POLYPUBLIE

Compilation par serveur persistant et génération rapide de code

Jérôme Collin

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

Date	1997
Type	Mémoire ou thèse / Dissertation or Thesis Mémoire de maîtrise/Master's thesis
URL de PolyPublie PolyPublie URL	https://publications.polymtl.ca/8972/
Conditions d'utilisation Terms of use	© Tous droits réservés / All rights reserved
Référence Citation	Collin, J. (1997). Compilation par serveur persistant et génération rapide de code [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8972/

Détails de la thèse
Thesis details

Directeurs de recherche Advisors	Michel Dagenais
Programme Program	Non spécifié

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal
https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

COMPILATION PAR SERVEUR PERSISTANT ET GÉNÉRATION RAPIDE DE CODE

JÉRÔME COLLIN
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JANVIER 1997

© Jérôme Collin, 1997.

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26462-9

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

COMPILATION PAR SERVEUR PERSISTANT ET GÉNÉRATION RAPIDE DE CODE

présenté par: COLLIN Jérôme

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président

M. DAGENAIS Michel, Ph.D., membre et directeur de recherche

M. MERLO Ettore, Ph.D., membre

À tous les membres de ma famille qui m'ont soutenu durant ces années d'étude.

Remerciements

Mes premiers remerciements sont pour mon directeur de recherche, M. Michel Dagenais, Ing., Ph.D., pour ses excellents conseils. Je veux également remercier le Systems Research Center (SRC) de la firme Digital Equipment Corporation® (DEC), et de façon particulière M. Allan Heydon, Ph.D., M. Greg Nelson, Ph.D., et Mme Annelyse Feiereisen, pour m'avoir accueilli durant deux mois à Palo Alto en Californie. M. William Kalsow de Critical Mass®, Inc. a patiemment répondu à mes questions sur les détails les plus complexes du compilateur SRC-Modula-3. Je le remercie.

Mes remerciements iront aussi à mes collègues étudiants, spécialement M. Louis-D. Dubeau, pour nos nombreuses conversations sur le langage Modula-3. Je remercie enfin le Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG) qui a soutenu financièrement mes travaux.

Résumé

Cette étude concerne l'accélération du temps requis pour compiler un programme. L'objectif était de réduire ce temps de manière à ce qu'un programmeur puisse obtenir rapidement les résultats d'une compilation et ainsi augmenter sa productivité. Des modifications ont été apportées au compilateur SRC-Modula-3 pour accélérer la compilation et démontrer les principes exposés.

La première a consisté à transformer le compilateur en un serveur qui s'exécute en permanence. De cette manière, il devient possible de conserver en mémoire des informations sur des compilations précédentes pour qu'elles soient directement disponibles pour des compilations subséquentes. Ces informations prennent la forme de structures de données internes au compilateur, suite à l'analyse lexicale et syntaxique des interfaces d'un code source et des librairies utilisées. On peut alors éviter de refaire cette analyse pour une interface qui n'a pas été modifiée depuis la dernière compilation. Éviter de refaire cette analyse élimine aussi la nécessité de procéder à la vérification statique des dépendances entre les types décrits dans les interfaces, comme le requiert le langage Modula-3. Un algorithme a été développé pour s'assurer

de la validité des informations retenues en mémoire. Cette amélioration est surtout apparente dans un contexte de fréquentes recompilations, lors de la mise au point d'un logiciel. Cette technique, à elle seule, permet un gain de performance de l'ordre de 10% à 20% par rapport au compilateur original. Cette technique a cependant peu d'impact dans un contexte où tous les fichiers d'un programme de taille importante doivent être recompilés.

Une seconde modification a été de substituer la partie du compilateur responsable de la génération du code par une seconde plus rapide. Ce nouveau générateur de code a été spécifiquement conçu pour une plate-forme PC-Linux. Ce générateur est plus léger que son prédécesseur qui lui peut générer du code pour différentes plates-formes et offre une grande souplesse dans sa configuration. Ce remplacement permet de réduire le temps de compilation total (incluant le temps d'édition de liens) de 50% à 70%. Ce résultat est intéressant puisqu'il permet de quantifier la perte de performance engendrée par la partie assurant la portabilité du compilateur.

Abstract

This study presents techniques to speed up the compilation. The goal is to provide a faster compilation cycle for programmers. To achieve this, major modifications were made to the SRC-Modula-3 compiler.

The first was to transform this compiler into a server which can keep a cache containing abstract syntax trees for the interfaces of a program and referenced libraries. By storing these from one compilation to another, the parsing of some interfaces can be avoided as well as the associated type checking. Interfaces may be modified by the programmer at any time. Thus, an algorithm was developed to validate entries in the cache. This server is particularly useful while debugging a program since many recompilations are required. It provides a speed-up of around 10% to 20%. The benefits of this technique are smaller when all source files of a big program need to be recompiled.

The second major modification was to replace the code generator of the compiler by a faster one. This code generator was targeted directly for the PC-Linux architecture and offers few configuration options. The previous code generator one was highly

configurable and was designed to generate code for different platforms. The new code generator can reduce the total compilation time (including the linking time) by 50% to 70%. This result shows the performance overhead due to the use of a portable code generator.

Table des matières

Dédicace iv

Remerciements v

Résumé vi

Abstract viii

Table des matières x

Liste des tableaux xiii

Table des figures xiv

Liste des sigles et abréviations xv

Introduction 1

Chapitre 1: La compilation rapide 5

1.1 Parties d'un compilateur 6

1.2 Modèle du temps de compilation 9

1.3 Interprétation et compilation dynamique 11

1.4 Approches matérielles et parallèles 12

1.5 Recompilation sélective 15

1.6 Recompilation incrémentale 16

1.6.1 Compilation incrémentale complète 17

1.6.2 Édition de liens incrémentale et chargement dynamique 19

1.6.3 Serveur de compilation et précompilation 21

1.7 Partie finale d'un compilateur 25

1.8 Sommaire 27

Chapitre 2: Serveur de compilation 29

2.1 Approche proposée de serveur de compilation 29

2.1.1 Interfaces en Modula-3 29

2.1.2 Cache de compilation 32

2.2 Réalisation du serveur de compilation 40

Chapitre 3: Génération de code 46

3.1 Problème de génération de code 47

3.2 Refonte de la génération de code 50

3.2.1 Approche proposée 50

3.2.2 Exemple de génération de code 52

3.2.3 Structure du générateur de code 56

3.2.4 Modifications effectuées 59

Chapitre 4: Évaluation de la vitesse de compilation 68

4.1 Méthodes d'évaluation de la performance 68

4.2 Performance du générateur de code 73

4.3 Performance du serveur de compilation 79

Conclusion 88

Références 94

Liste des tableaux

4.1 Ensembles compilés retenus pour l'évaluation de performance 70

4.2 Compilation avec générateur de code m3cgcl 72

4.3 Compilation avec générateur de code intégré 74

4.4 Compilation sans l'option "-g" 75

4.5 Compilation sans les options "-g" et "-fPIC" 77

4.6 Compilation avec compilateur compilé sans optimisation 78

4.7 Compilation avec compilateur compilé avec générateur de code intégré 79

4.8 Compilation avec compilateur-serveur et communication TCP/IP . . 80

4.9 Compilation avec compilateur-serveur et interface texte 81

4.10 Compilation avec générateur de code m3cgcl et 64Mo de mémoire vive 82

4.11 Compilation avec compilateur-serveur et 64Mo de mémoire vive . . . 83

4.12 Compilation avec compilateur-serveur et générateur de code intégré . 83

4.13 Recompilation de deux fichiers de l'ensemble postcard 84

4.14 Recompilation de quatre fichiers de l'ensemble postcard 85

Table des figures

1.1 Phase de compilation. 7

1.2 Contexte de compilation d'un fichier d'en-tête. 23

2.1 Dépendance des interfaces en Modula-3 30

2.2 Algorithme de validation 34

2.3 Structure simplifiée du compilateur SRC-Modula-3 standard 40

2.4 Structure simplifiée du compilateur SRC-Modula-3 modifié 41

2.5 Objet réseaux servant d'interface au serveur 43

3.1 Compilation d'une procédure simple. 53

3.2 Enregistrement d'activation 54

3.3 Structure du générateur de code. 57

3.4 Fichier objet relocalisable ELF. 60

3.5 Accès aux variables globales externes 66

Liste des sigles et abréviations

ABI ..... Application Binary Interface

AST ..... Abstract Syntax Tree

ACK ..... Amsterdam Compiler Kit

GCC ..... GNU C Compiler

COFF ..... Common Object File Format

ELF ..... Executable and Linkable Format

GOT ..... Global Offset Table

JIT ..... Just In Time

L.I. ..... Langage Intermédiaire

M3 ..... Modula-3

PC ..... Personal Computer

PIC ..... Position Independent Code

PLT ..... Procedure Linkage Table

RISC ..... Reduced Instruction Set Computer

SRC ..... Systems Research Center

TCP/IP ..... Transmission Control Protocol / Internet Protocol

Introduction

Une des plus simples définitions de l'ordinateur que l'on puisse imaginer est une machine électronique capable d'exécuter les instructions d'un programme. Transmettre ces instructions aux premiers ordinateurs était une tâche ardue. L'équipement électronique était peu fiable. De plus, ces ordinateurs pouvaient difficilement effectuer autre chose que des calculs arithmétiques. Lorsque des ordinateurs pouvant traiter des informations plus générales sont apparus, les méthodes de programmation ont également dû être revues.

Un des problèmes d'alors concernait l'environnement d'exécution des programmes. Des questions comme le chargement en mémoire du programme, la gestion des entrées/sorties devaient être considérées avec beaucoup de détails dans l'élaboration même du programme, en plus de l'information devant être traitée. On a ainsi débuté l'étude des systèmes d'exploitation pour permettre de faciliter l'exécution des programmes par les usagers, et leur élaboration par les programmeurs.

Un autre problème est également apparu, soit la complexité de spécifier des instructions aux ordinateurs, puisque ces derniers ne travaillent que sur des codes bi-

C'est alors que sont apparus les langages de programmation dits évolués et les méthodes pour traduire les programmes écrits dans de tels langages en une forme directement exécutable par l'ordinateur. Ces traductions sont appelées compilations. Les langages de programmation permettent donc d'écrire plus facilement des programmes et de les compiler pour des architectures d'ordinateurs différentes.

Le domaine des langages de programmation est encore aujourd'hui l'objet d'intenses recherches. Les dernières années ont surtout vu naître des développements intéressants du côté des techniques pour la programmation par objets, du support à l'exécution (*run-time*), et des méthodes permettant de faire face à des programmes de taille importante. Il va de soi que de nouvelles caractéristiques dans les langages entraînent aussi de nouveaux développements du côté des compilateurs.

Les avantages procurés par les systèmes d'exploitation et les langages de programmation ont grandement contribué à démocratiser la programmation. On compte ainsi aujourd'hui des millions de programmeurs à travers le monde. Pour ces gens, la mise au point d'un logiciel est une activité régulière voire quotidienne. Il n'est pas rare qu'ils aient à écrire quelques lignes de code, compiler, modifier quelque peu ces lignes, recompiler et reprendre ces étapes jusqu'à l'obtention du résultat désiré. Le temps passé par le programmeur à attendre les résultats de la compilation peut alors représenter une fraction non négligeable du temps de mise au point. D'ailleurs, une étude [28] a montré qu'environ 80% des recompilations via la commande *make* impliquent moins de 3 modules. Abman [1] présente deux autres raisons qui ont rendu inévitable

l'étude du problème. Premièrement, la taille des logiciels à développer tend à augmenter rapidement demandant un effort accru au système de compilation. Enfin, les nouveaux langages exigent davantage du compilateur. On peut ici penser à l'analyse statique des types de plus en plus exigeante, ou à des langages fort complexes comme C++ [45].

