...);
  i++;
} while(erreur);
```

Dans le cas d'une exécution normale (sans erreur), la fonction `f()` recevra comme paramètre `i` la valeur `0`, et pourra être spécialisée ou même évaluée statiquement. Mais comme le corps de la boucle implémente en même temps le cas d'erreur, la valeur de `i` doit être considérée comme dynamique, en général, et par conséquent on perd l'opportunité de spécialiser `f()`.

Une solution à ce problème consiste à séparer le deux cas, en déroulant une fois la boucle de la manière suivante :

```
i=0;
f(i, ...);
i++;
while(erreur) { // fin du chemin critique
  f(i, ...);
  i++;
}
```

Sous cette forme, le chemin critique correspondant à une hypothèse optimiste (pas d'erreurs) pourra être spécialisé par rapport à la valeur statique de `i`. Cette technique, intégré dans Tempo, permet de spécialiser la boucle de re-localisation du protocole `ipcCall`.

6.3 Spécialisation du code

Cette section présente la spécialisation d'une partie du protocole IPC de la figure 6.1, comprenant le code complet du protocole virtuel `ipcCall` et du protocole sous-jacent `ipcLocalCall`. La sous-section suivante explique comment Tempo — notre spécialiseur de programmes C — a pu être utilisé pour spécialiser le code de CHORUS, qui est écrit en C++. Les sous-sections restantes montrent les effets de la spécialisation sur un exemple simplifié, celui du protocole virtuel `ipcCall` de la figure 6.2.

6.3.1 Spécialisation de code C++ avec Tempo

Pour pouvoir spécialiser le code avec Tempo, nous avons utilisé le traducteur de C++ vers C d'AT&T nommé `cfront`. Cependant, un couplage direct `cfront`/Tempo n'a pas été possible à cause d'un problème de taille du code. Le problème, spécifique au langage C++, est que les fichiers d'en-tête (`*.H`) contiennent à la fois des déclarations et du code — les fonctions `inline` définies dans des définitions de classes. On obtient ainsi un code C de l'ordre de 20.000 lignes. Avec nos outils courants, la spécialisation n'est pas faisable sur cette taille de code.

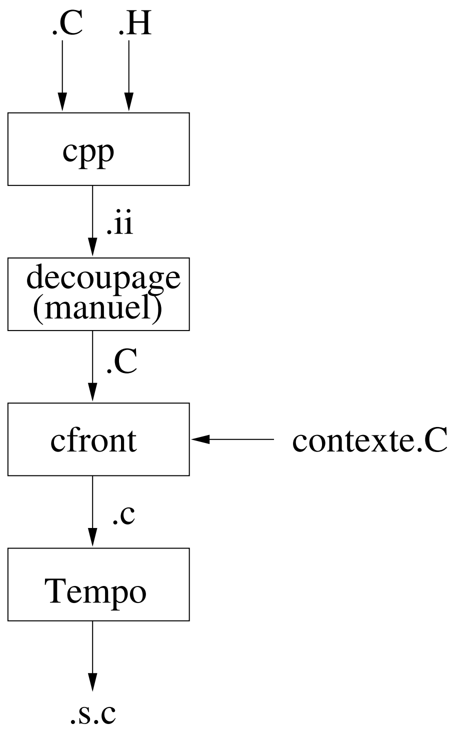


FIG. 6.3 – *Spécialisation de code C++*

Il existe déjà des techniques automatiques de découpage (*slicing*) d'une hiérarchie de classes C++ pour garder uniquement les déclarations de fonctions effectivement utilisées [112]. Mais un tel découpage risque de ne pas résoudre le problème de taille du code, car il n'exploite pas toute l'information existante dans notre cas. En effet, étant donné notre but — la spécialisation —, nous pouvons utiliser un critère de slicing plus fort que la non-utilisation du code : le degré de spécialisation. Ainsi, si une fonction prend en entrée uniquement des valeurs dynamiques, il n'y a pas d'opportunités de spécialisation dans son corps, donc il est inutile d'inclure cette fonction dans le fragment de code à spécialiser. De la même manière, si tous les arguments d'une fonction sont statiques et la fonction n'effectue pas des effets de bord, il est également inutile d'analyser son corps — il sera complètement évalué lors de la spécialisation. Pour les autres cas, intermédiaires entre ces deux cas extrêmes, il s'agit d'estimer l'impact de ce choix sur le degré de spécialisation du programme, et ultimement l'impact sur le gain obtenu par spécialisation. Dans l'état actuel de notre recherche, nous avons procédé à une étape manuelle de découpage de la hiérarchie des classes, avant le passage du code par `cfront` (voir figure 6.3). À terme, cette étape manuelle devra être facilitée par un outil traitant automatiquement les cas extrêmes mentionnés et offrant un choix interactif à l'utilisateur dans les cas plus complexes.

Notons qu'une étape similaire de découpage selon le critère du degré de spécialisation est nécessaire aussi sur du code C. Dans le cas du C, le découpage se réduit cependant à choisir l'ensemble des fonctions à spécialiser, et à inclure ensuite toutes les structures de données utilisées[1]. Dans le cas du C++, il faut procéder à un découpage simultané du code et de la

1. En effet, la spécialisation du protocole RPC de Sun a nécessité aussi une telle étape de découpage.

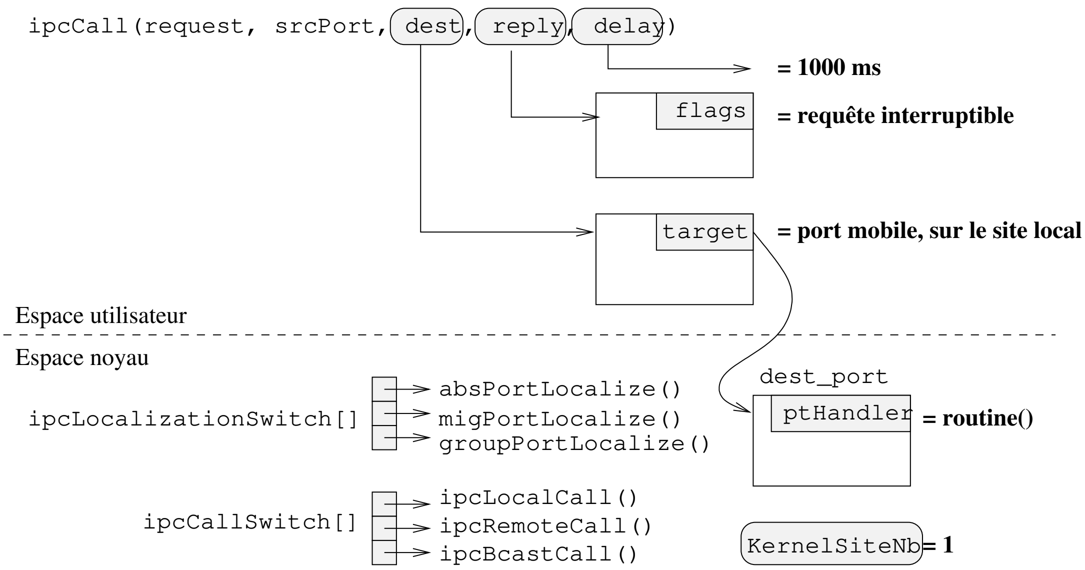


FIG. 6.4 – *Invariants à la compilation*

hiérarchie de classes, à cause de leur interdépendance.

6.3.2 Spécialisation à la compilation

Le protocole IPC a été spécialisé dans un premier temps par rapport à des invariants connus à la compilation. La figure 6.4 montre un ensemble de tels invariants, prenant des valeurs typiques pour le cas de spécialisation considéré, tel que décrit dans la section 6.2.1. Ainsi, on considère un délai d'attente d'une seconde, une requête interruptible vers un port mobile sur le site courant. De plus, on considère la situation d'un système mono-site, d'où le numéro du site étant statiquement connu comme égal à 1.

La figure 6.5 montre l'impact de ces invariants sur le protocole virtuel `ipcCall`. Sont représenté en gris tous les calculs dépendant uniquement des paramètres statiques, et qui seront donc éliminé par spécialisation. Ainsi, les options sont évaluées statiquement ; la sélection de la fonction de localisation est effectuée statiquement, ce qui est symbolisé dans la figure par l'effacement des fonctions non-utilisées. La fonction de localisation sélectionnée est `migPortLocalize()`, et pourra être complètement évaluée dans ce cas. Le gain ainsi obtenu est considérable, car cette fonction de localisation implique une recherche de port dans une table système. Par effet de chaîne, le résultat de la localisation est statique et donc l'autre appel indirect, vers le protocole IPC sous-jacent, est également transformé dans un appel direct à `ipcLocalCall()` ; ce protocole ne peut pas être complètement évalué, car son résultat est dynamique, mais sera simplifié et déplié dans `ipcCall()`. On peut remarquer également la séparation du chemin critique par le déroulement une fois de la boucle de re-localisation.

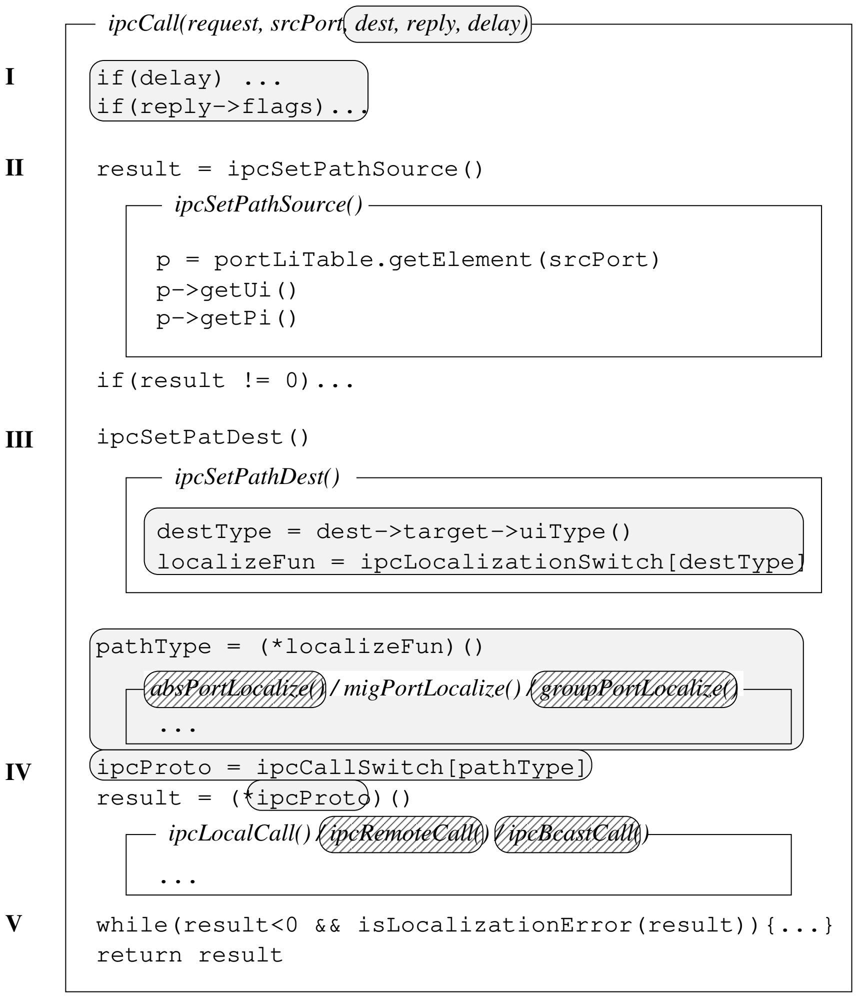


FIG. 6.5 – *Spécialisation à la compilation du protocole IPC*

6.3.3 Spécialisation à l'exécution

La spécialisation à la compilation présentée dans la section précédente n'a pas pu prendre en compte l'invariance du port source (la paramètre `srcPort` de `ipcCall()`), car sa valeur n'est pas connue avant l'exécution. En effet, l'interface de CHORUS impose de spécifier le port source sous la forme d'un identificateur local, et non pas sous la forme d'un identificateur unique, comme pour le port destination. La différence est que les identificateurs uniques peuvent être fournis par l'application (le système vérifie uniquement l'absence de toute collision) ; en revanche, les identificateurs locaux sont fabriqués par le noyau selon un algorithme non-déterministe, et sont donc inconnus avant l'exécution.

Typiquement, la spécialisation à l'exécution est utile dans ce type de situations. En effet, la figure 6.6 montre (en gris foncé) les calculs supplémentaires rendus statiques par la connaissance du paramètre `srcPort`. Ainsi, la procédure `ipcSetPathSource()` peut être simplifiée par l'élimination notamment de la recherche du port source dans la table système `portLiTable`, et de deux autres appels de bibliothèque. Enfin, le résultat de cette fonction devient maintenant statique, et peut être vérifié lors de la spécialisation.

Spécialisation incrémentale. La figure 6.6 montre donc une opportunité de spécialisation incrémentale : une partie des invariants prennent des valeurs qui sont connues avant l'exécution, et peuvent être exploités pour effectuer la spécialisation lors de la compilation ; une autre partie des invariants ne deviennent connus que pendant l'exécution, et nécessitent ainsi de retarder la spécialisation jusqu'à ce stade.

Techniquement, les versions d'IPC spécialisées à l'exécution peuvent être obtenues soit directement à partir de la version générique, soit à partir des versions déjà spécialisées à la compilation, en rendant le processus de spécialisation lui-même incrémental. Cette deuxième stratégie exploite le fait que la spécialisation à la compilation est une transformation source-vers-source : la version spécialisée est donc un programme source qui peut être re-spécialisé. Par rapport à la prémière stratégie, cette stratégie de spécialisation incrémentale offre des avantages de performance, à la fois en temps de génération du code spécialisé qu'en qualité du code produit. En effet, la spécialisation à l'exécution doit être efficace, et donc ne peut pas se permettre d'effectuer le même degré d'optimisation que la spécialisation à la compilation. Par conséquent, un processus de spécialisation incrémental, partant d'une version déjà spécialisée à la compilation, a toutes les chances d'obtenir un meilleur code qu'en spécialisant à l'exécution directement la version générique. Pour ces avantages, nous avons choisi d'utiliser la stratégie incrémentale pour obtenir les IPC spécialisés à l'exécution.

6.4 Intégration du code spécialisé

L'intégration du code IPC spécialisé dans le micro-noyau n'a pas été complètement implémentée au moment de la rédaction du document. Pour effectuer les mesures de performances, la version spécialisée a été incorporée au noyau sous la forme d'un point d'entrée système distinct du point d'entrée IPC d'origine. Ceci implique, entre autres, que la responsabilité d'appeler un des points d'entrée revient à l'application. En fait, la version spécialisée devrait

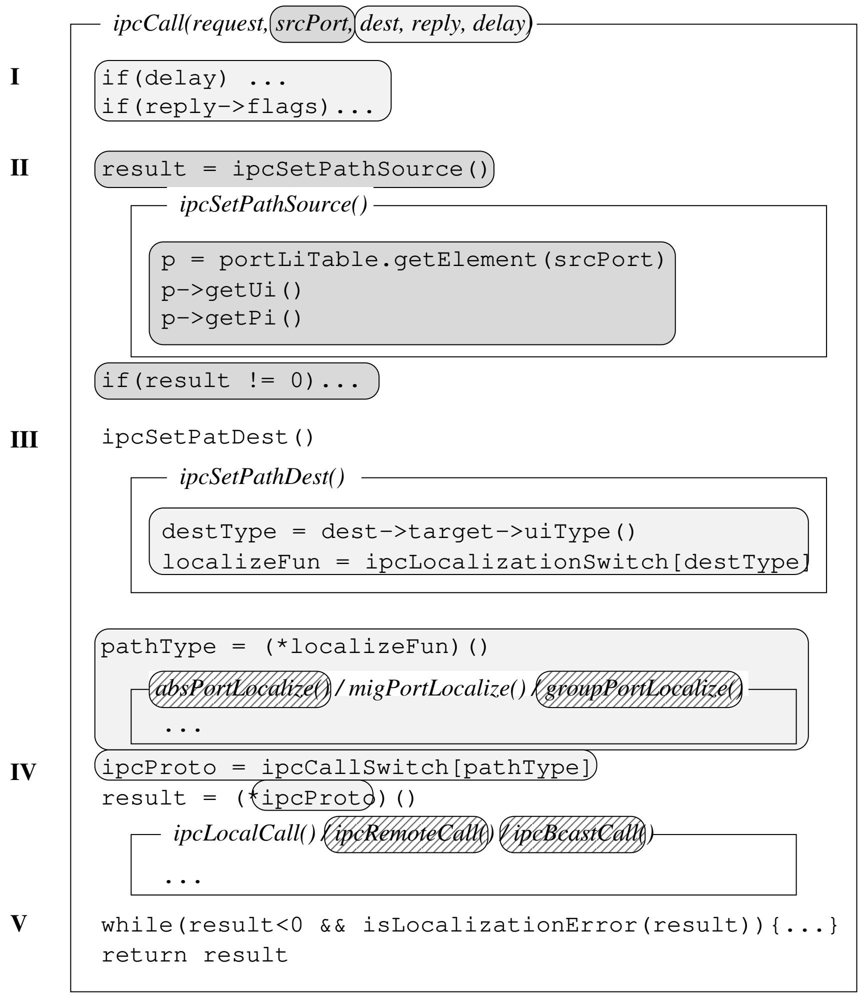


FIG. 6.6 – *Spécialisation à l'exécution du protocole IPC*

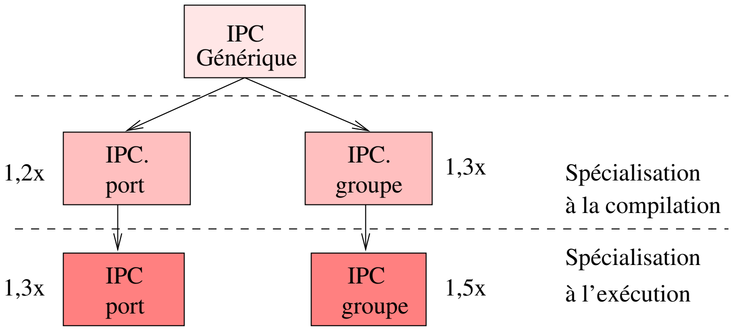


FIG. 6.7 – *Spécialisation incrémentale des IPC de* CHORUS

être invoquée automatiquement lorsque les valeurs des variables statiques concordent avec les valeurs de spécialisation qui ont été utilisées (*i.e.*, celles de la figure 6.4).

Pour finir la mise en œuvre, il faut donc rajouter des fragments de code correspondant aux *gardes* de Synthetix (voir page 22), afin de gérer les versions spécialisées selon la validité des quasi-invariants et des invariants à l'exécution.

6.5 Évaluation de performances

L'évaluation de performances a été effectuée sur une plate-forme à base de Pentium 100MHz, exécutant la version 3.0.1 du système CHORUS/ClassiX. Le code des IPC a été spécialisé pour quatre contextes différents, correspondants à quatre sous-cas d'un appel IPC local avec routine de traitement attachée au port de destination. La figure 6.7 montre ces cas de spécialisation et les facteurs d'accélération correspondants qui ont été mesurés. Les temps d'exécution sont donnés dans le tableau 6.2.

Le côté gauche de la figure 6.7 montre la spécialisation incrémentale décrite dans les sections 6.3.2 et 6.3.3, pour le cas où le port de destination est un port simple. On peut remarquer que la spécialisation à l'exécution apporte un gain supplémentaire significatif.

Le côté droit de la figure 6.7 montre une spécialisation similaire pour le cas où la destination est un groupe de ports. On peut remarquer que les facteurs d'accélération obtenus sont sensiblement plus grands que ceux pour un port simple. Ceci est dû au fait qu'un groupe de ports incorpore un niveau plus grand de généricité qu'un port simple ; cette généricité a été éliminée par spécialisation. Dans le code, cette généricité se traduit par une recherche supplémentaire dans une table système lors du service de localisation : une première recherche est effectuée pour retrouver le descripteur du groupe, et une deuxième recherche pour retrouver un port faisant partie de ce groupe et résidant sur le site local. Les deux recherches ont été éliminées dans le code spécialisé.

Il est à noter qu'on pourrait encore améliorer les facteurs d'accéleration pour tous les quatres cas, avec des modifications mineures en Tempo. En effet, ces versions de code ne contiennent pas tout le degré de dépliage qui existe dans la version générique. La raison en est que la version courante de Tempo n'implémente pas le dépliage à l'exécution. Une future version de Tempo intégrera cette facilité, ce qui permettra d'améliorer encore les résultats de la spécialisation.

Type de port	Générique	Spécialisé à compilation	Accélération à compilation	Spécialisé à l'exécution	Accélération à l'exécution
port	6.1	5.0	**1.2**×	4.7	**1.3**×
groupe	7.1	5.5	**1.3**×	4.7	**1.5**×

TAB. 6.2 – *Mesures de performace sur les IPC génériques et spécialisés (les temps sont données en μs).*

6.6 Conclusion

La spécialisation avec succès des IPC de CHORUS permet de tirer plusieurs conclusions.

Un premier constat est qu'à l'aide des techniques de spécialisation automatique on peut obtenir des gains de performance même sur des sous-systèmes fortement optimisés. L'automatisation permet premièrement une application systématique, pour traiter des cas beaucoup plus spécifiques qu'une approche manuelle. Ainsi, les IPC ont pu être optimisés pour une localisation donnée du client et du serveur, pour un type de port particulier, pour le cas d'un système mono-site, etc. Comme montré dans la section 6.2.1, ce cas spécifique est très courant à l'intérieur d'un micro-noyau, et donc sa spécialisation se justifie parfaitement. Deuxièmement, l'automatisation a permis d'exploiter des invariants difficilement accessibles à une approche manuelle : des constantes d'exécution, comme l'identificateur local du port source, ou des quasi-invariants, comme la localisation d'un port mobile.

Une deuxième conclusion est que l'utilité et l'efficacité de la spécialisation croît lorsque le système offre des abstractions de plus en plus complexes. Cet effet n'était pas évident à prédire avant d'avoir les résultats expérimentaux, car un service complexe implique plus d'interprétation, mais aussi plus de travail effectivement utile. Si le montant d'interprétation rajouté est moindre que le montant de traitement effectif, le gain par spécialisation devrait diminuer. Pourtant, l'exemple de spécialisation des IPC vers un groupe de ports, par rapport à un port simple, constitue un bon exemple où un niveau d'abstraction supplémentaire offert par le système augmente les opportunités de spécialisation.

Une troisième conclusion est que Tempo peut être effectivement utilisé comme moteur de spécialisation dans une plate-forme de spécialisation multi-langages.

Une autre conclusion est que dans certains cas, la séparation du chemin critique par rapport aux code de traitement des exceptions peut être faite de manière automatique. Une telle transformation fonctionnant sur des exemples simples comme la boucle de re-localisation a été incorporée dans Tempo. Dans l'avenir, il serait intéressant d'explorer une généralisation de ce principe de séparation, pour des cas plus complexes de traitement d'exceptions, telles que les constructions syntaxiques *throw* et *catch* de C++. Ceci permettrait d'incorporer dans Tempo des transformations adéquates à leur spécialisation.

Enfin, ce chapitre montre que certaines opérations complémentaires à la spécialisation proprement-dite, et concernant l'intégration et la gestion du code spécialisé, deviennent difficiles à maîtriser manuellement lorsqu'on considère des formes de spécialisation à l'exécution ou par rapport à des quasi-invariants. Ainsi, l'intégration du code spécialisé nécessite l'introduction des fragments de code pour choisir entre la version générique et la (ou les) versions spécialisées. La gestion du code spécialisé nécessite l'introduction d'autres fragments de code pour produire des nouvelles versions spécialisées lorsque des invariants nouveaux apparaissent en cours d'exécution. Pour répondre à cette nouvelle problématique, la dernière partie de cette thèse introduit une approche déclarative à la spécialisation.

Troisième partie

Composants adaptatifs par spécialisation

Chapitre 7

Une approche déclarative à la spécialisation

Les deux chapitres précédents ont montré comment on peut utiliser un évaluateur partiel pendant le développement de composants système, pour obtenir de manière fiable des composants optimisés.

Pour les utilisateurs du système d'exploitation et pour les développeurs d'applications, il faut que l'utilisation de la spécialisation soit transparente, autrement dit, interne au fonctionnement du système. De cette manière, les applications existantes continueront à fonctionner, tout en profitant du gain de performance relatif à la spécialisation. C'est pour cette raison qu'après l'étape de spécialisation proprement-dite, une autre étape est nécessaire, qui consiste à intégrer les versions spécialisées dans le composant d'origine.

Dans le cas du protocole RPC de Sun que nous avons traité dans le chapitre 5, cette étape a été facilement réalisée de manière manuelle, par l'introduction d'un test supplémentaire dans le service, qui distribuait le contrôle vers le code générique ou le code spécialisé, en fonction de la longueur du paquet reçu. Dans le cas du protocole IPC de Chorus présenté dans le chapitre 6, l'étape d'intégration a été déjà plus complexe, tout en restant maîtrisable par une approche manuelle.

Cependant, une approche manuelle à l'étape d'intégration du code spécialisé peut devenir rébarbative quand le contexte d'exécution à tester inclut de nombreuses structures de données, ou quand le nombre de versions spécialisées augmente, surtout lorsqu'elles sont générées à des étapes différentes (compilation, divers stades de l'exécution).

Par ailleurs, même pendant l'étape de production des version optimisées, il existe des manipulations manuelles complémentaires à la spécialisation proprement-dite : le développeur système doit sélectionner et extraire du système d'origine tous les fragments de code à spécialiser, et construire pour chacun un contexte de spécialisation dans le format accepté par son outil. Pour les contextes qui sont complètement connus lors de la compilation, l'utilisateur doit lancer le processus d'évaluation partielle à la compilation, et récupérer le fichier-source spécialisé. Pour les autres contextes de spécialisation, le développeur doit écrire du code qui lancera le processus d'évaluation partielle lors de l'exécution, et récupérera les versions spécialisées sous forme de code binaire.

Certes, les gains obtenus par spécialisation peuvent être suffisamment convainquants pour justifier l'effort investi. Cependant, pour permettre une utilisation vraiment généralisée de la spécialisation, nous proposons une approche de haut niveau, déclarative, pour maîtriser la complexité de tous le détails de production, intégration et gestion du code spécialisé.

Le résultat d'une telle démarche sera une nouvelle version du composant d'origine, offrant la même interface que celui-ci, mais intégrant un comportement *adaptatif par spécialisation* : en réponse à des changements dans son contexte d'utilisation, ce composant réagit en spécialisant sa propre implémentation. Un composant adaptatif par spécialisation offre ainsi en même temps les bénéfices de maintenabilité et réutilisation d'un code générique et l'efficacité d'une implémentation spécialisée.

7.1 Définition d'une approche de haut niveau à la spécialisation

Dans une approche de haut niveau à la spécialisation, l'utilisateur devrait uniquement *spécifier* les besoins de spécialisation de son application ; à partir de cette spécification, toutes les opérations de bas niveau impliquées par le processus de spécialisation devraient être dirigées automatiquement.

Quelques pas on été déjà franchis dans cette direction par des travaux antérieurs. Dans certains évaluateurs partiels, on enrichit le langage source avec des annotations pour délimiter les fragments de code à spécialiser ou pour indiquer comment gérer les versions spécialisées pendant l'exécution [9, 51]. Dans d'autres systèmes, le langage source reste inchangé, mais la sémantique de certaines constructions syntaxiques est modifiée pour définir des étapes de spécialisation potentielle [71]. Dans ce dernier cas, l'utilisateur doit pratiquement réécrire des fragments de son programme pour profiter au mieux de la nouvelle sémantique. Ces stratégies existantes automatisent une partie des opérations de bas niveau, mais ne résolvent que des aspects très ponctuels ; beaucoup de problèmes restent ouverts :

- *Spécification déclarative, séparée.* Si des annotations sont introduites dans le source du programme, sa lisibilité est affectée, surtout lorsqu'on veut spécifier plusieurs contextes de spécialisation différents. Nous prétendons que la spécification des spécialisations potentielles doit être vue comme *rajouter* de l'information à un programme, au lieu de modifier le programme, à l'instar des langages orienté aspects (AOP, pour *Aspect Oriented Programming*) [65] qui permettent de définir séparément la fonctionnalité d'un programme et les différents aspects complémentaires à sa fonctionnalité (distribution, synchronisation, ...).

- *Spécialisation incrémentale.* Plusieurs contextes de spécialisation d'un même composant peuvent correspondre à différentes étapes (la compilation ou différents stades à l'exécution — ouverture d'une session, entrée dans une région critique, ...). Si ces étapes sont imbriquées dans le temps, chacune rajoute ses propres invariants au contexte de spécialisation ; on veut pouvoir exprimer alors le caractère incrémental de la spécialisation [37].

- *Une approche uniforme.* Les progrès dans le domaine de la spécialisation ont augmenté le nombre de paramètres de ce processus. Il est maintenant possible de faire de la spécialisation manuelle ou automatique, à la compilation ou à l'exécution, etc. On doit pouvoir traiter dans le même cadre toutes ces techniques différentes.

- *Flexibilité du support à l'exécution.* La gestion des opérations de bas niveau concernant la spécialisation implique l'existence d'un support à l'exécution. Les paramètres de ce support (quand déclencher la spécialisation, quelles versions spécialisées garder et pour combien de temps, ...) devraient être inférés à partir des spécifications de l'utilisateur ; dans le cas où il existe plusieurs choix, l'utilisateur devrait avoir un moyen de préciser dans la spécification le choix à prendre.

7.1.1 Difficultés avec un langage tel que C

Il est très difficile de développer une approche déclarative à la spécialisation et satisfaisant les exigences ci-dessus pour un langage tel que C. La raison principale de cette difficulté réside dans le manque de structure d'un tel langage, de plusieurs points de vue :

- Maintenir une cohérence entre l'état d'exécution d'un programme et son état de spécialisation repose sur la possibilité de détecter efficacement les changements de certaines variables d'intérêt. La libre utilisation des pointeurs couplée avec l'absence de contrôle d'accès aux structures de données rend cette tâche particulièrement difficile en C.

- Une spécialisation potentielle est normalement associée à un groupe de fonctionnalités regroupées dans un composant logiciel (par exemple un type de données abstrait). L'absence de structuration du code dans des composants logiciels bien définis en C complique une description séparée de la spécialisation.

- Il n'existe pas un mécanisme au niveau du langage qui permette d'exprimer de manière naturelle l'aspect incrémental de la spécialisation.

Pour ces raisons, nous avons choisi de définir notre approche dans le cadre d'un langage structuré à base d'objets.

7.1.2 Survol de notre approche

Nous avons développé une extension pour les langages à objets qui permet d'exprimer la spécialisation d'une manière séparée et déclarative [115, 41]. Dans notre approche, l'utilisateur déclare quels composants doivent être spécialisés et pour quels contextes d'utilisation. Dans le cadre d'un langage à objets, cela revient à spécifier quelles méthodes doivent être spécialisées et par rapport à quelles variables statiques. À partir de ces déclarations, un compilateur dédié détermine comment les méthodes spécialisées seront générées et utilisées. Le résultat de la compilation est une version étendue du composant d'origine, intégrant un comportement adaptatif par spécialisation. Ce comportement concerne le déclenchement de la spécialisation lorsque nécessaire et le remplacement des versions spécialisées d'une manière transparente.

L'unité de déclaration est la *classe de spécialisation*. Elle rajoute de l'information à une classe existante. La relation entre les classes habituelles et les classes de spécialisation est définie par une sorte d'héritage dynamique, reposant sur les *classes prédicatives* développées par Chambers [24]. Une conséquence importante de cette filiation est la possibilité d'effectuer la spécialisation incrémentale à l'aide de l'héritage. C'est-à-dire que la spécialisation d'une classe n'est pas fixe ; elle peut évoluer au fur et à mesure que les valeurs de spécialisation deviennent disponibles.

Configurer un service ou composant système pour un contexte d'utilisation donné revient ainsi à déclarer, séparément du code, un ensemble de classes de spécialisation. Le composant ou service adaptatif par spécialisation sera dérivé automatiquement du composant d'origine, suivant ces déclarations.

Pour valider notre approche, nous l'avons implémentée sous la forme d'un compilateur pour des composants Java étendus avec les déclarations. Le compilateur réalise une transformation source-vers-source, qui produit des composants en Java standard, intégrant un support d'exécution sur mesure pour la spécialisation.

7.1.3 Exemple

Illustrons notre approche en décrivant l'utilisation d'une classe de spécialisation dans le contexte d'un système de fichiers. Supposons que ce système de fichiers définisse une classe `File` qui contient, parmi d'autres variables et méthodes, un mode d'ouverture et un compteur de références, deux méthodes pour lire et écrire dans le fichier, et une méthode pour dupliquer le descripteur de fichier (`dup()`). Quand un fichier est ouvert, une nouvelle instance de `File` est créée et sa variable `mode` est initialisée pour définir les opérations d'accès sur le fichier (lecture et/ou écriture) permises pendant cette session d'ouverture. Le compteur de références est initialisé à `1` jusqu'au partage du fichier par une duplication de son descripteur. Comme l'expliquent Pu *et al.* [98], les opérations de lecture et écriture peuvent être accélérées de manière significative lorsque le fichier n'est pas partagé et que le mode d'accès est connu (soit lecture, écriture ou lecture/écriture). Ceci permet d'éliminer certaines opérations qui sont effectuées normalement à chaque opération de lecture ou d'écriture : le verrouillage n'est pas nécessaire, du fait de l'accès exclusif, et les tests portant sur les permissions d'accès peuvent être exécutés une fois pour toutes quand le fichier est ouvert. Ces opportunités de spécialisation peuvent être déclarées par l'intermédiaire de la classe de spécialisation de la figure 7.1. À partir de ces déclarations, notre compilateur peut déduire que la spécialisation sera effectuée à l'exécution (puisque le mode devient connu uniquement lors de l'ouverture), et modifie l'implémentation de la classe fichier pour déclencher la spécialisation à chaque fois que le compteur de références devient égal à `1`.

Le reste de ce chapitre est organisé comme suit. Dans la section 7.1.4 nous examinons les aspects que doit couvrir une approche déclarative à la spécialisation. La section 7.2 décrit la syntaxe et la sémantique des classes de spécialisation, en donne un exemple complet et en esquisse le schéma de compilation. La section 7.4 discute des aspects particuliers liés à la spécialisation lors de l'exécution. La section 7.5 décrit l'état courant de nos expériences, et les sections 7.6.1 et 7.6.2 présentent quelques optimisations et extensions possibles.