La présente recherche avait comme objectif d'accélérer le travail effectué par un compilateur. L'étude tient très peu compte de la qualité du code objet à produire en terme de vitesse d'exécution ou de taille du fichier binaire résultant. Bien qu'il s'agisse de qualités souhaitables pour un exécutable, elles pourront toujours être considérées dans une compilation finale du code source par un compilateur modifié ou même différent. Le but est plutôt d'en arriver à compiler rapidement un programme en cours de développement.

Pour démontrer l'efficacité des techniques proposées, un compilateur existant, le SRC-Modula-3 [15] a été modifié. Plusieurs raisons ont motivé ce choix. En premier lieu, il s'agit d'un compilateur dont le code source était disponible. De plus, le langage Modula-3 (M3) [17] possède des caractéristiques qui en font l'un des plus avancés. Le compilateur est également disponible sur la plupart des plates-formes les plus utilisées. Les principes exposés dans cette étude peuvent cependant s'appliquer à la plupart des compilateurs. Le terme compilateur sera cependant ici réduit à une définition de "producteur de code objet exécutable", ce qui exclut par exemple les compilateurs de silicium.

Le premier chapitre de ce mémoire sera consacré à une revue des approches déjà rencontrées dans la littérature sur le sujet. Le début du chapitre 2 présentera les grandes parties du compilateur SRC-Modula-3. Le reste du chapitre exposera comment le fait de conserver des résultats de compilations précédentes en mémoire peut accélérer celles à venir du même code source. Le chapitre suivant sera consacré au générateur de code du compilateur. Ce chapitre démontrera l'importance de cette partie dans la performance globale du système.

Plusieurs facteurs extérieurs au compilateur comme le type de système d'exploitation et la vitesse du processeur peuvent influencer la vitesse de compilation. Pour éviter de tenir compte de ceux-ci, toutes les évaluations de performance ont été effectuées sur un même ordinateur Intel® Pentium® avec le système d'exploitation Linux. Les résultats sont présentés juste avant la conclusion.

Chapitre 1

La compilation rapide

Ce chapitre présente différentes études concernant le temps de compilation. D'abord, un modèle théorique du temps de compilation sera présenté. L'interprétation du code plutôt que sa compilation sera aussi discutée. La compilation est parallélisable ou peut être accélérée par du matériel (*hardware*) dédié. Quelques approches dans ce domaine seront soulignées. Les approches de compilations sélectives et incrémentales sont parmi les plus intéressantes dans la littérature et feront l'objet de deux sections. La génération de code sera le sujet de la dernière section. Pour débuter, la première section présentera les principales parties d'un compilateur, ce qui sera utile pour la suite de la lecture de ce mémoire.

Cette revue de littérature ne peut évidemment prétendre être complète. Un survol des principales techniques est présenté avec insistance sur les approches se rapprochant le plus de celles développées durant cette étude.

1.1 Parties d'un compilateur

Cette section se veut une présentation générale de la compilation et des compilateurs. Elle est un résumé du chapitre 1 "introduction à la compilation" de Aho, Sethi et Ullman [18]. Le but est de présenter quelques concepts de base qui seront utiles pour la suite de la lecture du texte. Les systèmes de compilation peuvent avoir de nombreuses variantes mais la discussion se restreint volontairement à ce qui est le plus typique.

Un compilateur a pour rôle de traduire un programme source en un programme cible, comme le montre la figure 1.1. La compilation comporte deux parties, l'analyse et la synthèse. L'analyse permet d'en arriver à une représentation intermédiaire après avoir séparé le code source en ses constituantes. La synthèse permet de générer le code cible à partir de cette représentation intermédiaire.

Un compilateur est souvent constitué de huit phases, tel qu'illustré à la figure 1.1. Les analyseurs lexical, syntaxique et sémantique sont associés à l'analyse. Les trois autres qui suivent permettent la synthèse. Une part non négligeable du travail du compilateur consiste également à fournir un rapport mentionnant les erreurs dans le programme source, ce qui est la responsabilité du gestionnaire d'erreurs. Le compilateur doit aussi maintenir une table des symboles tout au long de la compilation. Cette dernière regroupe les identificateurs présents dans le programme source (comme le nom d'une variable par exemple) et les attributs qui leur sont associés (type d'une variable, position en mémoire, etc.)

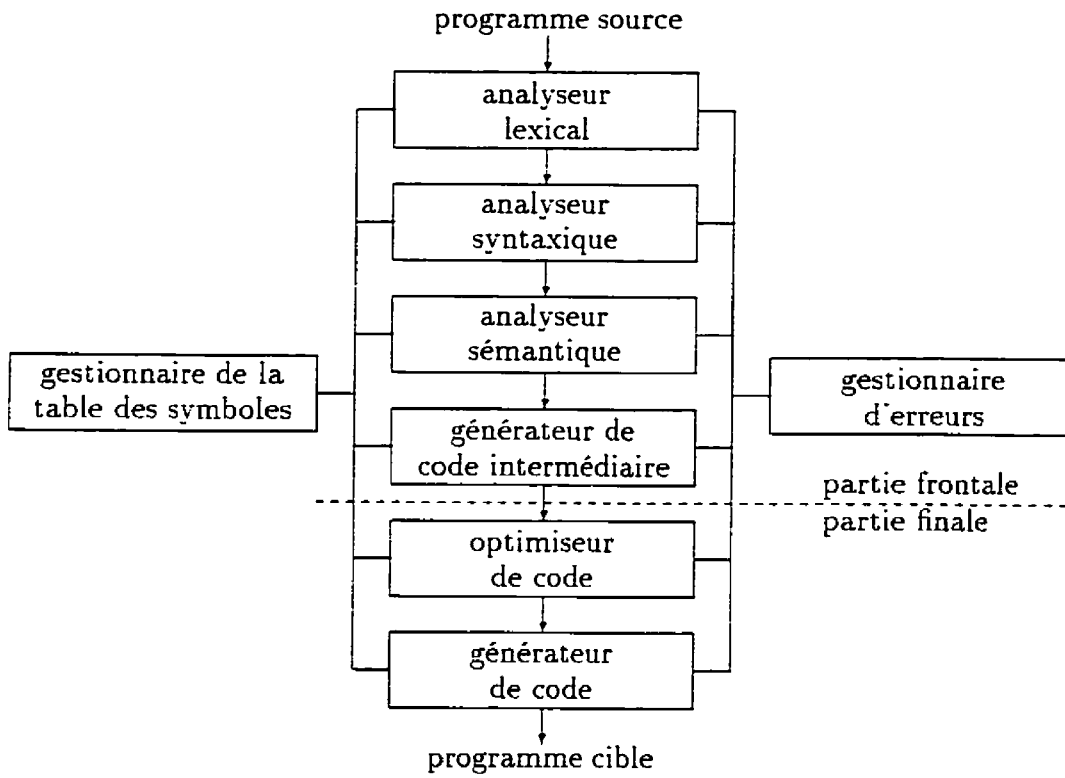


Figure 1.1: Phase de compilation.

L'analyse lexicale est la phase responsable de regrouper les suites de caractères du code source ayant une signification précise et cohérente que l'on nomme unité lexicale (*token*). Ce peut être un mot réservé du langage, un nom de variable ou un signe de ponctuation par exemple. L'analyse syntaxique structure l'ensemble des unités lexicales sous la forme d'un arbre appelé arbre syntaxique abstrait (*Abstract Syntax Tree–AST*) où un noeud représente un opérateur et ses fils ses opérandes. L'analyse sémantique s'assure que cet arbre forme un tout significatif. C'est ici que l'on aura soin de vérifier, par exemple, si un réel n'est pas employé comme indice

dans un tableau. Les algorithmes et techniques développés pour l'analyse ne seront pas détaillés d'avantage car ils débordent du cadre de cette étude. Cependant, comme les arbres abstraits sont une représentation intermédiaire de la compilation, le chapitre suivant montrera que leur réutilisation de compilation en compilation peut permettre d'accélérer l'exécution d'un compilateur.

Le générateur de code intermédiaire transforme l'AST produit par l'analyse en code intermédiaire. Ce code est produit suivant une syntaxe précise d'où le nom de "langage intermédiaire" souvent employé. Ce dernier sert en réalité d'interface entre deux parties d'un compilateur, la partie frontale (*front-end*) et la partie finale (*back-end*). La première génère le code intermédiaire qui est indépendant du code cible. La partie finale, à partie du code intermédiaire, produit un code spécifique à une machine avec un niveau d'optimisation désiré. Le chapitre 3 traitera plus en détail de la partie finale puisque cette étude s'y est intéressée. Cette division en deux parties permet aussi de regrouper des phases pour leur permettre de collaborer entre elles avec une plus grande efficacité. Il arrive également que l'on structure un compilateur en passes. Deux passes communiquent entre elles au moyen d'un fichier. Ainsi, il arrive que les parties frontale et finales correspondent à deux passes distinctes.

Un compilateur agit rarement seul. Souvent, le générateur de code produit un fichier en langage d'assemblage. Un assembleur doit, dans une autre passe, terminer la production d'un fichier objet relocalisable ou fichier ".o". De plus, le code source peut être contenu en plusieurs fichiers et peut aussi faire référence à des librairies.

Le chargeur/relieur, ou éditeur de liens (*linker*), complète les références prévues dans les fichiers objets relocalisables et génère un fichier objet exécutable. D'autre part, avant d'être transmis au compilateur, le code source peut également être traité par un pré-processeur, comme c'est le cas avec le langage C. Le préprocesseur permet de copier le contenu d'un fichier à inclure dans le texte du code source. Il permet aussi de définir des "macros" qui sont des abréviations pour des portions de code plus longues.

1.2 Modèle du temps de compilation

Shaw et al. [27] ont développé un modèle pour évaluer le temps de compilation. Le but de ce modèle est d'avoir une base d'analyse pour étudier le temps de compilation et pour comparer des compilateurs et les architectures pour lesquelles ils sont portés. Ils en arrivent à la formule suivante:

$$T = K V^a (V^*)^b \tag{1.1}$$

où T est le temps de compilation, K est le taux de traitement qui caractérise chaque compilateur et a et b sont deux constantes à fixer. V et V^* sont définis comme suit:

$$V = (N_1 + N_2) \log_2(n_1 + n_2) \tag{1.2}$$

$$V^* = (2 + n_2^*) \log_2(2 + n_2^*) \tag{1.3}$$

Ici, n_1 et n_2 représentent respectivement le nombre d'opérateurs et d'opérandes dans le code source alors que N_1 et N_2 sont respectivement les nombres totaux d'apparitions d'opérateurs et d'opérandes. n_2^* est semblable à n_2 mais dénombre uniquement les opérandes d'entrée/sortie.

Les auteurs ont évalué leur modèle pour quatre compilateurs Ada, sur quatre architectures différentes, avec une série de programmes reconnus pour l'évaluation de performance la des compilateurs Ada. Ils en sont arrivés à des valeurs de a et b de 0.48 et 0.07 respectivement. La corrélation entre le modèle théorique et les résultats expérimentaux varie entre 0.74 et 0.88, ce qui est bon. Il arrive cependant que le facteur V^* ne soit pas statistiquement significatif. Par ailleurs, plus le temps de compilation augmente, plus la différence entre le temps prédit et le temps observé augmente. Le modèle peut également moins bien correspondre aux résultats expérimentaux pour certaines caractéristiques du code source comme la concentration de certains opérateurs où le nombre plus ou moins élevé de fonctions par exemple. Il est cependant très intéressant de constater que le paramètre K permet de caractériser la performance du compilateur.