```
specclass ExclAccessFile specializes class File
{
    count == 1; // pas de lecteurs/écrivains concurrents
    mode; // mode d'ouverture connu (toute valeur est la bonne)

    read(); // produit des versions spécialisées :
    write(); // pas de contrôle d'accès, pas de vérouillage

}
```

FIG. 7.1 – *Déclaration d'un fichier adaptatif par spécialisation*

7.1.4 Problématique de la spécialisation déclarative

Une approche déclarative à la spécialisation d'un composant logiciel doit séparer les aspects dénotationnels (*quoi* spécialiser) des aspects opérationnels (*comment* spécialiser). Les déclarations doivent couvrir les aspects dénotationnels ; les aspects opérationnels doivent en être inférés, en principe. Cependant, lorsqu'il existe plusieurs choix pour un aspect opérationnel compatibles avec les spécifications, l'utilisateur devrait pouvoir influer sur ce choix, toujours de manière déclarative.

Examinons, tour à tour, les aspects dénotationnels et opérationnels du processus de spécialisation.

7.1.4.1 Aspects dénotationnels

Déclarer de la spécialisation implique d'identifier les composants logiciels à spécialiser, et de décrire les contextes d'utilisation auxquels ces composants doivent être adaptés. Pour les composants adaptables à plusieurs contextes différents, on doit éventuellement déclarer l'aspect incrémental de la spécialisation.

Composants logiciels. Dans les outils de spécialisation existants, les composants logiciels considérés pour la spécialisation sont : le programme tout entier, une procédure (ou un ensemble de procédures), ou un *bloc* de code. Le choix d'une entité plutôt qu'un autre peut dépendre de la syntaxe du langage ou de la puissance de l'outil de spécialisation — par exemple, si la spécialisation est intra/inter-bloc ou intra/inter-procédurale.

Contexte de spécialisation. Le contexte de spécialisation auquel un composant logiciel doit être spécialisé consiste en un ensemble de prédicats sur l'état du programme (ou sur des parties de cet état), qui sont vrais pour une période de temps donnée. Les spécialiseurs existants considèrent des prédicats consistant d'égalités. Cela revient à spécialiser le composant en fixant des valeurs pour certaines variables. Un prédicat est dit connu à la compilation si il peut être évalué avant l'exécution (sinon, le prédicat est dit connu à l'exécution).

Au delà de leur stade d'évaluation, les prédicats peuvent être classifiés par rapport à leur durée de vie, c'est-à-dire selon la possibilité de changer de valeur pendant l'exécution. On distingue ainsi des prédicats stables, qui ne changent jamais de valeur, et des prédicats instables, qui peuvent être (in)validés par un changement d'état du programme. Ces deux sortes de prédicats correspondent aux notions d'invariant et de quasi-invariant définis dans les chapitres précédents.

Spécialisation incrémentale. Un ensemble de prédicats peuvent ne pas être validés tous en même temps ; habituellement le sous-ensemble de prédicats satisfaits varie dans le temps. Par conséquent, il peut être utile de spécifier une suite de contextes de spécialisation pour un composant donné, pour permettre de raffiner progressivement son état de spécialisation, selon la disponibilité des valeurs.

7.1.4.2 Aspects opérationnels

Les aspects opérationnels concernent le déclenchement de la spécialisation et la gestion des versions spécialisées.

Déclenchement de la spécialisation. Une question importante dans l'application de la spécialisation est quand produire le code spécialisé. Pour des contextes consistant uniquement de prédicats connus à la compilation, la spécialisation peut être appliquée soit à la compilation soit à l'exécution. Pour les autres contextes, la spécialisation ne peut être appliquée qu'à l'exécution. Dans ce dernier cas, une forme de support à l'exécution doit détecter le moment où tous les prédicats du contexte deviennent vrais, déclencher la spécialisation et remplacer le code générique par le code spécialisé.

Assurer la validité de la spécialisation. Pour des contextes dépendant de prédicats instables, le support à l'exécution doit assurer la cohérence entre les versions spécialisées et l'état du programme. Ceci implique de détecter le moment où les prédicats sont invalidés et de remplacer le code spécialisé par une version compatible avec l'état courant.

Réutiliser les spécialisations. La spécialisation à l'exécution nécessite un *cache* qui associe les versions spécialisées avec leurs contextes respectifs, afin de ne pas dupliquer des spécialisations identiques. À cause de limitations d'espace mémoire, on peut vouloir garder uniquement quelques versions fréquemment utilisées. Ainsi, on doit utiliser des stratégies de cache pour gérer les versions spécialisées à l'exécution.

7.2 Classes de spécialisation

Après avoir décrit la problématique d'une approche déclarative à la spécialisation, nous présentons notre proposition, reposant sur le concept de classe de spécialisation.

Les classes de spécialisation définissent séparément le comportement adaptatif par spécialisation des classes existantes. Une classe de spécialisation :

- définit un contexte de spécialisation,
- indique quels composants logiciels doivent être spécialisés pour ce contexte, et

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

FIG. 7.2 – *La syntaxe des classes de spécialisation*

– choisit éventuellement quelques options pour la génération du support à l'exécution.

Nous choisissons comme grain d'un composant spécialisable un sous-ensemble des méthodes d'une classe (standard). Chaque classe de spécialisation est rattachée à une classe du programme d'origine. Plusieurs classes de spécialisation peuvent être rattachées à une même classe, révélant ainsi plusieurs opportunités de spécialisation. Si ces opportunités définissent une suite de stades de spécialisation incrémentale, les classes de spécialisation peuvent être définies par extension pas à pas, au lieu d'être toutes définies à partir de zéro.

La syntaxe des classes de spécialisation est donnée en figure 7.2. La syntaxe est définie comme une extension de la grammaire Java : les non-terminaux qui n'ont pas de définition (`identifier`, `integer`, `class_name`, `variable_name`, `method_name`, et `method_decl`) sont ceux de Java.

7.2.1 Sémantique des classes de spécialisation

Dans cette section, nous décrivons de manière informelle la sémantique des classes de spécialisation. Le schéma de compilation d'un programme Java étendu avec des classes de spécialisation vers un programme Java standard est discuté dans la section 7.3.

En Java, il existe déjà deux relations d'héritage. Premièrement, une classe (ou une interface) peut étendre une autre classe (ou interface, respectivement), par la relation `extends`. Deuxièmement, une classe peut implémenter une interface, par la relation `implements`. Nous avons gardé le même esprit, en rajoutant un troisième type d'héritage, qui attache une classe de spécialisation à une classe habituelle par la relation `specializes`.

L'ensemble des classes de spécialisation rattachées à une classe C forme une hiérarchie d'états de spécialisation possibles. La classe C est dite *classe racine*. Toute classe de spécialisation de l'ensemble soit spécialise directement la classe C, soit étend une autre classe de spécialisation de l'ensemble.

Les champs (variables ou méthodes) figurant dans une classe de spécialisation doivent exister dans la classe racine. Autrement dit, une classe de spécialisation ne peut rajouter des nouveaux champs à une classe ; les classes de spécialisation ne sont pas censées rajouter de la fonctionnalité à une classe habituelle, mais plutôt l'adapter à un contexte particulier. Pour définir ce contexte, une classe de spécialisation rajoute des contraintes sur l'état de la classe C, en définissant une liste de prédicats portant sur les variables de cette classe. Un objet de la classe C est considéré être "dans l'état de spécialisation S" lorsque *tous* les prédicats définis par S sont satisfaits. Dans ce cas, les méthodes spécialisées définies par S remplacent les versions génériques définies par C.

Un cas spécial intervient lorsqu'un objet de la classe C satisfait simultanément tous les prédicats dans deux classes de spécialisation différentes S' et S''. En principe, son état de spécialisation peut être choisi de manière arbitraire, car les deux cas sont applicables. Pratiquement, notre compilateur garantit que, si la relation d'héritage définit un ordre entre S' et S'', alors on choisira toujours l'état le plus spécialisé.

Les prédicats peuvent porter soit sur l'état local de l'objet, s'ils utilisent une variable d'instance de type scalaire (c'est-à-dire non-objet), soit sur un état non-local, s'ils utilisent une variable de type objet (appelé aussi type référence). Pour l'instant, nous ne traitons pas des variables de classe, ni des variables d'instance de type tableau[1].

Un prédicat portant sur une variable scalaire peut être soit connu à la compilation, soit connu à l'exécution. La syntaxe pour les prédicats connus à la compilation est "*nom_variable*`==`*valeur*`;`", où la valeur est une constante connue à la compilation. La syntaxe pour les prédicats connus à l'exécution est simplement "*nom_variable*`;`" sans aucune valeur pour la variable. Ce prédicat est satisfait pour toute valeur, et reste valide jusqu'à une nouvelle affectation de la variable.

Un prédicat sur une variable d'un type objet `class` C_1 contraint cet objet à être dans un état de spécialisation particulier S_1 (où S_1 est une classe de spécialisation attachée à C_1). Le prédicat est écrit "S_1 *variable_name*`;`", comme une déclaration de type de Java. En effet, on peut considérer que le type de l'objet a été redéfini comme un type plus précis.

Nous avons fait délibérément le choix de restreindre les prédicats à ces formes simples, qui sont directement utilisables par un évaluateur partiel. Cette restriction permet d'obtenir la version spécialisée de manière automatique, à partir de la version générique, lorsque la classe de spécialisation ne fournit aucun corps spécialisé. Toutefois, la classe de spécialisation peut spécifier un corps pour la version spécialisée, pour le cas où celui-ci a été obtenu par spécialisation manuelle.

Il est à noter qu'il n'y a pas de différence, au niveau syntaxique, entre les prédicats stables et instables. En fait, dans un langage à objets comme Java il est simple de déterminer statiquement quels prédicats ne sont jamais invalidés — il suffit de vérifier que les variables impliquées ne sont pas affectées en dehors des constructeurs.

Les constructions syntaxiques décrites jusque là permettent de spécifier les contextes de spécialisation. Il existe encore quelques constructions syntaxique par lesquelles l'utilisateur peut influer sur le côté opérationnel du processus de spécialisation (les mots-clés `runtime` et `cached`). Ces constructions concernent uniquement la spécialisation à l'exécution et seront

1. Le programme Java d'origine peut contenir tous ces types de variables ; la seule restriction est que ces variables ne peuvent pas être impliquées dans des prédicats.

discutées dans la section 7.4.

7.2.2 Construction d'un système de fichiers adaptatif par spécialisation

Dans une étude antérieure [98], Pu *et al.* ont justifié de manière pratique le besoin de spécialisation dans un système de fichiers, pour des raisons de performances. Cette expérience visait le système de fichiers de HP-UX. Un nombre de prédicats stables et instables ont été identifiés, qui ont permis une spécialisation manuelle avec des résultats très convainquants. L'intégration des versions spécialisées était assurée par des fragments de code appelées *gardes*, écrites et insérées manuellement dans le code. Les gardes implémentaient une forme d'adaptativité par spécialisation du système de fichiers, en installant ou dés-installant les versions spécialisées en fonction des changements de prédicats.

L'exemple que nous présentons ci-dessous est un programme Java directement inspiré de l'expérience sur HP-UX. Nous définissons les classes de spécialisation pour les objets principaux, et nous montrons comment elles rendent le système de fichiers adaptatif par spécialisation de manière automatique.

Les concepts-clé d'un système de fichiers à la Unix sont le *fichier* et l'*i-node* (pour simplifier, on ignore les v-nodes). Deux structures de données correspondent à ces entités : le descripteur de fichier et le descripteur d'i-node. En fait, la concept de fichier dans Unix est une abstraction pour tout flot de caractères. Il couvre beaucoup plus que les fichiers ordinaires sur disque : les sockets, les périphériques, les tubes (*pipes*), etc. Le type précis du fichier est stocké dans le descripteur d'i-node, qui est référencé dans le descripteur de fichier.

Il existe un descripteur d'i-node pour chaque périphérique, fichier, disque ou tube. Le descripteur d'i-node contient (voir figure 7.3), en plus de l'information de type, des droits d'accès. L'information de type (dont une partie en est le drapeau `pipe` et une autre partie est contenue dans le masque `mode`) ne change en fait jamais, une fois que l'i-node est créé. Les permissions d'accès (qui sont aussi contenues dans le masque `mode`) peuvent être changées via la méthode `chmod()`. Les méthodes les plus importantes d'un i-node sont les méthodes de lecture et d'écriture (`read()` et `write()`)[2]. La méthode de lecture est générique dans la mesure où elle contient de la fonctionnalité pour tous les types d'i-nodes.

La figure 7.4 montre une classe de spécialisation qui adapte l'i-node pour un fichier disque, accessible en lecture uniquement. Elle déclare qu'à chaque fois que les variables `pipe` et `mode` ont les valeurs spécifiées, une opération de lecture devrait invoquer non pas la version générique de `read()`, mais une version spécialisée par rapport à ces valeurs. Quand l'état change de nouveau (c'est-à-dire l'i-node n'est plus en lecture uniquement), la version générique doit être réinstallée.

On pourrait argumenter pour un autre choix de conception du système de fichiers, qui consisterait à créer une hiérarchie statique de l'i-node, avec une classe pour chaque type de fichier ; cela permettrait de séparer les fonctionnalités des différents types dans des méthodes surchargées distinctes, efficaces pour chaque cas. En pratique, ce choix de conception est rarement adopté, parce que les différentes fonctionnalités sont complexes et intimement

2. Pour la concision, la méthode `write()` est omise dans la figure.

```
public class Inode extends Object {
  short mode; // bits 'rwx' + périphérique/fichier
  boolean pipe; // Vrai pour un pipe
  // ...

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

  // change le mode d'accès (les bits 'rwx'):
  public void chmod(short mod) {
    // ... différents tests
    mode = (mode & ~RWX_BIT) | mod;
  }

  // lecture d'un octet:
  public int read() {
      // Version générique:
      // fichier, tuyau, device, ...
      // ... vérifie si accesible en lecture
  }
};
```

FIG. 7.3 – *La définition d'origine de l'i-node*

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

  read(); /* produit une version simplifiée:
                 fichier disque, read-only */
}
```

FIG. 7.4 – *Une classe de spécialisation pour l'i-node*

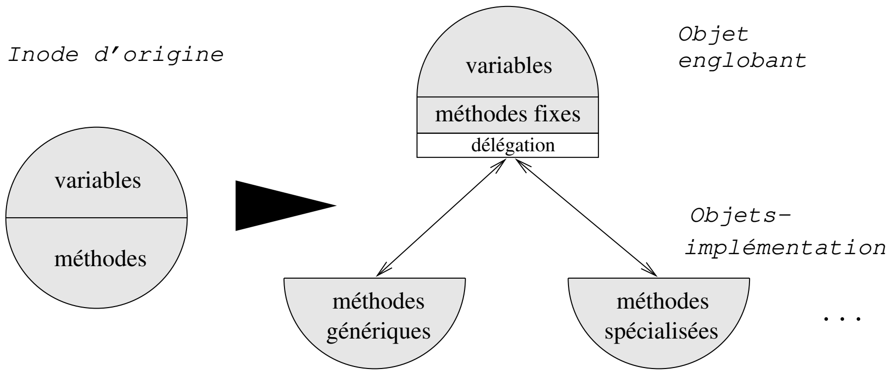


FIG. 7.5 – *L'i-node compilé*

reliées ; une telle séparation conduirait à une multitude de versions de la même méthode, écrites manuellement, et difficiles à maintenir d'une façon cohérente. La spécialisation est un meilleur choix dans ce cas, parce qu'elle permet de dériver les fonctionnalités particulières à partir d'une seule version générique. De plus, par spécialisation on peut éliminer des tests qu'il n'est pas possible d'encoder dans une hiérarchie statique, tels que les vérifications sur le mode d'accès au fichier, qui sont faites normalement à chaque lecture ou écriture.

7.3 Compilation des classes de spécialisation

Le schéma de compilation prend en entrée une classe Java et l'ensemble de classes de spécialisation associées. Elle produit une nouvelle définition pour la classe Java qui intègre le comportement adaptatif par spécialisation. Nous détaillons le processus de compilation sur l'exemple de l'i-node. L'algorithme complet de compilation est donnée en Annexe A.

Comme convention de nommage, les champs nouveaux (variables ou méthodes) introduits par le compilateur sont tous préfixés par '`sc`'.

Le compilateur éclate la fonctionnalité de l'i-node dans plusieurs objets (voir figure 7.5). Un *objet englobant* (montré en figure 7.6) reproduit la fonctionnalité de l'i-node d'origine, sauf la partie "spécialisable", qui est déléguée à un *objet-implémentation*. Dans notre cas, deux objets-implémentation sont créés : un générique (montré en figure 7.7) et un spécialisé (montré en figure 7.8) correspondant à la classe de spécialisation `ReadFileInode` (voir figure 7.4).

Un tel objet, qui peut changer dynamiquement d'implémentation est abstrait par l'interface `Mutable` de la figure 7.9. L'i-node compilé est enrichi pour implémenter cette interface. Il contient une nouvelle variable, nommée `scImpl`, qui réfère à l'objet-implémentation courant. La méthode spécialisable `read()` a été remplacée par une méthode de délégation, qui fait suivre tout appel vers l'objet-implémentation courant. Il est à noter que l'objet-implémentation doit exécuter les méthodes spécialisable comme si elles étaient exécutées par l'i-node d'origine. Pour ce faire, l'objet-implémentation maintient un pointeur (nommé `scEncl`) vers l'objet englobant. Toute auto-référence dans la méthode d'origine (via `this`) est substitué par une référence à l'objet englobant (via `scEncl`).

Afin de détecter des changements d'état qui pourraient affecter l'implémentation cou-

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

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

  // réécrit 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 & passe à celle-ci
    if (mode == (I_REGULAR | R_BIT) &&
        !pipe)
      scSwitchToImpl("ReadFileInode");
    else
      scSwitchToImpl("Inode");
    // propage la notification vers les clients:
    scClients.scNotify();
  }
  protected void scSwitchToImpl(String spec)
    {...}
  public void scAttach(Mutable m)
    {...} // rajoute client
  public void scDetach(Mutable m)
    {...} // efface client
  public boolean scIsA(String spec)
    {...} // inspecte l'état

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

FIG. 7.6 – *L'objet englobant de l'i-node*

```
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. 7.7 – *L'implémentation générique de l'i-node*

```
import sclib.*;

class ReadFileInodeImpl extends InodeImpl {
  // redéfinie certaines gardes:
  // (ici aucune)

  // redéfinie certaines méthodes spécialisables:
  int read() {
    // ... version simplifiée:
    // fichier disque, read-only
  }
};
```

FIG. 7.8 – *L'implémentation spécialisée de l'i-node*

```
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. 7.9 – *L'interface* `Mutable`

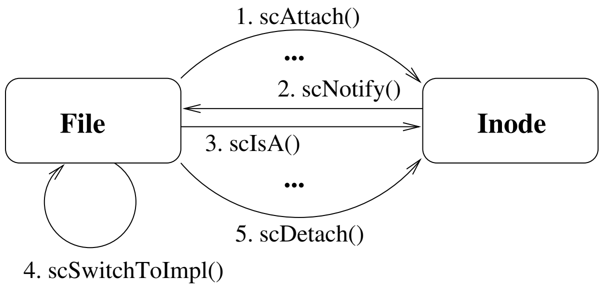


FIG. 7.10 – *Le protocole entre le fichier et l'i-node*

rante, les affectations à la variable `mode` sont gardées : toute affectation de cette variable est remplacée par un appel à une nouvelle méthode appelée `scSet_mode()`. Après exécuter l'affectation, cette méthode vérifie si la nouvelle valeur invalide les méthodes spécialisées. Si c'est le cas, la méthode `scNotify()` est invoquée pour déterminer le nouvel état de spécialisation. Plus précisément, elle procède en considérant tour à tour chaque classe de spécialisation attachée à l'i-node, et vérifie si tous ses prédicats sont satisfaits. Lorsqu'une telle classe de spécialisation a été trouvée (à défaut, la classe générique est prise), l'implémentation courante est mise à jour en appelant la méthode privée `scSwitchToImpl()`.

Pour pouvoir garder toutes les affectations aux variables instables, nous faisons l'hypothèse que ces variables ne sont affectées que dans les méthodes de leur classe, et jamais dans des méthodes d'autres classes, même si elles sont déclarées comme accessibles publiquement. Cela revient à supposer que le programme d'origine respecte le principe d'encapsulation des variables. Dans cette optique, les variables publiques ne sont que lues directement par les méthodes d'autres classes, ou affectées indirectement, par l'intermédiaire des méthodes de la classe.

Le passage à un nouvel état de spécialisation peut changer l'ensemble des variables gardées, car la nouvelle classe peut dépendre de prédicats portant sur d'autres variables. Remarquons aussi que les variables impliquées dans des prédicats stables (par exemple, la variable `pipe`) n'ont pas été gardées.

Quand l'implémentation courante est changée, la méthode `scSwitchToImpl()` produit une nouvelle instance de `InodeImpl` ou de `ReadFileInodeImpl`. Dans notre exemple, toutes les méthodes spécialisées sont produites à la compilation, mais en général, la création d'une nouvelle implémentation peut impliquer l'invocation au vol du spécialiseur à l'exécution.

Les affectations dans le constructeur ne sont pas gardées, car l'état n'a pas encore été complètement initialisé. Les constructeurs sont seulement étendus pour appeler la méthode `scNotify()`, juste avant de terminer. En effet, à ce stade l'état est complètement défini et les différents prédicats peuvent être évalués pour choisir l'implémentation courante.

7.3.1 Extension de l'exemple

Nous présentons maintenant une forme plus complète de l'exemple du système de fichiers adaptatif par spécialisation, où sont spécialisés à la fois l'i-node et le fichier. Cette version montre comment deux spécialisations peuvent être composées, en définissant un prédicat sur l'état non-local, et en préservant sa validité à l'exécution. Elle illustre aussi l'utilisation des prédicats connus à l'exécution et de la spécialisation incrémentale.

La définition d'origine du descripteur de fichier est donnée dans la figure 7.11. Un descripteur de fichier est créé à chaque fois qu'un fichier est ouvert. Le descripteur contient une référence vers l'i-node correspondant, un mode d'ouverture (qui ne peut pas être plus permissive que le mode d'accès défini par l'i-node), la position courante dans le fichier, et un compteur de références indiquant combien de processus utilisent ce fichier. Les variables `ino` et `mode` ne changent jamais après l'ouverture du fichier. La variable `count` change uniquement lorsque le fichier est dupliqué (via la méthode `dup()`). En pratique, cette opération est exécutée très rarement. La position dans le fichier est incrémentée à chaque accès.

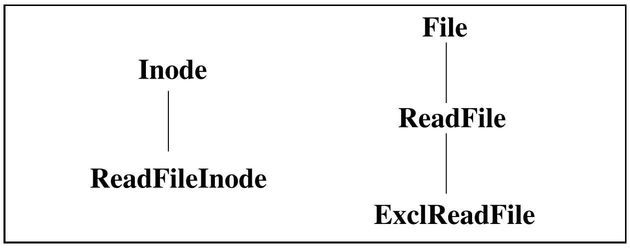
La classe de spécialisation de l'i-node est celle présentée auparavant (voir figure 7.12). Pour le fichier, deux classes de spécialisation sont définis. Un premier stade (correspondant à la classe de spécialisation `ReadFile`) définit un fichier ouvert sur disque. Le prédicat sur le mode d'ouverture est un prédicat connu à l'exécution — il est donc satisfait par n'importe quelle valeur. En effet, toute valeur de ce champ génère de la spécialisation utile. La variable `ino` est redéfinie comme ayant le type `ReadFileInode`. Ainsi, l'i-node correspondant est forcé à se trouver dans l'état de spécialisation `ReadFileInode`. À travers ce prédicat sur l'état non-local, la spécialisation procède de manière inter-procédurale, c'est-à-dire non seulement la méthode de lecture du fichier est spécialisée, mais aussi la méthode de lecture de l'i-node imbriqué. Dans notre exemple, la version spécialisée élimine les tests d'accès et déplie l'appel vers la méthode de lecture de l'i-node (spécialisée elle aussi).

Le deuxième stade de spécialisation d'un fichier rajoute une contrainte supplémentaire : le fichier doit être accédé exclusivement par un seul processus. Dans ce cas, la méthode de lecture peut être encore simplifiée, car les problèmes d'accès concurrents sur le descripteur de fichier peuvent être négligés. Ainsi, la branche avec verrouillage (via l'instruction `synchronized`) est enlevée dans le code spécialisé.