Peu d'autres études ont été effectuées pour développer un modèle permettant d'analyser le temps de compilation sur une base théorique. La plupart des techniques de compilation rapide sont évaluées par mesure directes du temps pris par le compilateur pour fournir un résultat.

1.3 Interprétation et compilation dynamique

Traditionnellement, la solution au problème de la lenteur de compilation a été de substituer la compilation par l'interprétation, où le code est généré au fur et à mesure de l'exécution. Il arrive ainsi que l'on ait recours à un interpréteur durant le développement d'un logiciel et de recourir au compilateur durant les dernières étapes seulement. Le principal inconvénient de cette méthode a toujours été la lenteur de l'exécution du programme. De plus, certaines caractéristiques comme la vérification statique des types peuvent difficilement se faire avec l'interprétation.

Pletzbert [42] propose une méthode pour laquelle l'exécution débute par une interprétation mais où une compilation s'effectue parallèlement. Les parties de code compilées sont appelées graduellement pour éliminer l'interprétation. L'auteur fait également le résumé d'une approche légèrement différente soit celle de la compilation juste-à-temps (*just-in-time ou JIT*) pour le langage orienté objet Java. La différence vient du fait qu'il n'y a pas d'interprétation. Les méthodes sont plutôt compilées juste avant leur appel au fur et à mesure du déroulement du programme. Le compilateur est donc appelé autant de fois que le nombre de méthodes différentes à être exécutées. L'exécution du code est également suspendue d'autant, le temps que la compilation soit terminée, ce qui varie avec la grosseur de la méthode à compiler. Il s'agit donc d'une compilation à la demande que l'on nomme compilation dynamique. Cette approche a surtout été développée pour le langage SELF. Hölzle et Ungar [13] vont encore plus loin dans une variante de SELF-93 où les méthodes déjà compilées une

première fois, mais qui sont fréquemment appelées, peuvent être recompilées pour être optimisées. Le but est toujours d'obtenir une réponse rapide à une modification dans le code source tout en ayant un programme s'exécutant rapidement.

1.4 Approches matérielles et parallèles

Il est reconnu que l'ajout de matériel dédié ou la parallélisation sont des approches à considérer pour accélérer une application s'exécutant sur un monoprocesseur. Quelques études qui ont suivi cette direction sont soulignées dans cette section.

La présentation des parties d'un compilateur au début de ce chapitre suggère une organisation pipeline du traitement d'un code source jusqu'à l'obtention d'un fichier binaire. Chu [2] présente un ensemble de co-processeurs qui suit ce modèle. Les analyses lexicale et syntaxique sont effectuées par des processeurs spécialisés alors que les autres phases continuent d'être effectuées par un microprocesseur conventionnel. Les processeurs fonctionnent sur 8 étages de pipeline (4 pour l'analyse lexicale et 4 pour l'analyse syntaxique). Les auteurs n'ont pas fourni de résultats dans le contexte d'un langage particulier.

Noyé [40] suggère une approche allant plus loin de ce côté pour l'accélération du langage Prolog où le matériel dédié devient une machine de plus haut niveau. L'ensemble du circuit a un jeu d'instruction qui est un mélange d'instructions Prolog de haut niveau et d'instructions d'un processeur RISC (*Reduced Instruction Set Computer*). Il a également sa propre mémoire et un environnement d'exécution logiciel

indépendant (pour les co-routines, le ramasse-miettes, etc...). Il communique avec une machine Unix au besoin. Le système est environ deux fois plus rapide qu'une station SunSPARC 2 offrant un support pour Prolog équivalent même si la vitesse d'horloge est deux fois moins rapide. Cette machine Prolog requiert un minimum de 32 Mega-octets de mémoire vive et est fabriquée de composants standards et de deux puces fabriquées sur mesure. Un support logiciel est aussi offert pour la mise au point (*debugging*) et l'auto-amorçage (*bootstraping*) du système.

La compilation présente un problème intéressant à paralléliser. Wortman et Junkin [22] présentent une des études les plus avancées dans cette voie. L'objectif était de paralléliser jusqu'à compiler les procédures d'un module sur des processeurs différents. Une analyse lexicale permet de séparer les procédures d'un module. Un processus superviseur distribue les tâches sur chaque processeur. Ici, une tâche correspond à la fois à une phase de compilation et à un processus. Il n'y a qu'à la toute fin de la compilation que le code généré par les différents processeurs est réuni pour former un fichier binaire. Le problème le plus important d'une telle approche est celui du maintien de la table des symboles. Il peut arriver qu'une tâche sur un processeur ait besoin d'information sur un symbole se trouvant défini sur un autre processeur. Les auteurs suggèrent 4 algorithmes à ce problème appelé *"pas encore connu"* (*"doesn't know yet"*). Le but étant ici de réduire le plus possible les blocages des tâches en attente de renseignements sur un symbole et de limiter le nombre de communications interprocessus. Un compilateur Modula-2+ a été conçu suivant cette organisation pour

une architecture Firefly. Les résultats montrent que l'accélération augmente avec le nombre de processeurs de manière assez satisfaisante tant que les modules du code source ont une taille suffisamment importante pour présenter un potentiel pour la parallélisation.

Khana. Ghafoor et Goel [6] proposent la parallélisation de la compilation suivant une autre méthode. Le partitionnement se fait selon la grammaire du langage. Ainsi. le même code source est présenté en entrée à tous les processeurs. Chaque processeur compilera les éléments de grammaire qu'il doit reconnaître et ignorera les autres. Toutes les parties compilées séparément sont regroupées à la fin pour former le fichier objet. Le problème de cette approche est d'arriver à bien regrouper les éléments de grammaire à confier à chaque sous-compilateur pour balancer les charges de travail sur chaque processeur. Les auteurs proposent le partitionnement d'une grammaire Pascal mais n'ont pas réalisé le compilateur suivant ce principe.

Comme on peut le voir. les approches matérielles et parallèles sont variées et souvent efficaces puisque le problème de la compilation s'y prête bien. Le coût du matériel dédié et des ordinateurs parallèles fait cependant en sorte que ces compilateurs ne soient pas très répandus. De plus. ils sont rarement présentés dans un contexte de mise au point d'un logiciel. où la recompilation de seulement quelques modules est très fréquente. ce qui est l'intérêt principal de la présente étude.

1.5 Recompilation sélective

Bien que l'approche visant à accélérer la capacité brute de traitement d'un compilateur soit la solution principale au problème de la compilation rapide, d'autres se sont également demandés si certaines compilations n'étaient pas purement inutiles. Réduire le code à recompiler est tout aussi important que de le compiler rapidement. Une approche longtemps employée a consisté à recompiler toutes les unités de compilation dépendant directement ou indirectement d'un fichier venant d'être modifié (recompilation en cascade), ce qui est déjà mieux que de recompiler tout le code à chaque modification. Des progrès ont cependant été faits dans ce domaine.

La "recompilation intelligente" (*smart recompilation*) de Tichy [46] a sûrement été l'une des plus importantes contributions à ce problème. L'idée consiste à maintenir pour chaque unité de compilation ce qu'il appelle un contexte. Un contexte représente les éléments extérieurs (types, variables, déclarations de procédures, etc...) auxquels une unité de compilation peut faire référence, ou ceux qu'il doit fournir. Lorsqu'un fichier est modifié, des éléments sont ajoutés, retirés ou modifiés au contexte. Ces derniers forment un ensemble de changements. Seulement les unités de compilation qui font référence à un élément de cet ensemble, directement ou indirectement, seront recompilées. Si une unité de compilation réfère à un élément en dehors de l'ensemble modifié, elle n'a aucunement besoin d'être recompilée. Un tel système, mis au point par l'auteur pour un compilateur Pascal, a montré que le coût engendré pour gérer l'information sur les contextes est généralement beaucoup moindre que celui qui serait

requis pour compiler les fichiers n'ayant pas besoin de l'être.

Plusieurs variantes plus ou moins agressives par rapport à la stratégie précédente ont été développées. Par exemple, l'approche de recompilation "plus intelligente" permet de recompiler moins d'unités de compilation que nécessaire, uniquement pour la durée d'un test, mais peut laisser le code source dans un état incohérent du point de vue des types. Si le test montre que le changement apporté au code n'est pas pertinent, une partie du code n'aura pas eu à être recompilée, puisque le code original avant la modification est de nouveau réutilisé. D'autres stratégies recompilent un peu plus de fichiers que nécessaire mais ont un système de gestion de contexte plus simple et plus rapide. Tichy et al. [8] ont classé et évalué ces différentes stratégies dans le cadre d'un projet de programmation avec Ada. D'après les résultats obtenus, jusqu'à 50% du temps de compilation peut être épargné avec un système de recompilation intelligent par rapport à la recompilation en cascade.

Le compilateur SRC-Modula-3 possède un système de recompilation intelligent bien intégré. Pour chaque librairie ou programme (*package*), l'information sur le contexte est placée dans un fichier ".m3x". Le chapitre suivant exposera des concepts qui feront intervenir ce fichier.

1.6 Recompilation incrémentale

La recompilation rapide est une des approches les plus intéressantes au problème du temps de compilation. La méthode générale consiste à conserver les résultats d'une

compilation précédente en vue de leur réutilisation dans une compilation subséquente. Plusieurs méthodes différentes suivent ce principe. Dans un certain sens, l'approche de recompilation intelligente de la section précédente est incrémentale puisque le contexte de recompilation est toujours déterminé par les compilations précédentes. Cependant, ce concept est beaucoup plus développé par certains auteurs. Quelques approches possibles sont présentées dans cette section.

1.6.1 Compilation incrémentale complète

Magpie [43] est un système de compilation complètement incrémental pour le langage Pascal. Il s'agit d'un système de compilation conventionnel mais où les structures de données internes au compilateur sont continuellement maintenues et modifiées au cours de l'édition du code source. L'incrément varie pour chaque phase du compilateur. L'éditeur transmet au compilateur chaque changement de caractère. L'analyseur lexical transmet à l'analyseur syntaxique et sémantique chaque changement d'unité lexicale. Le générateur de code, l'éditeur de liens et le chargeur travaillent au niveau d'incrément de la procédure. Chaque partie du système supporte l'ajout, le retrait et la modification dans le code source. Le temps de compilation peut ainsi devenir pratiquement nul puisque la compilation est toujours en cours. La contrepartie est évidemment la quantité de mémoire extrêmement élevée nécessaire à l'exécution d'un tel système. Magpie demande 1500 octets pour représenter une seule ligne de code source. Le système requiert également une station de travail contenant deux proces-

seurs MC68000. Le système n'a jamais pu compiler un code de plus d'environ 5000 lignes.

Ce genre d'approche demande une intégration très minutieuse des éléments du système de l'éditeur ou chargeur. L'éditeur devient ici un élément fondamental. Beetem [7] montre comment l'éditeur du système Galaxy utilise des listes de chaines de caractères pour maintenir les modifications effectuées au code source par l'usager. Il suggère aussi de laisser le soin à l'usager de débuter une compilation au moment jugé nécessaire, puisqu'il arrive lors de l'édition que le code soit dans un état de modification qui amènerait le compilateur à afficher inutilement des messages d'erreurs. Un effort accru est nécessaire de la part du système si du code exécutable optimisé est requis. Bivens et Soffa [21], entre autres, ont mis au point un système ayant de telles possibilités.