Pour garder la cohérence entre les états de spécialisation de deux objets interdépendants (comme le fichier et l'i-node), il existe un protocole simple par l'intermédiaire de l'interface `Mutable` (voir figure 7.9). Tout objet mutable, tel que l'i-node maintient une liste de “clients”, c'est-à-dire d'autres objets mutables qui dépendent de son état de spécialisation. Lorsqu'un fichier est connecté à un i-node, par une affectation à sa variable `ino`, le fichier se déclare comme un client de l'objet i-node, en invoquant la méthode `ino.scAttach()` (voir figure 7.10). Cette méthode enregistre l'objet fichier dans sa liste de clients (variable `scClients`). Par la suite, lorsque l'objet i-node change d'état de spécialisation, l'objet fichier en sera averti (via la méthode `scNotify()`). La méthode `scNotify()` examine l'état du fichier, y compris l'état de son i-node — en appelant la méthode `ino.scIsA()`. La méthode booléenne `scIsA("nom_etat")` permet à un client de vérifier si un objet mutable a atteint au moins l'état de spécialisation spécifié. Finalement, quand un fichier est fermé, il invoque `ino.scDetach()`, qui enlève ce fichier de la liste de clients de son i-node.

```
public class File extends Object {
   Inode ino; // référence au descripteur d'inode
   short mode; // mode d'ouverture ("r" et/ou
"w")
   long pos; // position courante dans le fichier
   int count; // Nombre des processus partageant
ce fichier

   public File(short mod, Inode inod) { //
constructeur
      mode = mod;
      ino = inod;
      count = 1;
      pos = 0;
   }

   File dup() {
      count++;
      return this;
   }

   public static void main(String argv[]) {
      Inode i = new Inode(Inode.I_REGULAR |
                          Inode.R_BIT |
Inode.W_BIT,
                          false);
      File f = new File(R_BIT, i);

      i.chmod(Inode.R_BIT);
      f.dup();
      i.chmod(Inode.R_BIT | Inode.W_BIT);
   }

   public int read() {
      if((mode & R_BIT) == 0)
         return new FileAccessError();
      if(count > 1)
         synchronized(this) { // vérouille le
descripteur
            pos += 4; // Avance d'un entier
         }
      else pos +=4; // évite le verrou autant que
possible
      return ino.read();
   }
};
```

FIG. 7.11 – *La définition d'origine du fichier avec le programme principal*




```
// Hiérarchie attachée aux i-nodes

specclass ReadFileInode specializes class Inode {
   mode == I_REGULAR | R_BIT;
   pipe == false;

   read(); /* produit une version simplifiée :
              fichier sur disque, en lecture
uniquement */
}


// Hiérarchie attachée aux fichiers:

specclass ReadFile specializes class File {
   mode; /* toute valeur est la bonne */
   ReadFileInode ino; /* restriction sur l'état de
l'i-node */

   read(); /* produit une version simplifiée :
             fichier sur disque, mode connu */
}

specclass ExclReadFile extends specclass
ReadFile
          cached Amortization[20] {
   count == 1;

   read(); /* version encore simplifiée :
             plus de verrouillage */
}
```

FIG. 7.12 – *Les classes de spécialisation pour le système de fichiers adaptatif par spécialisation*

Regardons comment l'exemple complet fonctionne, avec un petit programme principal, montré en figure 7.11 (voir la méthode `main()`). Le programme crée un i-node correspondant à un fichier sur disque, avec initialement des permissions de lecture/écriture, et ouvre un fichier sur cet i-node, en mode lecture uniquement. Les constructeurs des deux objets examinent leurs états. L'i-node choisit l'implémentation générique, car les droits d'accès ne sont pas ceux stipulés. Le fichier choisit aussi l'état générique, car son i-node n'est pas dans l'état `ReadFileInode`.

Ensuite, les droits d'accès à l'i-node sont changés à lecture uniquement. Puisque l'affectation à la variable `mode` est gardée, le prédicat sur `mode` est testé. Comme il est validé, l'i-node passe à l'état `ReadFileInode` (et donc change d'implémentation), et notifie l'objet fichier. Le fichier observe que son i-node est maintenant spécialisé et commute à `ExclReadFile`, puisque en plus sa variable `count` est égale à 1.

Finalement, une opération `dup()` appliquée au fichier déclenche une dé-spécialisation jusqu'à l'état `ReadFile`. Quand l'i-node retourne à un mode lecture/écriture, les deux objets retournent aux implémentations génériques : d'abord l'i-node, et puis le fichier, par propagation en arrière.

7.4 Problématique de la spécialisation à l'exécution

Quand une classe de spécialisation définit un prédicat connu à l'exécution (tel que le prédicat sur le mode dans `ReadFile`), l'implémentation spécialisée pour cet état ne peut pas être produite avant l'exécution, car la valeur précise de spécialisation n'est pas encore connue.

Par opposition, les prédicats connus à la compilation peuvent être exploités soit à la compilation soit à l'exécution. D'habitude, la spécialisation à la compilation est choisie dans ce cas, puisqu'elle n'implique aucun surcoût pendant l'exécution. Cependant, la spécialisation à l'exécution peut présenter d'autres avantages. Par exemple, si N classes de spécialisation sont définies pour une classe racine, une spécialisation à la compilation doit produire en avance toutes le N versions, de manière spéculative, alors qu'une spécialisation à l'exécution peut produire uniquement les versions utilisées à un moment donné. Le choix entre les deux formes de spécialisation doit donc être fait comme un compromis entre le temps et l'espace utilisés ; le meilleur choix dépend de chaque application. Notre solution, détaillée ci-dessous, consiste à proposer un choix par défaut, et fournir à l'utilisateur les moyens pour influer sur ce choix de manière toujours déclarative.

Notre compilateur calcule pour chaque classe de spécialisation un attribut nommé *temps de spécialisation*, qui peut être soit "spécialisable à la compilation" soit "spécialisable à l'exécution". Cet attribut est rattaché à la classe de spécialisation tout entière, et est inférée à partir des types de prédicats qui la composent, comme suit. Une classe de spécialisation ne contenant que des prédicats connus à la compilation est considérée spécialisable à la compilation. Autrement, la classe et considérée spécialisable à l'exécution. Néanmoins, l'utilisateur peut forcer une classe basée uniquement sur des prédicats connus à la compilation à être spécialisée à l'exécution en préfixant la déclaration de la classe de spécialisation avec le mot-clé `runtime`. Il n'existe pas de mot-clé pour forcer dans le sens inverse, car les prédicats connus à l'exécution ne peuvent pas être exploités pour une spécialisation à la compilation.

L'attribut “spécialisable à l'exécution” est hérité. C'est-à-dire que toute classe de spécialisation D dérivée d'une classe de spécialisation spécialisable à l'exécution R est elle aussi spécialisable à l'exécution. La raison en est que la classe D hérite de R des prédicats qui sont (ou ont été forcés par l'utilisateur) exploitables à l'exécution.

Dans un même programme peuvent coexister librement la spécialisation à la compilation et à l'exécution. Plus encore, une même classe racine peut avoir des classes de spécialisation avec des temps de spécialisation différents. Par exemple, l'implémentation générique est toujours produite à la compilation, tandis que certains descendants peuvent être générés pendant l'exécution. Dans notre exemple, toutes les implémentations de l'i-node sont produites à la compilation, alors que les deux implémentations spécialisées du fichier sont produites à l'exécution.

7.4.1 Techniques de *cache* pour les versions spécialisées

L'utilisation d'une technique de cache pour les versions spécialisées pendant l'exécution consiste à garder toutes ces versions dans une structure de données de taille bornée. En plus de limiter l'espace utilisé par la spécialisation, cette méthode a deux autre avantages. Le premier est d'éviter la duplication inutile d'une version spécialisée qui pourrait être partagée, à chaque fois que cela est possible. Par exemple, deux fichiers distincts qui sont dans l'état de spécialisation `ReadFile` peuvent partager la même implémentation s'ils ont le même mode d'ouverture.

Le deuxième avantage est de pouvoir garder des versions spécialisées qui ne sont plus utilisées mais qui ont des grandes chances d'être réutilisées dans l'avenir, ce qui permettrait d'économiser le temps de les produire une nouvelle fois. Par exemple, un fichier-répertoire peut être utilisé plusieurs fois ; garder sa version spécialisée même après la fermeture du fichier peut être avantageux.

Différentes applications peuvent avoir besoin de différentes stratégies de cache. Notre compilateur utilise une stratégie par défaut et fournit un nombre de stratégies alternatives. L'utilisateur peut choisir parmi ces alternatives, allouer de nouveaux caches et même définir des nouvelles stratégies. Pour sélectionner une stratégie de cache particulière pour une classe de spécialisation S, la déclaration de S doit inclure le fragment “`cached` *stratégie*`[`*taille*`]` (voir figure 7.12). Ce fragment spécifie qu'un cache de la taille spécifiée sera réservé pour la classe S. Le comportement de cache est hérité, lui aussi, donc ce même cache sera utilisé pour les descendants de S (sauf en cas de redéfinition).

Notre implémentation est extensible (voir figure 7.13). Une interface `Specializer` définit la fonctionnalité d'un spécialiseur (méthode `spec()`). Le spécialiseur à l'exécution implémente cette interface. Un cache est un “filtre” pour le spécialiseur à l'exécution : il satisfait certaines requêtes de spécialisation par lui-même — s'il peut réutiliser une version stockée dans le cache ; dans le cas contraire, il invoque le spécialiseur à l'exécution pour produire la version spécialisée. Dans ce schéma, l'utilisateur peut rajouter ces propres stratégies de cache, en sous-classant la classe abstraite `CachedSpec`.

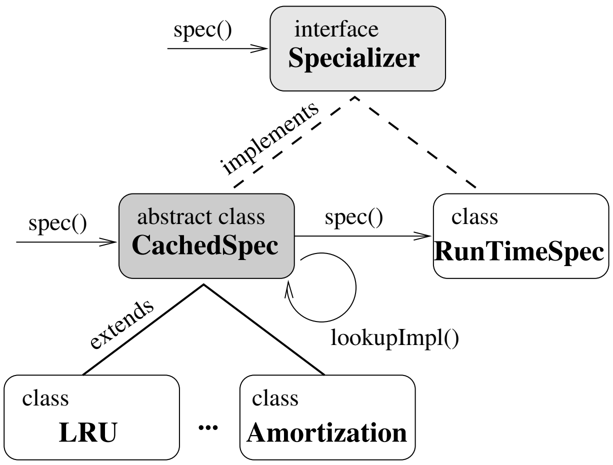


FIG. 7.13 – *La hiérarchie de caches*

7.4.2 Quelques stratégies de cache

On peut utiliser pour le cache de versions spécialisées un certain nombre de stratégies de caches générales (*e.g.*, LRU, associativité multi-voie, ...). Cependant, le fait de gérer des versions spécialisées à l'exécution (au lieu d'adresses mémoire, par exemple) apporte quelque spécificités :

- La spécialisation doit être *amortie*. Ignorer cette contrainte peut conduire à des spécialisation fréquentes, qui n'ont pas le temps de récupérer leur investissement.
- Les classes de spécialisation sont *partiellement ordonnées*, par une relation "plus spécialisée que", où les implémentations les plus spécialisées sont censées apporter le meilleur gain en performances.
- Cette relation d'ordre introduit aussi une relation de *compatibilité* entre les implémentations : une implémentation pour une classe de spécialisation donnée est applicable (probablement avec des performances moindres) à toutes les classes de spécialisation dérivées.

  Une des conséquences en est qu'une requête de spécialisation peut être refusée, en appelant à l'implémentation générique.

En considérant ces spécificités, on peut concevoir des nouvelles stratégies adaptées au domaine :

- *Amortissement :* Considère l'amortissement comme l'aspect le plus important. En conséquence, une implémentation spécialisée n'est jamais jetée du cache tant qu'elle est encore référencée. Cette stratégie conduit à ignorer quelques opportunités de spécialisation par manque d'espace, mais assure le meilleur amortissement pour les requêtes satisfaites.
- *Priorité par profondeur d'abord :* Les classes de spécialisation se voient affectées une priorité selon leur profondeur dans l'arbre d'héritage. Ainsi, dans le cas d'une éviction

du cache, l'implémentation la moins spécialisée est choisie. Cette stratégie favorise l'utilisation des versions les plus spécialisées.

- *Spécialisation approximative :* En réponse à une requête de spécialisation, cette stratégie cherche dans le cache une implémentation compatible avec celle demandée. On recourt à une éviction du cache uniquement si aucune version compatible n'a été trouvée. Cette stratégie encourage le réutilisation des versions, au prix d'une satisfaction approximative des requêtes.

7.4.3 Surcoût du support à l'exécution

La gestion des versions spécialisées à l'exécution impose un surcoût, à la fois en temps et en espace mémoire. Notre support à l'exécution essaye de minimiser le surcoût de temps dans le chemin critique — l'appel d'une méthode spécialisable. Autant de temps que l'état "sensible" ne change pas, tout appel à une méthode spécialisable implique seulement le surcoût d'un appel de méthode supplémentaire, pour déléguer vers l'implémentation courante. C'est uniquement lorsqu'il y a un changement d'état que le support à l'exécution est invoqué pour exécuter différentes tâches : déterminer quelle spécialisation choisir, appeler le spécialiseur à l'exécution (le cas échéant), propager les notifications, et remplacer l'implémentation.

Par ailleurs, le surcoût est différencié parmi les différentes instances d'une même classe, selon l'état de spécialisation de chaque instance. Dans l'exemple du système de fichiers (voir figure 7.12), les affectations à la variable `count` ne sont pas gardées dans les instances génériques, car cette partie de l'état ne peut pas influer sur l'implémentation courante.

7.5 État courant

Nous avons implémenté le schéma de compilation décrit ci-dessus. Notre compilateur est écrit en Java et utilise le générateur de compilateurs JavaCC, avec l'extension JJTree pour la construction des arbres syntaxiques décorés. La compilation accepte la grammaire complète de Java version 1.0.2, étendue avec les classes de spécialisation.

Un évaluateur partiel de Java est en phase de prototypage. Pour pouvoir réutiliser Tempo, notre moteur de spécialisation de programmes C, un traducteur de bytecode Java vers C, nommé Harissa, a été développé [87]. La chaîne complète de spécialisation est montrée en figure 7.14. Le compilateur de classes de spécialisation transforme le composant Java de départ guidé par l'ensemble des classes de spécialisation associées. Ensuite, le résultat est compilé en bytecode en utilisant le compilateur de Sun, `javac`, et après traduit en C par Harissa.