Certains auteurs montrent également que le choix du langage de programmation à compiler a une influence certaine. Galaxy [7] et surtout INC [26] ont été développés dans ce but de faciliter le traitement incrémental. Reconnaissant la lourdeur de l'approche incrémentale complète, certains ont suggéré son emploi uniquement pour certaines parties de la compilation comme le montrent les deux prochaines sous-sections.

1.6.2 Édition de liens incrémentale et chargement dynamique

Beaucoup d'attention est porté à la compilation. Cependant, comme il a été mentionné dans l'introduction, Linton et Quong [28] ont démontré que 80% des compilations impliquaient moins de 2 modules. Par contre, l'édition de liens doit tenir compte de tous les modules pour son traitement, y compris ceux des librairies référencées directement ou indirectement. Il devient donc pratiquement impossible, autrement que par une approche incrémentale, de rendre la tâche effectuée par l'éditeur de liens proportionnelle à une modification plutôt qu'à tout le code. Reconnaissant que l'approche incrémentale réservée uniquement à l'édition de liens pourrait être très avantageuse, Quong et Linton ont développé un éditeur de liens incrémental, Inclink [19]. Ce dernier s'exécute en permanence et maintient l'information nécessaire à l'édition de liens. Cette information est présente sous forme de graphe de dépendance d'utilisation. Ainsi, lorsqu'un symbole est déplacé, une liste fournit les adresses à modifier pour tenir compte du changement. Pour diminuer les déplacements d'un trop grand nombre de symboles à chaque édition de liens, Inclink alloue 24% plus d'espace à un module que nécessaire. Cette stratégie fait en sorte que 97% du temps, un nouveau module un peu plus gros pourra s'intercaler à la place de l'ancien qu'il remplace, ce qui évite de déplacer les autres non affectés par ce changement. Des mesures ont montré que 88% des éditions de liens s'effectuent en moins de 2 secondes sur un MicroVAX-2, et ce peu importe la taille du programme. Un code source composé de 59 modules et 5 librairies a nécessité cinq secondes à Inclink alors qu'il en a fallu 45 à l'éditeur

de liens usuel. 8 Méga-octets de mémoire virtuelle ont été utilisés pour maintenir les structures de données internes de Inclink.

Une approche légèrement différente de l'édition de liens incrémentale est le chargement dynamique (*dynamic loading*). Ici, au lieu de créer un nouvel exécutable pour chaque exécution du programme, un module est ajouté ou remplacé directement au processus s'exécutant en mémoire. Crowe [4] présente une approche pour le chargement dynamique sous Unix. Du point de vue du programmeur, il suffit d'appeler une fonction avec le nom du fichier en paramètre pour que de nouveaux modules soient appelés dynamiquement dans le programme. Ce que fait en réalité cette fonction est d'appeler le compilateur pour créer un fichier objet relocalisable. Il faut ensuite déterminer la taille de ce module objet et réserver de l'espace mémoire correspondant avec la fonction `malloc()` sous Unix. Le module peut alors être copié en mémoire et sa table de symbole peut être combinée à celle du reste du programme. Si ce nouveau module en remplace un autre, il faut détruire l'ancien (`free()`) ainsi que la vieille table des symboles qui lui est associée. Pour rendre possible cette approche, les appels de procédures ne se font plus en prenant directement leur adresse. À la place, une table d'indirection est maintenue en mémoire. Lorsqu'une procédure est déplacée en mémoire, il suffit de modifier l'adresse dans cette table au lieu de celles en tous les points d'appel. Cette solution est plus simple mais l'exécution du code est légèrement plus lente, étant donnée l'indirection. L'auteur a modifié un assembleur, et quelque peu un compilateur C, pour arriver à ce chargement dynamique. Une approche assez

semblable a aussi été réalisée pour Ada par Inveradi et Mazzanti [16].

Beaucoup de systèmes d'exploitation tendent à offrir un support pour le chargement dynamique. C'est le cas pour le format de fichier binaire ELF (*Executable and Linkable Format*) [34] dont il sera question plus loin dans cet ouvrage.

1.6.3 Serveur de compilation et précompilation

Durant le processus de compilation, un temps plus important peut être consacré à certains traitements qu'à d'autre. En offrant un meilleur support par compilation incrémentale pour le traitement plus lent, un facteur d'accélération intéressant est envisageable. Par ailleurs, beaucoup de langages offrent un mécanisme permettant d'interfacer des portions de code ou de partager des éléments communs. En langage C et C++, les fichiers d'en-tête ou fichiers "*.h*" (*header files*) en sont un exemple. En Modula-3, les interfaces jouent ce même rôle. Réutiliser les résultats des compilations précédentes de ces fichiers peut être très avantageux car plusieurs autres fichiers source leur font référence, ce qui demande un traitement important par le compilateur. De plus, ces fichiers d'en-tête ou interfaces sont moins fréquemment recompilés que les autres. L'approche incrémentale est donc appropriée. En général, avec cette approche, le compilateur devient un serveur qui s'exécute continuellement, ce qui permet de maintenir les structures de données internes associées à ces fichiers. Une approche alternative consiste à sauvegarder ces derniers dans un fichier qui est relu à chaque compilation; cette approche impose cependant une relecture du fichier précompilé à

chaque appel du compilateur.

Comme le langage C et ses variantes orientées objet (C++, Objective-C et COB) sont très étudiés, plusieurs approches de serveur de compilation ou de fichiers d'en-tête précompilés ont été développées pour ces langages. La mise au point de ces systèmes représente cependant un problème assez complexe. Sans trop élaborer sur les détails de chaque approche, quelques particularités associées aux fichiers "*.h*" sont intéressantes à souligner. La source du problème vient de la manière dont est réalisée la directive `#include`. En rencontrant celle-ci, un pré-processeur copie le fichier d'en-tête à cet endroit dans le fichier source. Par la suite, chaque fichier "*.c*" sera compilé séparément pour former un fichier objet relocalisable ("*.o*") correspondant. Un fichier "*.h*" aura donc été copié dans d'autres fichiers "*.h*" ou des fichiers "*.c*" du code source autant de fois que précisé par la directive `#include`. Le fait qu'il ne soit pas compilé séparément pour mener à un fichier objet pause un problème puisqu'il génère des entrées dans les tables de symboles des fichiers "*.c*" dans lesquels il est inclus. Ce problème serait peu important si ce n'était du fait qu'un fichier "*.h*" peut voir son contenu interprété de manière différente par le pré-processeur selon les directives conditionnelles de compilation (`#ifdef et #ifndef`). Ainsi, selon la valeur de `DEBUG` dans le programme de la figure 1.2, seulement une des deux déclarations de variable sera copiée à l'endroit de la directive `#include` par le pré-processeur.

Ce problème est majeur puisque la compilation d'un fichier d'en-tête dépend du contexte créé par les directives conditionnelles de compilation. La plupart des ser-

```
#ifdef DEBUG
int x
#else
int y
#endif
```

Figure 1.2: Contexte de compilation d'un fichier d'en-tête.

veurs de compilation ou des systèmes utilisant les fichiers d'en-tête précompilés se distinguent suivant leur approche par rapport à ce problème. Horspool [24] classe ses approches suivant 6 catégories. La première catégorie est celle où simplement l'analyse lexicale du fichier d'en-tête est maintenue comme représentation intermédiaire. D'autres utilisent une représentation précompilée d'un fichier d'en-tête qui doit être inclus dans un fichier "*.c*" avant les autres fichiers "*.h*" et pour la même séquence de directives conditionnelles de compilation. En dehors de ces conditions, les fichiers sources sont utilisés à la place des précompilés. On s'assure ainsi que le contexte de compilation ne change pas. Les approches les plus sophistiquées vont jusqu'à maintenir des représentations internes individuelles pour chaque fichier inclus en tenant compte du contexte de compilation. La gestion des tables de symboles devient alors assez complexe dans ses détails. Clairement, les mécanismes du pré-processeur posent de nombreuses difficultés pour un serveur de compilation ou un mécanisme de précompilation des fichiers d'en-tête. Avec COB, Onodera [41] a réalisé un serveur de compilation qui maintient uniquement les interfaces précompilés spécifiant des objets. Les fichiers d'en-tête en langage C peuvent être utilisées avec COB mais le serveur ne peut en maintenir une version precompilée.

Gutknecht [31] présente un système d'interfaces précompilés pour le langage Modula-2 en plus de classer les différentes approches pour ce langage. Comme les fichiers de définitions en Modula-2 (interfaces en Modula-3) ne peuvent contenir de code exécutable, contrairement aux implantations (modules en Modula-3), l'auteur emploie le terme "fichier de symboles" (*symbol file*) pour désigner l'interface précompilée. Il utilise deux critères pour le classement soit la quantité d'information contenue dans le fichier de symboles et l'encodage utilisé pour les renseignements à enregistrer. Ainsi, si l'information contenue dans le fichier reprend simplement celle du fichier source correspondant, cette approche est de type A. Par contre, si le fichier contient en plus l'information sur le contenu des fichiers récursivement importés, l'approche est alors de type B. Le fichier de symboles contient alors toute l'information sur un fichier source de définitions en plus de celle sur tous les autres dont il dépend. Le classement peut aussi se faire suivant la tendance de la présentation de l'information à l'intérieur du fichier de symboles. Si cette information est sous une forme très semblable à celle du code source, on la classe de type α. Si cette même information encode l'information de la table des symboles du compilateur, elle est plutôt de type β. L'auteur présente une approche de type Bβ et détaille les structures du système sans toutefois fournir de données sur sa performance. L'approche de type Bβ est potentiellement plus intéressante que les autres car un seul fichier de symboles peut contenir de l'information sur plusieurs fichiers source et l'information encodée qu'il contient peut plus directement permettre de reconstituer la table des symboles du

compilateur.

1.7 Partie finale d'un compilateur

Le chapitre 3 traitera d'un générateur de code rapide. Quelques aspects de la littérature sur cette partie d'un compilateur sont présentés dans cette section avec emphase sur l'aspect de la vitesse de compilation.

La partie frontale d'un compilateur dépend fortement des caractéristiques du langage à compiler et d'algorithmes d'analyses spécifiques. La partie finale, quant à elle, peut être réalisée avec passablement de variantes pour un même langage. Ainsi, la partie finale de la figure 1.1 au début de ce chapitre, où un optimiseur est suivi par un générateur de code, n'est qu'une des dispositions possibles. Il arrive également qu'il y ait un second optimiseur avant ou après le générateur de code. On peut aussi se passer d'optimisation, ou la réaliser avec la génération de code. Certaines approches vont jusqu'à proposer un générateur de code qui produit un code en langage C, qui peut être compilé par la suite. Le langage C joue ainsi le rôle d'un langage d'assemblage de haut niveau. Comme il existe un compilateur C sur presque n'importe quel ordinateur, la portabilité du générateur de code devient, elle aussi, assurée. Cette approche est évidemment coûteuse en termes de vitesse de compilation. Plusieurs réalisations ont eu lieu suivant ce principe, entre autre pour les langages Cedar [20] et ML [25].

Fondamentalement, le problème de la génération de code est NP-complet si on considère qu'il faut produire la meilleure séquence de code machine en terme de per-

formance et de compacité. Il faut en plus tenir compte du fait que certains standards doivent être respectés au niveau binaire: des registres du processeur peuvent être réservés à des fins prédéterminées, le fichier binaire doit obéir à un certain format, etc. Dans la pratique, il est possible de générer du code de façon relativement simple. Le problème est que le code produit sera souvent peu performant par rapport à celui optimisé. C'est la raison pour laquelle la majorité des études réalisée sur la partie finale sont dirigées vers l'optimisation du code. On cherchera aussi à réaliser une partie finale qui puisse être modifiée sans trop de difficultés pour produire un code pour des architectures différentes.

Tanenbaum et al. [9] présentent un générateur de code rapide pour leur "Amsterdam Compiler Kit" (*ACK*). ACK est d'abord composé de six parties frontales pour autant de langages différents dont Modula-2, C et Pascal. Des parties finales existent pour de nombreuses Architectures. Chaque partie frontale peut être jumelée avec n'importe quelle partie finale pour former un nouveau compilateur pour une architecture donnée car le langage intermédiaire est toujours le même. Le problème était que la partie finale était passablement lente. Elle était constituée d'une phase de génération de code et trois d'optimisation, en plus de passer par un assembleur à la toute fin. De plus, quelques-unes de ces phases étaient des passes. Pour des fins de mise au point du code, ils ont jugé que la vitesse d'exécution devait l'emporter sur la qualité du code produit. Ils ont donc retiré les optimiseurs et l'assembleur de la partie finale. Le générateur de code a été modifié pour produire directement du code

relocalisable. Il effectue aussi quelques optimisations faciles à réaliser au passage avec la génération du code. La partie frontale a été légèrement modifiée pour appeler directement les procédures du générateur de code, au lieu d'écrire le code intermédiaire dans un fichier à être traité par la partie finale. En comparant leur compilateur pour le langage C aux compilateurs commerciaux disponibles, ils ont noté une réduction du temps de compilation par un facteur de 3 à 4.

D'autres approches proposent de passer par une phase qui fera les optimisations les plus importantes tout en acceptant un degré variable de pénalité sur la performance. McKenzie [38] ainsi que Davidson et Whalley [11], entre autres, ont suivi cette stratégie. Dans ce genre d'approche, on se limite généralement à une optimisation à lucarne (*peephole optimisation*). Cette technique examine des séquences d'instructions en essayant de les substituer par des séquences équivalentes plus courtes et plus performantes. La lucarne correspond à cette petite fenêtre qui est déplacée sur le code cible pour délimiter un groupe d'instructions. Il peut arriver qu'une amélioration puisse donner lieu à de nouveaux remplacements. L'opération nécessite donc un certain nombre de passes.

1.8 Sommaire

Comme le montre ce chapitre, la compilation est un processus complexe qui se subdivise en plusieurs sous-problèmes bien identifiés. Le problème de la vitesse de compilation peut donc être abordé sous plusieurs angles différents: approches maté-

rielles par accélérateurs dédiés, parallélisation du processus, réutilisation des résultats de compilations précédentes, limitation de ce qui doit être recompilé, etc ... Ces stratégies sont souvent orthogonales et peuvent être combinées au besoin.

Les deux chapitres suivants feront référence à certains concepts présentés dans cette revue de littérature. Le chapitre 2 traitera d'un serveur de compilation, ce qui a été introduit à la section 1.6.3, alors que le chapitre 3 présentera des techniques de génération de code se rapprochant de celles présentées à la section précédente.

Chapitre 2

Serveur de compilation

Comme l'a montré le chapitre précédent, l'approche incrémentale est un des axes majeurs menant à une solution au problème de la compilation rapide. Ce chapitre présente un serveur de compilation pour le langage Modula-3 qui permettra de résoudre les problèmes exposés à la section 1.6.3 à propos du pré-processeur du langage C et des fichiers d'en-tête. Les concepts développés seront d'abord exposés suivis de certains détails de réalisation.

2.1 Approche proposée de serveur de compilation

2.1.1 Interfaces en Modula-3

Comme le serveur développé dans cette étude repose sur les interfaces, quelques détails sur celles-ci seront d'abord précisés. Les interfaces jouent un rôle fondamental

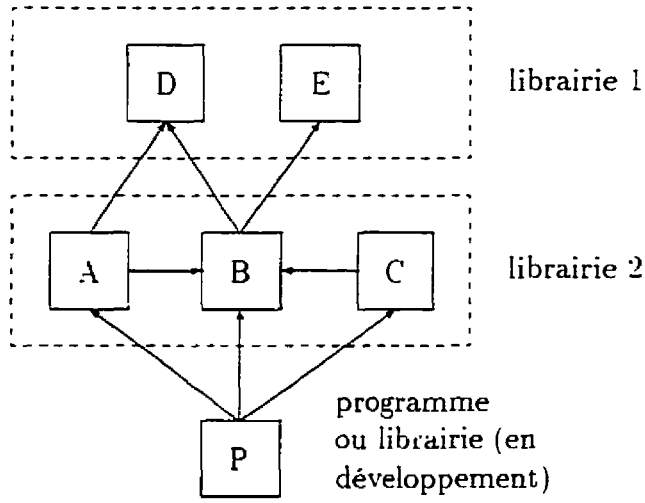


Figure 2.1: Dépendance des interfaces en Modula-3

en Modula-3. Elles permettent de déclarer les types, variables, exceptions et procédures qui peuvent être importés (utilisés) par les modules ou d'autres interfaces. Ce mécanisme est à la base de la programmation modulaire. Il permet d'établir une hiérarchie dans la structure d'un programme, ce qui est très utile lors du développement de logiciels de taille importante. De plus, les interfaces établissent une séparation entre une spécification et sa mise en oeuvre.

La figure 2.1 montre les relations de dépendance entre des interfaces en Modula-3. La notation est celle de Harbison [32] où une flèche va de l'interface ou module important vers l'interface importée (A importe B et D sur la figure). Comme la figure le montre, l'ensemble des importations pour un programme ou une librairie suit un graphe acyclique dirigé. Cette absence de cycles dans le graphe est une condition importante car les types déclarés dans une interface conservent leur définition

indépendamment des modules ou interfaces clients (qui les importent). De plus, le langage Modula-3 n'a pas de directives conditionnelles de compilation et chaque interface peut être compilée séparément pour mener à un fichier relocalisable distinct de ses clients. Par ailleurs, les déclarations d'une interface ne produisent jamais de code exécutable dans ce fichier relocalisable. Tout au plus, des espaces mémoires pour les variables pourront s'y retrouver ainsi que des instructions pour le débogueur. Ce mécanisme d'interface est donc beaucoup plus structuré que celui des fichiers d'en-tête en langage C discuté à la section 1.6.3.

Cette hiérarchie dans la dépendance des interfaces impose cependant un ordre dans la compilation des interfaces. Un module ou une interface ne peut être compilée que si toutes les interfaces importées ont été compilées précédemment. Pour le cas de la figure 2.1, D et E par exemple devront être compilés avant B qui devra lui-même l'être avant A et C. Par la suite seulement, P pourra l'être. Il arrive que l'ordre de compilation n'ait aucune importance. Ici, D peut être compilé avant ou après E par exemple. La même remarque s'applique pour A et C. Donc, une modification de l'interface B, suivie de sa recompilation, rend potentiellement invalides les fichiers objets associés à A, C et P. Un système de recompilation sélectif, tel qu'introduit à la section 1.5, peut déterminer si des recompilations seront effectivement effectuées.

La figure 2.1 montre également que des interfaces et modules se regroupent pour former une librairie ou un programme. Une librairie ou un programme se nomme *ensemble compilé* (*package*). Cette notion ne fait pas partie de la définition du lan-

gage mais est une particularité du système de compilation SRC-Modula-3. À chaque ensemble compilé correspond un fichier ".m3x". Pour un programme du nom de *prog*, par exemple, correspond un fichier *prog.m3x*. Ce fichier est généré et mis à jour par le compilateur à chaque compilation. Il contient l'information nécessaire au système de recompilation intelligent. Il est séparé en autant de sections qu'il y a de modules et d'interfaces dans l'ensemble compilé. Chaque section décrit les éléments déterminant le contexte de compilation. On nomme souvent le système de recompilation intelligent de SRC-Modula-3 *pré-éditeur de liens* (*pre-linker*).

Pour assurer la vérification statique des types entre l'ensemble présentement compilé et les librairies, le compilateur SRC-Modula-3 compilera certaines interfaces deux fois. La première sera pour compiler la librairie elle-même, ce qui est tout à fait normal. Cependant, les interfaces importées des librairies, lors de la compilation d'une autre librairie ou d'un programme, doivent également l'être pour fin de comparaison des types. Ainsi, même si la librairie standard libm3 a été compilée et placée dans un répertoire connu, la compilation d'une application demandera tout de même la recompilation des interfaces de libm3 importées directement ou indirectement par l'application.

2.1.2 Cache de compilation

Comme il en a été fait mention au chapitre précédent, conserver des interfaces en mémoire vive, de compilation en compilation, peut s'avérer une stratégie très

rentable du point de vue de la vitesse de compilation. Comme elles représentent les liens qui se tissent entre les modules, le compilateur leur fait souvent référence pour vérifier statiquement la cohérence des types. De plus, les interfaces sont généralement moins longues que les modules. Il semble donc raisonnable de conserver les résultats de leur compilation en mémoire, même pour une application de taille relativement importante, dans la recherche d'un compromis entre la vitesse de compilation et la quantité de mémoire requise pour la persistance. L'approche proposée est donc partiellement incrémentale et est décrite dans cette sous-section.

Les arbres syntaxiques abstraits du langage Modula-3 ont été décrits par Jordan [33]. En plaçant un tel arbre obtenu par compilation d'une interface dans une *cache*, ce résultat peut être réutilisé au moment de recompiler cette interface. Avec un tel système de persistance, le problème le plus important est de s'assurer que les arbres syntaxiques abstraits dans la cache soient valides. Il faut, dans le cas contraire, recompiler les fichiers sources d'origine.

L'algorithme de validation des éléments de la cache est celui de la figure 2.2. Il est basé sur des temps de modifications et de validations effectuées. Il s'agit d'une procédure récursive, *Rendre Valide*, qui est appelée avant d'utiliser un arbre syntaxique abstrait présent dans le cache. La procédure retourne une valeur booléenne affirmative si l'arbre syntaxique abstrait est toujours valide. Dans le cas contraire, une valeur négative est retournée ce qui implique que le compilateur devra recompiler le code source de l'interface. D'ailleurs, la variable locale *valide* est constamment mise à jour

```
1.  FONCTION RendreValide (VARIABLE interface: StructureEntreeCache)
    RETOURNE un BOOLEEN
2.      VARIABLE valide ← VRAI
3.      DEBUT
4.          SI interface = NULLE ALORS RETOURNER FAUX
5.          FIN du SI
6.          Pour chaque interface importée par interface. FAIRE
7.              valide ← valide ET RendreValide (importée)
8.          FIN du FAIRE
9.          SI valide est VRAI
10.             SI interface.temps_valide < temps_présent ALORS
11.                 SI interface.temps_m3x > interface.temps_valide ALORS
12.                     SI interface.temps_source > interface.temps_valide ALORS
13.                         valide ← FAUX
14.                     FIN du SI
15.                 FIN du SI
16.             FIN du SI
17.         FIN du SI
18.         SI valide est VRAI
19.             interface.temps_valide ← temps_présent
20.         AUTREMENT
21.             interface ← NULLE
22.         FIN du SI
23.         RETOURNER valide
24.     FIN de la FONCTION RendreValide
```

Figure 2.2: Algorithme de validation

dans la procédure pour indiquer la validité de l'arbre dans la cache. Le seul argument de la procédure est une structure de données (de type StructureEntreeCache) comportant quelques champs dont une référence à l'arbre syntaxique abstrait et quelques variables permettant d'évaluer la validité de ce dernier. La cache elle-même est une table qui contient des références vers ces structures de données. À chacune de ces structures correspond un fichier source d'une interface.