Pour que ce programme C puisse être spécialisé par Tempo, il faut encore un fichier décrivant le contexte de spécialisation correspondant à chaque état de spécialisation. Pour garder la transparence par rapport aux transformations et renommages effectuées par Harissa, le compilateur de classes de spécialisation génère ces contextes sous la forme de code Java — une méthode `scTempoInit()` est rajoutée dans chaque classe-implémentation. Cette méthode effectue les initialisations des invariants avec des valeurs statiques. Pour les invariants

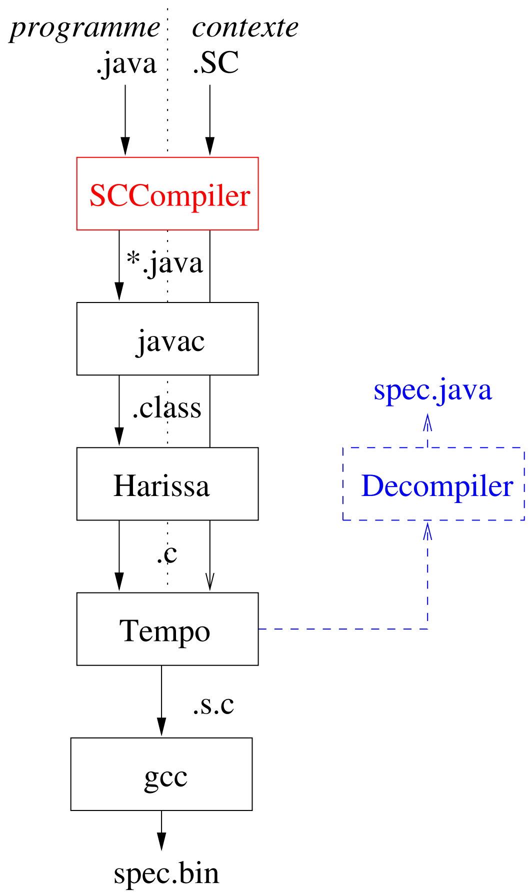


FIG. 7.14 – *La chaîne de spécialisation d'un composant Java*

connus à la compilation, les valeurs sont effectivement celles définies dans les prédicats correspondants. Pour les invariants connus à l'exécution, des valeurs statiques fictives sont utilisées pendant l'analyse ; les valeurs réelles seront fournies lors de la spécialisation à l'exécution.

Après spécialisation, il existe deux choix possibles. Dans le prototype courant, le code C spécialisé est compilé vers du code binaire natif. Pour l'avenir, nous envisageons de fournir aussi la possibilité de reconvertir le code C spécialisé vers du source Java. L'avantage en est de maintenir la portabilité du code spécialisé, avant compilation vers du code natif sur une machine particulière. Par exemple, un éditeur de logiciels pourrait ainsi fournir des composants spécialisés sous forme de source ou de bytecode Java, qui peuvent être compilés par chaque client sur sa propre plate-forme.

7.6 Extensions

Par rapport au schéma de compilation décrit précédemment, il est possible d'apporter plusieurs extensions, soit pour améliorer l'efficacité du code produit, soit pour augmenter l'expressivité des déclarations. Certaines de ces extensions ont déjà été intégrées à notre prototype, mais n'ont pas été décrites dans le schéma de base afin de simplifier sa présentation.

7.6.1 Optimisations

Plusieurs améliorations peuvent être apportées au schéma de compilation afin de produire un code plus efficace (c'est-à-dire qui implique un surcoût réduit pour la spécialisation).

Par exemple, dans les méthodes spécialisables il existe actuellement un surcoût d'une double indirection, pour accéder aux champs de l'instance courante, et qui se retrouvent maintenant dans l'objet englobant. Plus précisément, l'accès à un champ `x` a été réécrit par le compilateur dans l'expression `scEncl.x`. Une première indirection est nécessaire pour récupérer la valeur de `scEncl` (qui est une variable d'instance de l'objet-implémentation), via le pointeur `this`, passé en "paramètre caché" à chaque méthode d'instance ; une deuxième indirection est nécessaire pour accéder le champ `x` de l'objet englobant. En comparaison, le code d'origine effectuait une unique indirection, par l'intermédiaire du pointeur `this`.

Il est possible d'atteindre le même coût en présence des classes de spécialisation. Pour cela, il faudrait passer le pointeur `scEncl` vers l'objet englobant en paramètre caché. Avec ce schéma, il n'est plus nécessaire de stocker `scEncl` dans les objets implémentation, et donc le pointeur `this` ne sera plus jamais utilisé dans les méthodes spécialisables. Par conséquent, ces méthodes pourraient être définies comme des méthodes de classe (`static`) ; l'objet implémentation devient alors une simple table de méthodes virtuelles.

Une autre optimisation possible concerne l'appel des méthodes spécialisables. En utilisant une technique standard d'optimisation des langages à objets, on peut diminuer le surcoût d'un tel appel dans le cas où il existe un nombre restreint d'implémentations (par exemple deux). Dans ce cas, la délégation vers l'objet implémentation peut être faite par une suite de conditionnelles plutôt que par un appel de méthode virtuelle.

Enfin, le surcoût des gardes peut être diminué dans le cas d'une suite d'affectations à des variable instables pendant laquelle on n'appelle aucune méthode spécialisable. Dans ce cas, une seule garde pourrait être mise après toutes les affectations, pour calculer l'effet total du changement d'état provoqué par cette suite.

7.6.2 Extensions de la puissance d'expression

L'expressivité des classes de spécialisation peut être étendue de plusieurs manières.

D'abord, on peut élargir le type de prédicats atomiques utilisés pour déclarer des invariants.

Un premier type d'extensions est de traiter d'autres types de prédicats qui peuvent toujours être exploités pour une spécialisation automatique à l'aide d'un évaluateur partiel. Ainsi, on peut considérer des invariants portant sur des variables de classes ou sur les paramètres formels d'une méthode. Dans ces deux cas, il est nécessaire d'intégrer dans le support à l'exécution un mécanisme de *multi-dispatch*, car lors d'un appel de méthode spécialisable on doit prendre en compte plusieurs fragments d'état afin de sélectionner la version à invoquer. Il existe maintenant des techniques très efficaces de multi-dispatch, qu'on pourrait utiliser directement (voir par exemple [5, 4], ou les très récents travaux sur les méthodes “parasites” [20]). Toujours dans cette catégorie des prédicats exploitables automatiquement, on peut considérer des invariants portant sur des tableaux. Par exemple, un filtre d'images pourrait être spécialisé par rapport à un masque de filtrage particulier avec la déclaration suivante :

```
class Filter {
  int mask[];                                   // masque de filtrage
  ...
  public int[] filter(int[] image) { ... }
  ...
}

specclass BlurFilter specializes class Filter {
  mask == {1,2,1,0,0,0,-1,-2,-1};

  filter();
}
```

Un problème potentiel ici est que les tests d'égalité sur le tableau (utilisés par exemple dans les gardes) n'ont plus un temps d'exécution constant, mais linéaire en fonction de la taille du tableau. Comme ces tests sont déclenchés à chaque modification d'un élément du tableau, l'initialisation du tableau devient quadratique dans la taille du tableau ce qui peut être inacceptable dans beaucoup de cas. La solution que nous avons retenue et implémentée dans notre compilateur est de considérer comme invariants uniquement les tableaux qui ne sont qu'initialisés, et jamais modifiés ensuite. Pour assurer cette discipline, le programme original doit utiliser comme variable `mask` non pas un tableau d'entiers, mais un objet d'un type pré-défini `ArrayOfInt`. Cet objet ne fournit pas de méthode pour affecter un champ individuel du tableau, mais uniquement une méthode pour affecter tout le tableau à la fois. Autrement dit, le tableau est traité comme un objet atomique, et son initialisation ne déclenche qu'un seul test et reste donc d'une complexité linéaire.

Pour les tableaux d'objets, il faudrait que les gardes détectent deux types de modifications : (i) les affectations aux éléments du tableau et (ii) les changements d'état internes aux éléments-objets eux-mêmes. Concernant le premier type de modifications, on peut utiliser la même stratégie de gardes que pour les tableaux de scalaires. Concernant le deuxième type de modifications, on peut utiliser le même mécanisme que pour les invariants sur l'état non-local définis dans la section 7.2.1. Dans l'exemple suivant le tableau `openFiles[]` est déclaré invariant et ses éléments sont contraints à satisfaire la classe de spécialisation `ReadFile`.

```
class Process {
  File stdin, stdout, stderr;                 // les fichiers d'E/S standard
  File openFiles[MAXOPENFILES];               // les autres fichiers ouverts
  ...
}

specclass ReadFile specializes class File {...}
specclass WriteFile specializes class File {...}

specclass ReaderProcess {
  ReadFile stdin;
  WriteFile stdout, stderr;
  ReadFile openFiles[];                       // tous les fichiers ouverts sont en lecture
  ...
}
```

Une deuxième catégorie d'extensions concerne des prédicats plus généraux, qui ne sont pas exploitables par évaluation partielle, mais peuvent être utilisés pour une spécialisation manuelle. Ainsi, à côté des égalités on peut permettre d'autres opérateurs booléens de Java : inégalité, différence, test d'appartenance d'un objet à une classe ou interface (`instanceOf`), ainsi que des conjonctions ou disjonctions entre ces formules atomiques.

Dans l'exemple suivant, une classe tableau est optimisée pour utiliser un autre algorithme de tri dans le cas où il contient un petit nombre d'éléments :

```
class Array {
  Object elements[];
  ...
  void sort() {...}                           // Algorithme de tri, en O(n log n)
}

speccclass SmallArray specializes class Array {
  elements.count <= 20;

  void sort() {...}                           // Algorithme carré,
                                              // plus rapide pour des petits vecteurs.
}
```

Pour surveiller la validité du prédicat, toutes les variables impliquées dans le prédicat seraient sujets à des gardes. Dans un cas encore plus général, on pourrait permettre comme prédicat toute expression Java retournant un booléen et n'effectuant pas d'effets de bord.

7.7 Conclusion

Les classes de spécialisation définissent une approche simple et efficace à la construction des composants adaptatifs par spécialisation. Le comportement de spécialisation est embarqué dans les composants et déclenché automatiquement en réaction aux changements du contexte d'utilisation. Le langage associé est défini comme une extension naturelle des langages à objets et couvre un éventail de possibilités de spécialisation, qui peuvent être composées librement.

Chapitre 8

Conclusion

Nous assistons à une évolution des systèmes d'exploitation vers des formes de plus en plus génériques, incorporant de plus en plus de fonctionnalités différentes. En poursuivant la philosophie traditionnelle selon laquelle le système d'exploitation implémente une machine virtuelle avec un nombre limité de fonctionnalités "universelles", les systèmes deviendront de plus en plus lents et inefficaces. Une solution prometteuse à ce problème semble s'esquisser sous la forme des systèmes d'exploitation extensibles, qui permettent d'incorporer dans le noyau des services personnalisés pour les besoins spécifiques de chaque application.

Traditionnellement, l'écriture de services systèmes optimisés pour un contexte particulier requiert non seulement un niveau très important d'expertise système, mais aussi une connaissance intime de l'implémentation du système en question. Les optimisations ainsi réalisées portent sur différents niveaux : recalibrage des politiques de gestion de ressources, nouveaux algorithmes et structures de données, réécriture des détails de bas niveau pour tirer profit d'un matériel particulier, et spécialisation des algorithmes génériques existants à des cas particuliers critiques en performance.

Ces différentes sortes d'optimisations produisent évidemment un effet de synergie en termes d'amélioration de performances. Pour cette raison, elles sont d'habitude effectuées simultanément. Nous avons étudié séparément la dernière stratégie d'optimisation, par spécialisation. Par opposition aux autres techniques d'optimisation mentionnées, la spécialisation n'est pas liée, en principe, à un système ou à un composant particulier ; elle peut être effectuée sur un programme quelconque, pour peu qu'on y définisse un contexte d'utilisation particulier.

En exploitant cette caractéristique d'universalité de la spécialisation, nous avons cherché à généraliser et à systématiser son utilisation.

8.1 Contributions

La contribution principale de cette thèse est d'avoir proposé une méthodologie d'optimisation de composants système plus accessible à des non-experts en système, reposant sur la spécialisation de programmes. Par rapport à une démarche par spécialisation manuelle, notre approche présente deux niveaux de simplification : la spécialisation est (1) automatisée par l'utilisation de l'évaluation partielle et (2) dirigée par une spécification déclarative.

Spécialisation automatique. L'optimisation des composants système par évaluation partielle apporte plusieurs avantages. Reposant sur un nombre relativement restreint de transformations de programmes élémentaires, l'évaluation partielle est simple et prévisible. Elle évite le risque d'introduction d'erreurs, en préservant la sémantique du composant d'origine. Elle résout le problème de maintenabilité d'un ensemble de versions fonctionnellement équivalentes. Enfin, l'évaluation partielle est une technique d'optimisation portable, pouvant être effectuée sur des systèmes différents, sur des composants différents, voire même sur des langages différents (nous avons vu des exemples portant sur C, C++ et Java).

Nous avons montré, en utilisant notre prototype d'évaluateur partiel, que cette méthodologie est applicable à l'heure actuelle à des systèmes existants, commerciaux, et qu'elle peut apporter des gains significatifs même sur des systèmes déjà optimisés. En effet, nous avons montré sur l'exemple des IPC de CHORUS qu'au delà des optimisations classiques existantes sur un IPC local, on peut gagner jusqu'à un facteur supplémentaire de 1.5 par spécialisation. Ceci prouve bien la complémentarité de la spécialisation avec les autres types d'optimisations.

L'application automatique de la spécialisation encourage une utilisation systématique. Nous avons montré sur un exemple concret comment il est possible de spécialiser un code selon plusieurs contextes différents, à la compilation ou à l'exécution. Nous avons montré l'intérêt pratique de la spécialisation d'un service par rapport à des invariants très fins, valables le temps d'une session client-serveur. Ceci nous a amené au concept d'un code jetable, dépendant d'un ensemble de quasi-invariants.

Approche déclarative. La richesse des opportunités d'optimisation élargit nettement le champ d'application de la spécialisation automatique, mais rend en même temps sa gestion de plus en plus complexe. Nous avons introduit une approche déclarative et non-intrusive à la spécialisation, permettant de maîtriser cette complexité. À l'aide de déclarations de haut niveau, tous les détails de la production, de l'intégration et de la gestion des versions spécialisées peuvent ainsi être cachés.

Le résultat de cette démarche est un composant adaptatif par spécialisation, qui incorpore un comportement d'optimisation réactif à son contexte d'utilisation. Nous avons vu dans le chapitre 7, sur l'exemple d'un système de fichiers, qu'un composant adaptatif par spécialisation peut intégrer de manière transparente des formes de spécialisation incrémentale.

L'approche que nous proposons résout ainsi une bonne partie des problèmes de performance associés aux composants génériques, tout en encourageant leur réutilisabilité et tout en améliorant leur maintenabilité.

8.2 Perspectives

Dans l'avenir immédiat, il serait intéressant d'évaluer l'impact des IPC spécialisés de CHORUS sur des applications réelles. À cette fin, nous envisageons d'effectuer des mesures sur des applications utilisant un serveur superviseur propre, par exemple pour la pagination.

Une autre possibilité serait de mesurer le gain au niveau d'une couche *middleware* utilisant les IPC de manière intense, telle que l'interface objet COOL (Chorus Object-Oriented Layer) [69].

Il est également possible de spécialiser d'autres services largement utilisés du micro-noyau Chorus. Par exemple, dans une étude antérieure [38] nous avons identifié dans la routine de copie mémoire un important degré de généricité, qui pourrait parfaitement se prêter à une spécialisation, statique ou dynamique. D'autres applications possibles pourraient être de spécialiser l'opération de changement de contexte par rapport à un fil d'exécution ou un acteur particulier, ou bien de traiter certains verrous du système comme des quasi-invariants afin de spécialiser les primitives d'exclusion mutuelle.