Le cas le plus simple qui peut amener un rejet par l'algorithme de validation est celui où l'arbre syntaxique abstrait n'est pas dans la cache (lignes 4 et 5). Dans ce cas, la variable *interface* est nulle et il est inutile de poursuivre l'algorithme. Une seconde possibilité qui peut faire qu'une interface est invalide est l'importation d'une ou plusieurs interfaces qui sont elles-mêmes invalides. Pour vérifier ce cas, la procédure *Rendre Valide* est appelée récursivement pour les interfaces importées (lignes 6 à 8). On "remonte" ainsi l'arbre d'importation des interfaces de la figure 2.1. Éventuellement, certaines interfaces (D et E) n'en importent aucune autre, ce qui arrête la récursion et permet la validation du graphe acyclique dirigé des feuilles vers le tronc. Si une interface importée est invalide, la présente interface le devient elle-même (la variable *valide* devient fausse).

Si l'interface est toujours valide (ligne 9) après vérification des interfaces importées, l'algorithme se poursuit avec une autre vérification (lignes 10 à 16). Par ailleurs, si le fichier source de l'interface a été modifié après que l'arbre syntaxique abstrait ait été placé dans le cache, il faut rendre invalide ce dernier, ce qui est réalisé par les lignes 12 à 14. Ici, *interface.temps_source* représente le temps de la dernière modification du fichier source. Cette valeur est généralement obtenue par un appel au système dont la commande n'est pas détaillée sur la figure. La variable *interface.temps_valide* indique le moment de la dernière vérification de la validité de l'interface dans le cache.

Quelques détails sur ce moment de dernière validation et sur le rôle des lignes 10 et 11 méritent une explication approfondie. Pour obtenir la valeur de *interfa-*

ce.temps_source. le système d'exploitation prendra un certain temps puisqu'il faut obtenir le temps de la dernière modification du fichier. ce qui nécessitera un accès au disque dans bien des cas. Si ce disque est situé sur un autre ordinateur dans un réseau local. ce temps est encore plus important. Dans le but de minimiser ces accès. l'algorithme n'y a recours que le moins souvent possible. Il faut ici se rappeler qu'une interface peut être importée par plusieurs interfaces et ou modules. Cette partie de l'algorithme peut donc être réévaluée plusieurs fois pour la même interface au cours d'une même compilation d'un programme ou d'une librairie. Cependant. un seul appel au système pour déterminer *interface.temps_source* est nécessaire. D'ailleurs. une seule vérification de la validité de l'interface est nécessaire. Pour éviter ces répétitions de vérifications. le compilateur note le temps du début de la compilation et l'inscrit dans la variable *temps_présent*. Cette variable reste cependant à une valeur constante pour la durée de la compilation en cours. Ainsi. si une validation positive de l'interface a eu lieu lors d'une compilation précédente. la variable *interface.temps_valide* est nécessairement inférieure à *temp_présent* à la ligne 10. Par contre. à cette même ligne. au cours d'une autre vérification durant la même compilation. cette condition ne sera pas rencontrée puisqu'à la ligne 19. à la vérification précédente. *interface.temps_valide* a été ajustée à *temps_présent*. Cette ligne 19 marque ainsi le temps de la dernière validation d'une interface dans le cache. Comme *temps_présent* reste constant pour une même compilation. la ligne 10 évitera de revalider une interface déjà validée par une précédente évaluation de l'algorithme dans sa partie la plus coûteuse en terme de

temps. soit aux lignes de 12 à 14 (accès au disque possible).

La ligne 11 pose une autre condition qui peut aussi éviter l'évaluation des lignes de 12 à 14. Ici. *interface.temps_ m3x* représente le temps de la dernière modification au fichier ".m3x" associé à l'interface considérée. Il faut se souvenir que ce fichier est mis à jour dès qu'une librairie ou programme est recompilé. Le fichier ".m3x" est donc plus récent que toutes les interfaces compilées pour lesquelles il détient les informations sur le contexte de compilation. Dans la très grande majorité des cas. plusieurs interfaces sont rattachées à un même fichier ".m3x". On peut penser ici aux librairies libm3 et m3core qui contiennent environ 200 interfaces susceptibles d'être importées par un programme. Il est donc plus efficace d'obtenir d'abord le temps de modification du fichier ".m3x" et de le comparer à *interface.temps_ valide*. Il n'y aura qu'un appel de système pour valider potentiellement un grand nombre d'interfaces. Ce cas est fréquent puisque plusieurs interfaces de la cache proviennent de librairies standards très peu souvent modifiées. Néanmoins. il peut arriver qu'une librairie A a été recompilée alors que seule une interface ou un module ait été édité. Le fichier ".m3x" est alors mis à jour. Les interfaces de la librairie qui n'ont pas été modifiées ont encore un arbre abstrait syntaxique présent dans le cache qui est valide. Conséquemment. on ne peut compter. dans ce cas. sur le temps de modification du fichier ".m3x" pour valider un programme ou une librairie B qui importe des interfaces de A. On doit connaître le temps de modification de chaque interface de A utilisée par B et tester la validité de l'arbre syntaxique abstrait correspondant à chacune.

Autrement dit, la condition de la ligne 11 est vrai et le code des lignes 12 à 14 est évalué.

Tel qu'expliqué précédemment, si l'arbre abstrait présent dans la cache est valide après la ligne 17, il faut noter le temps de la dernière validation (ligne 19) pour éviter de refaire une partie de l'algorithme de validation inutilement. Si au contraire la validation est négative, l'arbre est retiré de la cache (ligne 21). Il ne reste plus qu'à retourner le résultat de la vérification à la procédure appelante (ligne 23).

Lorsque la procédure *Rendre Valide* retourne une valeur négative, l'interface doit être recompilée et celles qui l'importent devront également l'être lorsque l'algorithme évaluera leur validité. Ainsi, plus une interface invalide est près des feuilles dans le graphe d'importation, plus le nombre d'interfaces qui en dépendent risque d'être élevé et plus l'effort de recompilation risque d'être important. Néanmoins, le majorité des interfaces les plus profondes dans l'arbre appartiennent à des librairies très stables n'ayant aucunement besoin d'être recompilées dans le cadre du développement d'une application. Au contraire, celles qui risquent de l'être plus souvent sont celles du présent ensemble compilé. De plus, le système de recompilation intelligent ne s'applique qu'aux modules ou interfaces de l'ensemble compilé considéré. Il ne peut permettre d'éviter la recompilation des interfaces importées des librairies pour la vérification statique des types.

Il faut préciser un point dans cette stratégie de serveur de compilation. Les interfaces génériques (".ig") n'ont pas d'arbre syntaxique abstrait associé maintenu dans la

cache. Les interfaces et modules génériques sont programmés en fonction d'un type T non encore connu. Ce code a l'avantage de pouvoir être compilé pour différents types. On peut, par exemple, avoir une liste chaînée générique qui pourra être compilée pour une liste chaînée de réels, d'entiers, de caractères, etc... Il est cependant quelque peu compliqué de maintenir dans la cache un arbre abstrait pour une interface dont le type n'est pas encore déterminé. Ce problème est tout de même mineur puisque les librairies génériques standards sont souvent instanciées pour les types les plus courants. Ces instances deviennent alors des interfaces qui peuvent mener à des arbres syntaxiques abstraits pouvant être placés dans la cache. De plus, les modules et interfaces génériques ne constituent généralement pas une portion très importante d'un code source.

L'avantage majeur du serveur proposé est l'indépendance du contenu de la cache par rapport à son utilisation. Ainsi, l'utilisation d'abres syntaxiques abstraits de la cache pour la compilation d'un ensemble compilé P n'empêche pas leur réutilisation pour la compilation d'un autre ensemble compilé Q. La validité d'un arbre syntaxique abstrait repose uniquement sur le fichier source correspondant et la validité des interfaces importées. Au contraire, en langage C, la validité d'un fichier d'en-tête peut dépendre des fichiers sources qui l'utilisent. Toutes les approches de serveurs de compilation ou de pré-compilation doivent tenir compte de ce contexte d'utilisation dans la validation de la cache, alors qu'ici le mécanisme d'importation permet d'établir plus simplement les dépendances entre les fichiers du code source. Aucune approche de ser-

veurs de compilation n'a été répertoriée pour des langages comme Ada ou Modula-2, qui offrent une hiérarchie semblable à Modula-3 dans l'organisation des interfaces ou des fichiers de définitions.

2.2 Réalisation du serveur de compilation

Cette section présente les modifications effectuées au compilateur SRC-Modula-3 pour en faire un serveur. La discussion se limite aux points les plus importants. Un recours à quelques simplifications est également nécessaire en quelques endroits.

La figure 2.3 montre la structure générale du compilateur SRC-Modula-3. Les rectangles représentent des programmes distincts alors que les ovales symbolisent des librairies. Les flèches en trait plein signifient l'importation de certaines interfaces d'une librairie, alors que celles en traits discontinus montrent qu'un programme est démarré par un appel de système ("exec" sous plusieurs systèmes). Ainsi, *m3build* est un programme qui utilise la librairie *quake*. Cette dernière peut appeler, par un appel de système, le compilateur SRC-Modula-3 (souvent surnommé "driver" et dont la structure sera expliquée plus loin dans cette section).

Le code de *m3build* est très simple et très court. Il précise certaines informations qui sont directement transmises à *quake*. *Quake* est en réalité un petit interpréteur qui fait l'analyse lexicale et syntaxique des fichiers "m3makefile". Ces derniers sont des fichiers qui accompagnent le code source et qui précisent, suivant une syntaxe simple, les fichiers à compiler, les options de compilation, les librairies à importer,

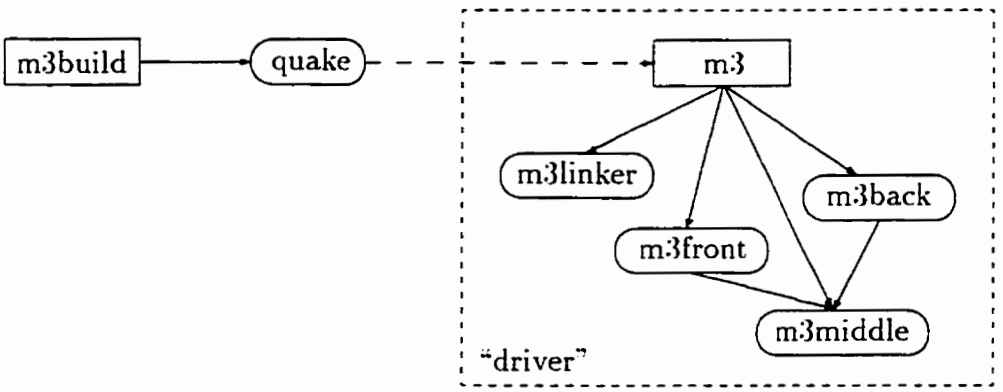


Figure 2.3: Structure simplifiée du compilateur SRC-Modula-3 standard

etc... Suivant cette analyse, *quake* appelle le compilateur en lui transmettant de nombreux paramètres. *Quake* joue un peu le même rôle que le programme *make* [29] sous Unix.

La figure 2.4 montre la structure du même compilateur après l'ensemble des modifications effectuées. On remarque deux changements majeurs. D'abord, le programme *m3build* a été remplacé par le programme *m3server* qui devient une interface du serveur de compilation pour l'extérieur. Le compilateur devient accessible par des programmes clients. De plus, le programme *m3* a été modifié pour devenir une librairie. *Quake* peut donc appeler le compilateur de manière procédurale, évitant ainsi un appel au système pour démarrer un processus indépendant. Quelques détails sur ces deux modifications suivent.