Plus généralement, il serait intéressant d'évaluer l'efficacité de la spécialisation orientée données (voir page 43) sur des services système, et de la comparer avec la spécialisation par évaluation partielle. Il est très probable que sur des services incorporant beaucoup d'interprétation — telles que les RPC de Sun —, la spécialisation orientée données ne donne pas de résultats aussi bons que l'évaluation partielle, car elle ne réduit pas le flot de contrôle statique. En revanche, sur les IPC de Chorus cette forme de spécialisation pourrait certainement apporter un gain suffisant, en éliminant les opérations de base redondantes, qui ont un poids beaucoup plus important par rapport aux optimisations du flot de contrôle. Il est possible que cette situation arrive aussi bien dans d'autres composants système, tels que par exemple un système de fichiers.

La spécialisation orientée données serait par ailleurs utile à intégrer avec le modèle des classes de spécialisation. Ceci permettrait d'explorer la combinaison entre la spécialisation orientée données et les quasi-invariants ; combinaison qui n'a pas été étudiée auparavant, à notre connaissance.

Sur le plan théorique, il serait intéressant de formaliser le modèle des classes de spécialisation. Les classes de spécialisation constituent clairement un progrès vers la conceptualisation de la génération dynamique de code. On peut mesurer par exemple le chemin parcouru depuis les structures de données exécutables de Synthesis — qui consistaient à inclure dans les données des pointeurs de fonctions nécessaires à leur traversée. Cette technique, utilisée en Synthesis de manière *ad hoc* pour accélérer la traversée de la file des processus par l'ordonnanceur, est subsumée par les classes de spécialisation. En effet, pour obtenir le même résultat il suffit de définir une méthode de traversée associé à un élément de la file et de la spécialiser à l'exécution par rapport à chaque élément-processus, en définissant une classe de spécialisation appropriée. On voit bien sur cet exemple les deux couches langages rajoutée par notre approche : l'automatisation de la génération de code par évaluation partielle et l'intégration dans un modèle à objets. Pourtant, il manque encore une sémantique formelle des classes de spécialisation. Cette formalisation serait utile pour comprendre la complexité des interactions avec la multitude des mécanismes des langages objet : héritage, visibilité de champs, etc. Par ailleurs, la définition d'une sémantique formelle devrait être plus facile que pour les classes prédicatives, car les classes de spécialisation respectent un sous-typage comportemental [78]. Le schéma de compilation pourrait être ensuite prouvé correct par rapport à cette sémantique formelle.

Dans une perspective à plus long terme, nos travaux laissent entrevoir plusieurs directions de recherche.

Les classes de spécialisation décrivent actuellement une forme particulière d'adaptativité, que nous avons appelée adaptativité par spécialisation. On pourrait généraliser les classes de spécialisation pour décrire des systèmes adaptatifs d'une classe plus large. Pour ce faire, on pourrait concevoir un langage spécifique à la description des différentes étape du processus d'adaptation (voir page 20). La facette d'introspection permettrait d'exprimer des informations relatives, par exemple, à la charge ou à la fréquence d'appels de procédures, à l'occupation mémoire ou au temps d'exécution des opérations critiques. La facette de décision contiendrait des prédicats adaptés pour évaluer des seuils de déclenchement relatifs aux informations collectées dans l'étape d'introspection. La facette d'action inclurait des formes d'adaptation comme la spécialisation ou le changement d'implémentation d'une méthode, mais aussi le re-dimensionnement des structures de données par exemple. Un nouveau projet concernant cette forme généralisée des classes de spécialisation, nommées *classes d'adaptation*, va démarrer bientôt dans le groupe Compose. L'approche conserve des qualités essentielles comme celle de non-intrusion par exemple, et enrichit très largement le pouvoir d'expression et les domaines d'application possibles.

Une autre direction de recherche prometteuse serait d'étudier une nouvelle méthodologie de programmation générique, reposant sur les outils de spécialisation existants. Dans cette thèse, l'objectif a été d'optimiser des composants existants. Lorsqu'on conçoit de nouveaux composants spécialisables ou adaptatifs par spécialisation, comment peut-on profiter au mieux des outils de spécialisation pour écrire du code plus générique, plus réutilisable ou simplement plus efficace? Un deuxième projet nouveau à commencer bientôt procédera à une réécriture complète des IPC de Chorus, utilisant autant que possible la spécialisation incrémentale et les classes de spécialisation, afin d'évaluer l'impact de la nouvelle méthodologie sur le coût de développement, sur la maintenabilité ou sur la performance. Cette première étape d'évaluation sera éventuellement poursuivie par une recherche plus approfondie d'un nouveau style de programmation.

Enfin, il existe actuellement une tendance manifeste vers une “componentisation” des logiciels. Différentes formes d'architectures logicielles, de modèles de conception réutilisables (*design patterns*), et de structures d'applications réutilisables (*application frameworks*) émergent et tendent à s'imposer dans de nombreux domaines. Ces nouvelles formes de réutilisation ne vont pas sans intégrer des niveaux supplémentaires de généricité (et donc d'interprétation) dans le code. La conception de composants spécialisables et adaptatifs par spécialisation aura de fait de plus en plus d'importance et d'applications dans l'avenir.

Bibliographie

[1] M. Abbott and L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, Feb. 1993.

[2] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *1986 Summer Usenix Conference*, pages 93–112, 1986.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] E. Amiel, E. Dujardin, and E. Simon. Fast algorithms for compressed multi-method dispatch tables generation. Technical Report 2977, INRIA, Rocquencourt, Sept. 1996.

[5] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA '94 Conference Proceedings*, volume 29:10 of *SIGPLAN Notices*, pages 244–258. ACM, Oct. 1994.

[6] L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[7] P. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[8] T. Anderson. The case for application-specific operating systems. In *the Third Workshop on Workstation Operating Systems*, pages 92–94, 1992.

[9] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In PLDI'96 [97], pages 149–159.

[10] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[11] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In PEPM'94 [95], pages 119–132.

[12] A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.

[13] A. Berlin and R. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In PEPM'94 [95], pages 133–141.

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

[15] B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP95 [107], pages 267–283.

[16] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[17] T. Blackwell. Fast decoding of tagged message formats. In *Fifteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, San Francisco, CA, Mar. 1996.

[18] G. Bochmann. Usage of protocol development tools: The result of a survey. In *Seventh International Conference on Protocol Specification, Testing and Verification*, Zurich, Switzerland, May 1987.

[19] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3^{rd} European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 1990.

[20] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for java. In OOPSLA97 [93], pages 66–76.

[21] 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, Oct. 1995. IEEE Computer Society Press.

[22] 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, Sept. 1993.

[23] 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, Apr. 1995.

[24] 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, July 1993.

[25] Chorus. Chorus kernel v3 r5 network architecture. Technical Report CS/TR-93-35.2, Chorus Systemes, 1993.

[26] Chorus. Chorus kernel v3 r5 implementation guide. Technical Report CS/TR-94-73.1, Chorus Systemes, 1994.

[27] D. Clark, V. Jacobson, J. Romkey, and M. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

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

[29] C. Consel. A tour of Schism. In PEPM'93 [94], pages 66–77.

[30] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[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, Jan. 1993. ACM Press.

[32] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 1998. To appear.

[33] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Partial evaluation for software engineering. *ACM Computing Surveys, Symposium on Partial Evaluation*, 1998. To appear.

[34] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. 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, Feb. 1996.

[35] C. Consel, G. Muller, and E. Volanschi. Adaptation dynamique de programmes système par spécialisation incrémentale. In *Actes des Séminaires Action Scientifique – Journées Systèmes et Applications Répartis*, pages 117–121. France Telecom, July 1996.

[36] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, Jan. 1996. ACM Press.

[37] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In PEPM'93 [94], pages 44–46. Invited paper.

[38] C. Consel, E. Volanschi, and G. Muller. Adaptation dynamique de programmes système par spécialisation incrémentale. Rapport, CNET/France Telecom, octobre 1998. À paraître.

[39] K. Cooper, M. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), Apr. 1993.

[40] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems*, Annapolis, MD, May 1996.

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

[42] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 30(6), June 1995.

[43] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In USENIX Association, editor, *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 292–306, Berkeley, CA, USA, Winter 1994. USENIX.

[44] R. Draves, B. Bershad, R. Rashid, and R. Dean. Using continuations to implement thread management and communication in operating systems. In SOSP91 [105], pages 122–136. Also Tech report CMU-CS-91-115, Carnegie Mellon University, School of Computer Science.

[45] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In SOSP93 [106], pages 189–202.

[46] D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In SIGCOMM'96 [104], pages 26–30.

[47] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [107], pages 251–266.

[48] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. Technical Report UUCS-95-014, Department of Computer Science, University of Utah, Mar. 1995.

[49] A. Forum. ATM user-network interface specification version 3.0, 1993.

[50] A. Gopal, N. Islam, L. B.-H., and B. Mukherjee. Structuring operating systems using adaptive objects for improving performance. In *Proceedings of IWOOOS'95*, pages 130–133, Lund, Sweden, Aug. 1995.

[51] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In PEPM'97 [96], pages 163–178.

[52] B. Guenter, T. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

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

[54] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–199, Oct. 1993.

[55] L. Hornof. *Static Analyses for the Effective Specialization of Realistic Applications.* PhD thesis, Université de Rennes I, June 1997.

[56] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM'97 [96], pages 63–73.

[57] L. Hornof, J. Noyé, and C. Consel. Accurate partial evaluation of realistic programs via use sensitivity. Research Report 1064, IRISA, Rennes, France, June 1996.

[58] W. Hsieh, M. Kaashoek, and W. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *4th Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993. IEEE Computer Society.

[59] N. Hutchinson, L. Peterson, M. Abbott, and S. O'Malley. RPC in the *x*-kernel: evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 91–101, Litchfield Park, AZ, 3–6 December 1989, Dec. 1989. ACM Operating Systems Reviews, 23(5),ACM Press.

[60] ISO. Specification of abstract syntax notation one (ASN.1). ISO standard 8824, 1988.

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

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

[63] N. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[64] D. Keppel, S. Eggers, and R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science, University of Washington, Seattle, WA, 1991.

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

[66] A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: a formal framework for specifying, implementing and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 338–352, Toronto, Ontario, Canada, June 1991. ACM SIGPLAN Notices, 26(6).

[67] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. Meyer and G. Snelting, editors, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.

[68] T. Knoblock and E. Ruf. Data specialization. In PLDI'96 [97], pages 215–225. Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[69] R. Lea, C. Jacquemot, and E. Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–46, sept 1993.

[70] M. Leone and P. Lee. Optimizing ML with run-time code generation. Technical Report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, Dec. 1995.

[71] M. Leone and P. Lee. A declarative approach to run-time code generation. In WCSSS'96 [119], pages 8–17.

[72] R. Levin, E. Cohen, C. W., F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the 5th ACM Symposium on Operating System Principles*, pages 175–188, Austin, Texas, 1975. ACM Operating Systems Reviews, 9(5),ACM Press.

[73] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Sept. 1986.

[74] W. S. Liao, S.-M. Tan, and R. H. Campbell. Fine-grained, dynamic user customization of operating systems. In L.-F. Cabrera and N. Islam, editors, *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems: October 27–28, 1996, Seattle, Washington*, pages ??–??, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[75] J. Liedtke. Improving IPC by kernel design. In SOSP93 [106], pages 175–188.

[76] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report 933, German National Research Center for Information Technology, Nov. 1995.

[77] S. B. Lim. Adaptive caching in a distributed file system. Technical Report 1936, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1995.

[78] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, Nov. 1994.

[79] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letter on Programming Languages and Systems*, 2(1–4):213–232, Mar.–Dec. 1993.

[80] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105, New Haven, CT, USA, Sept. 1991. ACM SIGPLAN Notices, 26(9).

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

[82] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *The Proceedings of the 11th Symposium on Operating System Principles*, Nov. 1987.

[83] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. Technical Report 94–20, Department of Computer Science, The University of Arizona, 1994.

[84] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. In SIGCOMM'96 [104], pages 26–30.

[85] D. Mosberger, L. Peterson, and S. O'Malley. Protocol latency: MIPS and reality. Technical Report 95-02, Department of Computer Science, The University of Arizona, 1995.

[86] G. Muller, R. Marlet, E. 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, May 1998. IEEE Computer Society Press. To appear.

[87] 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, June 1997. Usenix.

[88] G. Muller, E. 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, Mar. 1997.

[89] G. Muller, E. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM'97 [96], pages 116–125.

[90] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. Rapport de recherche 1065, IRISA, Rennes, France, Nov. 1996.

[91] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[92] S. O'Malley, T. Proebsting, and A. Montz. USC: A universal stub compiler. Technical Report TR94-10, University of Arizona, Department of Computer Science, 1994. Also in Proc. Conf. on Communications Archi. Protocols and Applications.

[93] *OOPSLA'97 Conference Proceedings*, Atlanta, USA, Oct. 1997. ACM Press.

[94] *Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[95] *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

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

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

[98] 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 [107], pages 314–324.

[99] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

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

[101] 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 - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, Apr. 1992.

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

[103] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 124–129, Cape Cod, Ma, May 1997. IEEE Computer Society.

[104] *SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford University, CA, Aug. 1996. ACM Press.

[105] *Proceedings of 13th ACM Symposium on Operating Systems Principles*, Asilomar, Pacific Grove, CA, Oct. 1991. ACM Operating Systems Reviews.

[106] *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, Asheville, NC, USA, Dec. 1993. ACM Operating Systems Reviews, 27(5), ACM Press.

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

[108] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 27(4):412–418, July 1981.

[109] A. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experience with the amoeba distributed operating system. *CACM*, 33(12):46–63, Dec. 1990.

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

[111] S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. Rapport de recherche RR-3218, INRIA, Rennes, France, July 1997. Revised version to appear in the proceedings of the USENIX conference on Domain-Specific Languages (DSL'97), Santa Barbara, October 1997.

[112] F. Tip, J.-D. Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 179–197, New York, Oct.6–10 1996. ACM Press.

[113] M. Tokoro. Apertos: An object-oriented, distributed, real-time operating system. *IEEE parallel and distributed technology: systems and applications*, 1(3):84–84, Aug. 1993.

[114] R. van Renesse. Masking the overhead of protocol layering. In SIGCOMM'96 [104].

[115] E. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In OOPSLA97 [93], pages 286–300.

[116] E. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In WCSSS'96 [119], pages 24–28.

[117] E. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Research Report 2903, INRIA, Rennes, France, June 1996.

[118] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In SOSP93 [106], pages 203–216.

[119] *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, Tucson, AZ, USA, Feb. 1996.

[120] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[121] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *1993 Summer Usenix Conference*, Cincinnati, OH, June 1993.

[122] Y. Yokote. The apertos reflective operating system: the concept and its implementation. In *OOPSLA'92 Conference Proceedings*, pages 414–434, New York, USA, Oct. 1992. ACM Press.

[123] W. Zwaenepoel, B. Carter, and K. Bennett. Implementation and performance of Munin. In SOSP91 [105].

Annexe A

L'algorithme de compilation des classes de spécialisation

Entrée: une classe Java X, et un ensemble de N classes de spécialisation qui *spécialisent* la classe X.

Sortie: une nouvelle classe X (la classe englobante), et un ensemble de $N + 1$ classes implémentation pour X.

1. Opérations préliminaires:

   a) Calcule l'ensemble V_{obj} de toutes les variables qui:
      - appartiennent à la classe X, et
      - sont de type référence, et
      - apparaissent dans une classe de spécialisation.

   b) **pour** toute classe de spécialisation S
      - **pour** tout prédicat P dans S
        calcule un temps de spécialisation pour P:
        - **si** P est de la forme "$v == \mathit{const\_expr};$" **alors** P est connu à la compilation
        - **si** P est de la forme "$v;$" **alors** P est connu à l'exécution
        - **si** P est de la forme "$S'\ \ v;$" **alors** P a le temps de spécialization de S'
          NOTE: Les cycles ne sont pas permis.
      - calcule un temps de spécialization pour S:
        si
        - S est annotée `runtime`, ou
        - S étend (`extends`) une classe de spécialisation spécialisable à l'exécution, ou
        - il existe un prédicat spécialisable à l'exécutiondans S

        alors S est spécialisable à l'exécution
        sinon S est spécialisable à la compilation
      - **si** S est spécialisable à la compilation
        alors pour toute méthode m dans S, spécialise m comme suit:
        - appelle le specialiseur à la compilation pour m avec les valeurs de spécialization derivées des prédicats dans S et ses super-classes de spécialisation(les cas échéant)

    - collecte la méthode spécialisée m_S

  sinon pour toute méthode m dans S, prépare la spécialization à l'exécution pour m :

    - appelle l'étape de pré-traitement du spécialiseur pour m avec le contexte de spécialization de S et ses super-classes de spécialisation(le cas échéant)
    - collecte le specialiseur à l'exécution pour m_S

  - calcule des attributs de stabilité :

    pour tout prédicat P dans S

    si la variable v apparaissant dans P n'est jamais affectée en dehors des constructeurs et du finaliseur
    alors P est stable
    sinon P est instable
    NOTE : La variable v est dite aussi stable, respectivement instable. NOTE : La determination des variables stables nécessite une analyse inter-procédurale et sensible au contexte d'appel, parce que le contructeur peut appeler (directement ou indirectement) d'autres méthodes qui affectent les variables d'instance.

c) Construit un arbre de décision, suivant la hiérarchie des classes de spécialisation. Cet arbre de décision sera utilisé pour calculer un nouvel état de spécialisation, en réponse à un changement d'état local. Chaque nœud dans l'arbre correspond à une classe de spécialisation S, et teste une conjonction des prédicats suivants :

  - **pour** chaque prédicat P dans S de la forme "$v == \mathit{const\_expr};$" :
    $v == \mathit{const\_expr}$ (*i.e.*, P lui-même)
  - **pour** chaque prédicat dans S de la forme "S' $v;$" :
    `v.scIsA(S')`
    NOTE : Les prédicats connus à l'exécution sont toujours vérifiés, et donc ne sont pas inclus dans les tests.

2. Construit la classe englobante X :

  a) Rajoute aux parents de X `implements Mutable`.

  b) Copie toutes les variables de X (que ce soit des variables d'instance ou de classe).

  c) Rajoute les nouvelles variables suivantes :

    - `XImpl scImpl;`
    - `List scClients;`

  d) Définie le code d'initialisation suivant, dans une méthode `private void scInit()` :

    - initialise la liste des clients à vide :
      `scClients = null;`
    - calcule l'état initial :
      `this.scNotify();`

  e) Réécris les constructeurs :

    pour chaque constructor de X

    - copie le corps d'origine, *sans garder* les affectations aux variables instables

– rajoute un appel au code d'initialisation : `scInit();`

NOTE : S'il n'existe aucun constructeur, rajoutes-en un, contenant juste `scInit();`.

f) Réécris le finaliseur :

– **pour** chaque variable $v \in V_{obj}$, détache de v :
`if(`v`!=null)` v`.scDetach();`

– copie le corps d'origine

g) Copie les méthodes d'instance, en réécrivant les affectations à des variables instables v comme suit :
v `=` *expr*`;` $\longrightarrow$ `scImpl.scSet_`v`(`*expr*`);`

h) Copie les méthode de classe, d'origine, inchangées.

i) Rajoute le comportement `Mutable` :

– une méthode `public void scNotify()`, qui recalcule l'état de spécialisation courant, en utilisant l'arbre de decision construit préalablement, et commute à l'implémentation correspondante. Avant de retourner, `scNotify()` propage la notification à tous les clients :

```
for tout client c dans scClients
  c.scNotify();
```

– une méthode pour changer l'implémentation courante :

```
protected void scSwitchToImpl(String state)
{
  Specializer s = getTheSpec(state);
    // either CT or RT specializer
  /* cree une nouvelle implementation
     pour l'objet courant,
     specialisee pour l'etat donne : */
  scImpl = s.spec(this, state);
}
```

NOTE : Le spécialiseur à la compilation crée simplement une nouvelle instance d'une implémentation specialisée. Le specialiseur á l'exécution genére une nouvelle classe à partir d'un squelette existant, y instancie les méthodes spécialisées avec des versions spécialisées au vol et instancie cette classe pour obtenir un objet-implémentation.

– trois méthodes pour l'interface avec les clients :

```
public void scAttach(Mutable m)
  // enregistre m comme client
  { scClients.insert(m); }
public void scDetach(Mutable m)
  // d\'etache m
  { scClients.delete(m); }
public boolean scIsA(String state)
  /* verifie si l'object est au moins dans
     un etat de specialisation donne
  { ... }
```

j) **pour** chaque méthode spécialisable m, remplace-la avec une méthode de délégation :
type m`(`*args*`) { return scImpl.`m`(`*args*`) }`

3. Construit les $N+1$ implémentations pour X :

   a) Construit une implémentation générique pour X,
   `class` X`Impl extends Impl`:

   – définie un pointeur an arrière vers l'objet englobant :
   `protected` X `scEncl;`

   – définie un constructeur :
   X`Impl(Mutable m) { scEncl = m; }`

   – définie des méthodes de mise à jour :
   pour chaque variable instable v (de type T)
   si v apparaît dans une sous-classe de spécialisationS_{down} de X
   alors rajoute :

   ```
   T scSet_v(T val)
   // methode 'enabler' :
   {
     scEncl.v = val;
     if(P_v) scEncl.scNotify();
     return val;
   }
   ```

   sinon rajoute :

   ```
   T scSet_v(T val)
   // simple affectation :
   {
     scEncl.v = val;
     return val;
   }
   ```

   – **pour** chaque méthode spécialisable, copie le corps d'origine (générique), dans lequel `this` a été substituée par `scEncl`.

   b) **pour** chaque classe de spécialisation S, construit une implémentation spécialisée `class` S`Impl extends` C`Impl` (où C est le père de S — soit la classe X soit une classe de spécialisation S_{up}) :

   – definie un constructeur :
   S`Impl(Mutable m) { scEncl = m; }`

   – **si** le spécialiseur n'est pas celui hérité de C, **alors** rajoute :
   `static { theSpec = new` *strategie*`(`*taille*`); }`
   NOTE : Le spécialiseur doit être redéfini si S est spécialisable à l'exécution et C est spécialisable à la compilation, ou si S définie sa propre stratégie de cache. Dans le premier cas, la *strategie* et la *taille* prennent des valeurs par défaut. Dans le dernier cas, les valeurs sont celles définies par S.

   – redéfinie certaines méthodes de mise à jour :

     – **pour** chaque variable instable v (de type T) apparaissant dans S, rajoute :

     ```
     T scSet_v(T val)
     // methode 'disabler' :
     {
       scEncl.v = val;
     ```

```
    scEncl.scNotify();
    return val;
  }
```

- **pour** chaque sous-classe de spécialisation S_{down} de S

  pour chaque variable instable v (de type T) apparaissant dans S_{down} et n'apparaissant pas dans S, rajoute :

  ```
  T scSet_v(T val)
  // methode 'enabler' :
  {
    scEncl.v = val;
    if(P_v) scEncl.scNotify();
    return val;
  }
  ```

  NOTE : Un cas particulier doit être traité afin de ne pas dupliquer la méthode `scSet_`v`()` lorsque plusieurs sous-classes de spécialisation définissent v comme invariant. Dans ce cas, définie une unique méthode `scSet_`v"(), dans laquelle le test précédent `scNotify()` est omis.

- **pour** chaque méthode m dans S

  si S est spécialisable à la compilation
  alors rajoute la version spécialisées de m, dans laquelle `this` a été substitué par `scEncl`
  sinon déclare la méthode m comme `native` ; son implémentation sera générée dynamiquement est chargée dans la classe.

NOTE : `Impl` est une classe abstraite, définie dans la bibliothèque de support à l'exécution (ici omise). Elle définie une variable de classe qui réfère au spécialiseur par défaut (à savoir, le spécialiseur à la compilation) :

```
protected final
  class Specializer theSpec = CtSpecializer.theSpec;
```

NOTE : Les méthodes de mise à jour de chaque variable $v \in V_{obj}$ contiennent quelque code supplémentaire (omis dans l'algorithme, pour des raisons de simplicité) qui effectue un détachement de l'ancien objet et un attachement au nouvel objet.

Table des figures

3.1 Implémentation classique, avec les couches ISO 28
3.2 Le protocole virtuel VRPC choisit le protocole sous-jacent selon la destination du message (serveur local, serveur distant, ou groupe de serveurs) 32

4.1 Définition de `mini_printf()` 38
4.2 Spécialisation de `mini_printf()` avec `fmt` égal à `"n = %d"` 38

5.1 Le protocole RPC de Sun 50
5.2 Trace abstraite de la partie encodage d'un appel distant à `rmin` 51
5.3 Lecture ou écriture d'un entier long : `xdr_long()` 52
5.4 Écriture d'un entier long : `xdrmem_putlong()` 52
5.5 Routine d'encodage `xdr_pair()` utilisée dans `rmin()` 53
5.6 Routine spécialisée d'encodage `xdr_pair()` 53
5.7 Comparaison de performance entre IPX/SunOS et PC/Linux 56

6.1 Le protocole IPC de Chorus 63
6.2 Le protocole virtuel `ipcCall` 64
6.3 Spécialisation de code C++ 70
6.4 Invariants à la compilation 71
6.5 Spécialisation à la compilation du protocole IPC 72
6.6 Spécialisation à l'exécution du protocole IPC 74
6.7 Spécialisation incrémentale des IPC de Chorus 75

7.1 Déclaration d'un fichier adaptatif par spécialisation 85
7.2 La syntaxe des classes de spécialisation 87
7.3 La définition d'origine de l'i-node 90
7.4 Une classe de spécialisation pour l'i-node 90
7.5 L'i-node compilé 91
7.6 L'objet englobant de l'i-node 92
7.7 L'implémentation générique de l'i-node 92
7.8 L'implémentation spécialisée de l'i-node 92
7.9 L'interface `Mutable` 93
7.10 Le protocole entre le fichier et l'i-node 93
7.11 La définition d'origine du fichier avec le programme principal 95
7.12 Les classes de spécialisation pour le système de fichiers adaptatif par spécialisation 95

7.13 La hiérarchie de caches 98
7.14 La chaîne de spécialisation d'un composant Java 100

Titre en anglais

An Automatic Approach to Specializing System Components

Abstract

Operating systems are nowadays evolving towards increasingly generic forms. This trend is motivated on one hand by the increasing complexity and diversity of emerging hardware platforms and on the other hand by the integration of new features such as quality of service or security.

Pursuing the traditional philosophy of implementing an abstract machine with a limited number of “universal” services results in systems which are less and less efficient and flexible. The traditional approach to alleviate this problem consists of manually optimizing each system component for a small number of critical cases. However, the rewriting of such optimized components requires a very important expertise in system programming.

This dissertation presents an approach to the optimization of system components that is more accessible to non-expert system programmers. In comparison to a manual approach to specialization, the present approach is (1) automated by using partial evaluation and (2) guided by a declarative specification.

This automatic approach brings several important advantages. It avoids the risk of introducing errors, by preserving the semantics of the original component. It solves the maintainability problems associated with a set of code versions which are functionally equivalent. Finally, it encourages a generalized use of specialization, unconceivable in the case of a manual approach.

Using a prototype partial evaluator, it is shown that this methodology is currently applicable to off-the-shelf system components and can result in significant speedups. Thus, a first example reveals a speedup of up to 3.7 on the XDR layer of the Sun RPC implementation, resulting in a speedup of 1.5 on a distant RPC roundtrip. A second example shows a speedup factor of up to 1.5 on a local IPC of the Chorus microkernel, even though this IPC case is already heavily optimized by hand.

Résumé

Les systèmes d'exploitation évoluent actuellement vers des formes de plus en plus génériques. Cette tendance est motivée d'un côté par le besoin de couvrir un matériel de plus en plus complexe et diversifié, et d'autre part par le besoin d'intégrer de nouveaux types de fonctionnalités tels que la qualité de service ou la sécurité.

En poursuivant la philosophie traditionnelle selon laquelle le système d'exploitation implémente une machine virtuelle avec un nombre limité de fonctionnalités "universelles", les systèmes d'exploitation deviendront de plus en plus lents et inefficaces. Pour pallier à ce problème, l'approche traditionnelle consiste à optimiser manuellement chaque composant système pour l'adapter à un nombre restreint de cas, considérés comme critiques. L'écriture de tels composants systèmes optimisés requiert un niveau très important d'expertise système.

Cette thèse propose une méthodologie d'optimisation de composants système plus accessible à des non-experts en système, reposant sur la spécialisation de programmes. Par rapport à une démarche par spécialisation manuelle, notre approche présente deux niveaux de simplification : la spécialisation est automatisée par l'utilisation de l'évaluation partielle et dirigée par une spécification déclarative.

Cette approche automatique apporte plusieurs avantages essentiels. Elle évite le risque d'introduction d'erreurs, en préservant la sémantique du composant d'origine. Elle résout le problème de maintenabilité d'un ensemble de versions fonctionnellement équivalentes. Enfin, elle encourage une application généralisée, inconcevable dans le cas d'une approche manuelle.

En utilisant Tempo, notre évaluateur partiel, nous montrons que cette méthodologie est applicable à des systèmes existants, commerciaux, et qu'elle peut apporter des gains significatifs. Ainsi, dans un premier exemple, nous obtenons un facteur d'accélération de jusqu'à 3,7 sur la couche XDR du protocole Sun RPC, qui détermine une accélération de 1,5 sur un aller-retour RPC distant. Dans un deuxième exemple, nous obtenons un facteur d'accélération allant jusqu'à 1,5 sur un IPC local du micro-noyau Chorus, malgré le fait que ce service ait déjà été optimisé au moyen de techniques manuelles.

Titre en anglais

An Automatic Approach to Specializing System Components

Mots-clés

Spécialisation de programmes, évaluation partielle, appel de procédure à distance, systèmes d'exploitation extensibles, comportement adaptatif, approche déclarative.

Discipline

Informatique

École Doctorale SPI de l'Université de Rennes 1

Campus de Beaulieu - 35042 Rennes cedex