Le nouveau programme *m3server* conserve la fonctionnalité du programme *m3build*. Ainsi, le système de compilation peut continuer d'être utilisé comme auparavant sans changements visibles pour l'usager. La cache de compilation n'est alors

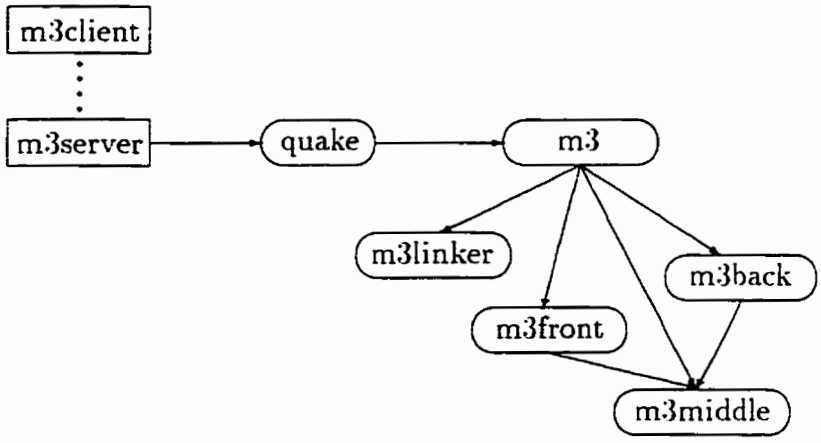


Figure 2.4: Structure simplifiée du compilateur SRC-Modula-3 modifié

pas maintenu. Cependant, deux options ont été ajoutées.

La première fait en sorte qu'une fois une première compilation terminée, l'usager est invité à préciser si une recompilation est désirée. Ainsi, si la première compilation a montré des erreurs dans le code source, le programmeur peut les corriger et relancer immédiatement une autre compilation. Il peut également préciser un autre programme ou librairie à compiler. Le dialogue se fait via une interface texte simple. Le compilateur devient ainsi un processus s'exécutant tant que l'usager redemande des compilations. La cache des arbres syntaxiques abstraits peut ainsi être maintenu de compilation en compilation.

La seconde option d'utilisation permet de démarrer en tant que serveur. Pour réaliser une relation client/serveur, les *objets réseaux* (*network objects* [14]) ont été utilisés. Un objet réseau est un objet dont les méthodes sont accessibles par un autre processus via un réseau d'ordinateurs. Donc, pour l'extérieur, *m3server* (et donc

```
INTERFACE M3Server;
IMPORT NetObj, Thread, Wr, Pathname, TextList;
EXCEPTION
  Error (TEXT);
TYPE
  T = NetObj.T OBJECT METHODS
    compile(init_dir: Pathname.T; options: TextList.T; writer: Wr.T)
  RAISES NetObj.Error, Thread.Alerted, Error;
  END;
END M3Server.
```

Figure 2.5: Objet réseaux servant d'interface au serveur

tout le système de compilation) se présente sous forme d'un objet qui est celui de la figure 2.5. Cet objet est constitué d'une seule méthode, *compile*, qui exécute la compilation d'un programme ou d'une librairie située dans le répertoire *init_dir* avec les options de compilation *options*. Un canal de sortie (*output stream*) *writer* doit également être fourni pour que le serveur puisse y acheminer les messages sur le déroulement de la compilation et les erreurs.

Un petit programme client de quelques lignes, *m3client*, a été développé pour accéder au serveur. Cependant, n'importe quel programme écrit en Modula-3 pourrait être modifié pour accéder l'objet réseau au besoin, y compris un éditeur ou un environnement de développement logiciel. Il est aussi à noter que même si plusieurs programmes clients tentent d'accéder au serveur au même moment, ils verront tous leur requête se réaliser mais le serveur effectuera chaque compilation une à la suite de l'autre.

La transformation de *m3driver* en une librairie était essentielle pour établir la cache de compilation. En effet, la cache est maintenu par une variable globale dans *m3*. Il ne fallait donc pas que le compilateur ("driver") termine avec la compilation comme c'était le cas lorsque démarré par *quake* par un appel de système. La structure de la figure 2.4 présente donc un système de compilation qui formera un seul processus s'exécutant en permanence pour ainsi assurer la persistance de la cache. Ce fait d'avoir un seul processus persistant qui redirige l'information à fournir à l'usager par le canal de sortie *writer* a nécessité des modifications importantes dans la gestion des erreurs dans tout le système de compilation. En effet, il est impératif d'éviter la terminaison du serveur pour des erreurs peu importantes tout en continuant d'informer l'usager de leur cause.

Une cache des arbres syntaxiques abstraits existait déjà avec le compilateur SRC-Modula-3. Les arbres étaient placés dans cette cache et pouvaient être réutilisés au cours de la même compilation d'un programme ou d'une librairie. La cache n'était cependant pas maintenu entre les compilations et les résultats des compilations précédentes étaient perdus. L'algorithme de validation présenté précédemment a été ajouté.

On pourrait qualifier *m3* comme étant la partie centrale du compilateur. La cache est maintenu à cet endroit bien que les arbres syntaxiques abstraits soient le résultat de l'analyse par la partie frontale du compilateur qui a lieu dans *m3front*. De nombreuses informations sont transmises à *m3* par *quake* au début de la compilation. Il appartient ensuite à *m3* d'organiser le déroulement du reste du traitement. Le sys-

tème de recompilation intelligent s'intègre au compilateur sous forme de librairie dans *m3linker*. Il assiste *m3* pour déterminer quels fichiers doivent être recompilés. Ceux qui doivent l'être sont confiés à *m3front* qui construit l'arbre syntaxique abstrait, et *m3back* (la partie finale) qui génère le fichier binaire correspondant. Dans *m3middle*, on retrouve des types et des procédures utilisés par toutes les parties du compilateur. En particulier, le langage intermédiaire y est défini sous forme d'un type objet. La variable associée est cependant créée et initialisé dans *m3back*. Cet objet est retourné à *m3* par une procédure juste avant de débuter la compilation d'un fichier. Il est alors passé immédiatement à *m3front* par *m3*. Les appels des méthodes de cet objet par *m3front* assurent la génération du code binaire qui a lieu dans *m3back*. Des modifications mineures ont été effectuées dans toutes ces librairies. Pour compléter la compilation, quelques autres programmes, comme l'éditeur de liens par exemple, sont nécessaires. Ils sont lancés soit par *quake* ou *m3* par des appels de système, bien que ces détails soient absents des figures précédentes.

La présente réalisation du serveur ne permet pas son utilisation dans un contexte multi-usagers. La sécurité et le partage de ressources seraient des points à évaluer avant d'étendre son utilisation à plusieurs usagers. De plus, aucune politique visant à retirer de la cache certaines interfaces n'a été envisagée. Il peut donc arriver qu'une interface placée dans la cache y demeure sans être réutilisée par la suite.

Chapitre 3

Génération de code

Dans ce chapitre, les techniques employées pour en arriver à une génération de code plus rapide sont exposées. Comme l'a souligné la revue de littérature, une approche de compilation rapide requiert souvent une simplification extrême de la partie finale. Ainsi, peu de nouveaux concepts théoriques sont développés pour cette partie du compilateur suivant cette approche. En revanche, le générateur de code doit faire face à de nombreuses contraintes sur le plan pratique (particularités du processeur, du système d'exploitation, etc...) Ce chapitre insistera donc sur les points majeurs de la réalisation du générateur de code. L'approche proposée ressemble à celle de Tanenbaum et al. [9]. Les distinctions de la présente approche par rapport à cette dernière seront soulignées au passage, de même qu'au chapitre suivant lors de la présentation des résultats sur la performance. Cependant, il convient d'abord de présenter quelques concepts généraux sur la génération de code. Ceux-ci sont tirés du chapitre 9 sur la

production de code du livre de Aho, Sethi et Ullman [18].

3.1 Problème de génération de code

Tel que mentionné précédemment, le rôle de la partie finale est de transformer le code intermédiaire en code exécutable. Ce problème se subdivise en plusieurs sous-problèmes et contraintes qui sont brièvement soulignées dans cette section.

Le langage intermédiaire produit par la partie frontale peut prendre des formes variées selon le compilateur. Il peut s'agir d'arbres syntaxiques abstraits ou d'arbres acycliques dirigés. Le recours à la notation postfixée est aussi possible. L'utilisation de code à trois adresses, où toutes les instructions sont présentées sous la forme $x := y op z$, est aussi courant. Plus souvent, cette forme intermédiaire représente le fonctionnement d'une machine abstraite (virtuelle) comme une machine à pile. Cette représentation doit être traduite pour une machine réelle par le générateur de code. La table des symboles joue également un rôle dans la production du code puisqu'elle permet d'associer des noms de variables à des adresses mémoire. Pour des raisons d'efficacité, il arrive également que la partie frontale ait accès à certaines caractéristiques de la machine cible. Cette pratique permet de tenir compte, très tôt dans la compilation, de contraintes pratiques qui faciliteront la génération de code.

La machine cible est très souvent limitée par le nombre de registres disponibles pour les opérations. L'utilisation optimale de ces registres amène le problème d'assignation de registres au variables, qui est reconnu NP-complet. À ce problème se

superpose des contraintes du processeur ou du système d'exploitation, qui réservent l'utilisation de certains registres à des fins particulières comme un exemple dans ce chapitre le montrera. Des algorithmes d'optimisation, qui analysent le temps passé par une variable dans un registre, sont souvent utilisés pour améliorer une première assignation simple des registres.

L'utilisation judicieuse des instructions et des modes d'adressages du processeur est aussi un facteur de première importance dans la génération de code. Il est généralement souhaité que les instructions s'exécutent le plus rapidement possible et qu'elles occupent le moins d'espace possible dans le fichier binaire. Évidemment, le choix des instructions dépend directement de l'architecture cible.

Une préoccupation majeure du compilateur est la gestion de l'emplacement mémoire des variables. Pour les variables globales, puisqu'elles sont actives pour toute la durée de l'exécution du programme, une section de mémoire leur est réservée directement. Généralement, la table des symboles et la partie frontale travaillent à établir la liste des variables qui s'y retrouveront. Il en va autrement pour les variables locales, les paramètres de procédures, et les variables temporaires qui sont créées au cours de la compilation pour conserver un résultat partiel. Le générateur de code gère généralement la mémoire qui leur sera assignée. Ces variables sont regroupées dans ce qui est appelé un enregistrement d'activation (*stack frame*) pour chaque procédure. Un enregistrement d'activation représente donc l'espace mémoire requis pour qu'une procédure puisse s'exécuter. La procédure appelante y laisse également son adresse

de retour, pour continuer son exécution après l'appel de la procédure courante. La gestion de la mémoire pour les enregistrements d'activation peut se faire suivant deux méthodes, l'allocation en pile et l'allocation statique. Avec la première, un nouvel enregistement est créé à chaque appel d'une fonction. Il est placé sur une pile et est dépilé lorsque la procédure termine. Avec l'allocation statique, l'espace mémoire de chaque enregistrement est déterminé à la compilation et est le même pour chaque appel d'une procédure donnée.

Le langage Modula-3 n'utilise que l'allocation en pile pour gérer les enregistrements d'activation. Un exemple plus loin dans ce chapitre montrera la création d'un de ceux-ci pour une procédure simple. De la mémoire pour des variables peut également être allouée dynamiquement. Cependant, cette gestion de la mémoire fait intervenir d'autres parties du système de compilation que ceux introduits dans le cadre de cet ouvrage.

Il a été mentionné depuis le début de ce mémoire que la partie finale génère du code pour la machine cible. Il faut cependant ajouter qu'il y a, en général, trois types de fichiers binaires. Il y a le fichier relocalisable, qui est produit par la partie finale et le fichier exécutable produit par l'éditeur de liens. Le troisième n'a pas encore été introduit. Il s'agit du fichier objet partagé. Il réside en mémoire comme le fichier exécutable mais son code est utilisé par plusieurs processus de manière à éviter que chacun de ces derniers en ait une copie identique. De la mémoire vive peut ainsi être économisée à l'exécution. De plus, de l'espace disque est aussi épargné puisque la

taille des fichiers exécutables peut être réduite. Le générateur de code peut jouer un rôle dans la formation des fichiers objets partagés. Il en sera question vers la fin de ce chapitre.

Pour terminer, il faut souligner qu'on exige d'abord de la partie finale qu'elle produise un code exempt d'erreurs. La testabilité d'un tel code peut représenter une portion très significative du temps de développement puisqu'il y a plusieurs cas à considérer.

3.2 Refonte de la génération de code

Cette section décrit l'approche de génération de code rapide réalisée pour accélérer le compilateur SRC-Modula-3. Une description du problème, et la solution proposée, seront l'objet de la première sous-section. Un exemple de génération de code pour une courte procédure, et une présentation de la structure du générateur de code, suivront. Des aspects particuliers du système Linux ont dû être considérés pour la nouvelle méthode de génération de code. La fin du chapitre leur est consacré.

3.2.1 Approche proposée

Le compilateur SRC-Modula-3 actuel est constitué d'une partie frontale écrite en Modula-3 et d'une partie finale basée sur GCC (*GNU C Compiler*) [44] pour produire le code objet. GCC est un compilateur pour les langages C, C++ et Objective-C. Il est disponible gratuitement et est porté sur de très nombreuses architectures. En réalité,

un fichier a été ajouté à la partie finale de GCC pour interfacer le langage intermédiaire de Modula-3 à celui de GCC. La partie finale modifiée de GCC assure la portabilité de Modula-3 sur plusieurs plates-formes. La contrepartie à un système aussi flexible est une vitesse de compilation passablement réduite. Dans un tel contexte, une partie finale comme GCC doit prévoir une méthode de génération de code pour tous les types différents de processeurs, de systèmes d'exploitation, pour différentes configurations, etc... Les parties frontales et finales correspondent également à deux passes distinctes qui communiquent entre elles via un fichier, ce qui est moins efficace.

Ce qui est proposé ici est de remplacer le générateur de code actuel basé sur GCC par un générateur de code intégré, rapide et peu configurable pour améliorer le temps de compilation. Ce nouveau générateur de code utilise des algorithmes simples et n'a pas recours à un assembleur à l'étape finale. Il est intégré à la partie frontale pour former une passe avec elle, court-circuitant la communication par fichier entre ces deux parties du compilateur. De plus, il est dédié à une plate-forme unique, soit une architecture de type Intel386® [3] opérant le système d'exploitation Linux. Bref, il s'agit du générateur de code le plus simple pouvant être conçu.

Comme point de départ, un générateur de code existant a été retenu, celui utilisé pour l'architecture Intel386 mais pour le système d'exploitation Windows-NT®. Ce générateur avait dû être développé précédemment puisque GCC n'était pas encore porté pour ce système d'exploitation, rendant ainsi nécessaire le recours à une partie finale particulière. Ce générateur de code est écrit en Modula-3 et comporte envi-

ron 11 300 lignes de code. Les modifications apportées à ce générateur de code pour l'adapter au système Linux ont cependant été importantes. Pour situer comment et à quels endroits elles ont eu lieu. un exemple visant à donner un aperçu du langage intermédiaire Modula-3 est d'abord exposé.

3.2.2 Exemple de génération de code

La figure 3.1 *a*) montre une partie d'un code source Modula-3. Il s'agit simplement d'une procédure qui retourne le résultat de l'addition de deux entiers i et j passés en paramètres. En *b*). on peut voir la description correspondante en langage intermédiaire. Le langage intermédiaire est défini par un ensemble de méthodes d'un objet du point de vue de sa programmation. On voit donc les méthodes appelées dans une écriture simplifiée.

Lorsque la procédure débute. la méthode *begin_procedure* est appelée. En argument. on passera des renseignements sur cette procédure (nom. localisation. symbole associé. nombre de paramètres. etc. . .) Le générateur de code produira alors systématiquement en *c*) le code binaire des 6 premières instructions présentées ici sous format assembleur AT&T ® [5]. L'appel de cette méthode permet de générer les instructions nécessaires pour sauvegarder certains registres contenant des valeurs appartenant à la procédure appelante et à en ajuster certains autres à des valeurs précises. ce qui constitue le prologue d'une procédure. Certains autres détails sur ce prologue seront donnés tout au long de cette sous-section.

```
PROCEDURE Add(i, j:  INTEGER):  INTEGER =
  BEGIN
    RETURN i + j;
  END Add;
```

(a)

```
begin_procedure(...) ──┐pushl %ebp
                       │movl %esp,%ebp
                       │subl $0x4,%esp
                       │pushl %ebx
                       │pushl %esi
                       │pushl %edi
load(...)              │
load(...)              │
add(...)  ─────────────┐movl 0x8(%ebp),%edx
                       │addl 0xc(%ebp),%edx
exit_proc(...) ────────┐movl %edx,%eax
end_procedure(...) ────┐popl %edi
                       │popl %esi
                       │popl %ebx
                       │leave
                       │ret
```

(b) (c)

Figure 3.1: Compilation d'une procédure simple.

La figure 3.2 montre que ces instructions du prologue construisent en réalité l'enregistrement d'activation pour la procédure. Le registre ebp (*base pointer*) sert de référence à l'enregistrement d'activation. Il doit être sauvegardé lors de la création d'un enregistrement d'activation précédent. L'utilisation du mode d'adressage indexé permet d'atteindre chacun des mots de 4 octets par déplacement (*offset*) par rapport à la valeur de ce registre. La procédure appelante y a précédemment placé les valeurs des paramètres actuels i et j en plus de l'adresse de retour de l'endroit d'appel.

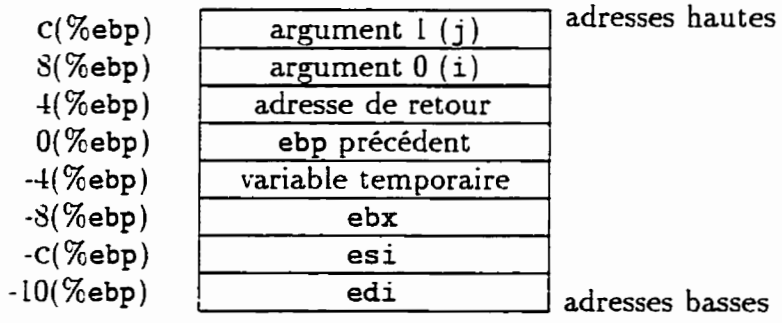


Figure 3.2: Enregistrement d'activation

Rigoureusement, les paramètres i et j appartiennent à l'enregistrement d'activation précédent alors que les valeurs suivantes appartiennent à celui présent. Le prologue est construit conformément à un standard, le *System V Application Binary Interface–Intel386 Architecture Processor Supplement* [35]. Ce standard, qui sera appelé ABI dans la suite du texte, précise également d'autres conventions concernant les fichiers binaires et la génération de code.

Par la suite, la méthode *load* est appelée à deux reprises, une fois pour chacun des paramètres i et j, bien qu'aucun code binaire ne soit généré. Ce que montre ce mécanisme est que le langage intermédiaire décrit le fonctionnement d'une machine à pile simple. La pile ainsi maintenue par le générateur de code est souvent qualifiée de virtuelle parce qu'elle n'amène aucune production directe de code mais permet une gestion interne des opérandes. Puisqu'on désire additionner les deux opérandes de la pile, la méthode *add* est appelée. Par ailleurs, le processeur exige qu'au moins un des deux opérandes de l'instruction machine `addl` soit situé dans un registre. Par conséquent, la valeur de i sera placée dans le registre `edx` par l'instruction `movl`. On

suppose que le choix de ce registre plutôt qu'un autre est sous la responsabilité de l'algorithme d'assignation de registres qui ne sera pas détaillé ici. L'instruction `addl` pourra ensuite être générée. Le résultat de l'addition est alors le seul opérande de la pile virtuelle. Comme la procédure doit retourner ce résultat, la méthode *exit_proc* est appelée. Une instruction `movl` transfère alors le résultat du registre `edx` au registre `eax`. La procédure appelante aura la responsabilité d'aller y chercher le résultat. C'est une convention de retour de procédure. L'appel de la méthode *exit_proc* rend également vide la pile virtuelle. Lorsqu'il y a affectation d'une variable dans le code source, la méthode *store* est appelée au lieu de *exit_proc*. Enfin, l'appel de la méthode *end_procedure* permet de générer les instructions de l'épilogue de procédure qui effectuent l'inverse de ce qui se fait pour le prologue. Il en résulte l'élimination de l'enregistrement d'activation courant.

Le code généré à la figure 3.1 est produit très rapidement et sans phase d'optimisation. Quelques instructions dans l'exemple sont générées inutilement. L'ABI précise que les registres `ebx`, `esi` et `edi` appartiennent à la procédure appelante. C'est pourquoi ils sont sauvegardés par la procédure appelée dans l'enregistrement d'activation. Cependant, la procédure *Add* n'utilise pas ces registres pour ses propres opérations. Leur sauvegarde est donc inutile dans le prologue de procédure. De même, le résultat de l'addition se retrouve dans le registre `edx` alors qu'il est transféré dans `eax` à l'instruction suivante. L'utilisation directe de `eax` à la place `edx` aurait suffi. De plus, avec l'instruction `subl $0x4,%esp`, le prologue réserve inutilement quatre octets pour une

variable temporaire. Cet espace était prévu au cas où la valeur retournée par la procédure nécessiterait d'y être placée pour une courte période de temps. Ce besoin n'est pas survenu dans la suite de la génération du code. On peut donc remarquer que le code est généré de manière conservatrice sans vérifier si certaines situations possibles à partir d'un certain point dans la génération se concrétiseront réellement par la suite. Une phase subséquente d'optimisation à lucarne pourrait facilement éliminer ce code inutile mais aux dépens de la vitesse de compilation.

Le générateur de code effectue quelques optimisations à lucarne faciles à réaliser lorsque c'est possible. Par exemple, l'appel de la méthode *load* pourrait directement amener un opérande dans un registre en produisant une instruction `movl`. Au contraire, la séquence de code intermédiaire *load–load–add* a été accumulée avant de produire un code machine correspondant. De cette manière, un seul registre, *edx*, a été utilisée au lieu de deux pour effectuer l'addition. Comme l'architecture Intel386 possède peu de registres à usage général, on peut considérer la portion de code générée comme étant optimisée. De plus, au lieu de produire deux instructions `movl`, une seule a été nécessaire, ce qui donne un code plus compact. Tanenbaum et al. [9] utilisent une stratégie semblable dans leur système.

3.2.3 Structure du générateur de code

La figure 3.3 montre la structure simplifiée du générateur de code modifié. La structure montre également qu'il pourrait y en avoir d'autres comme par exemple

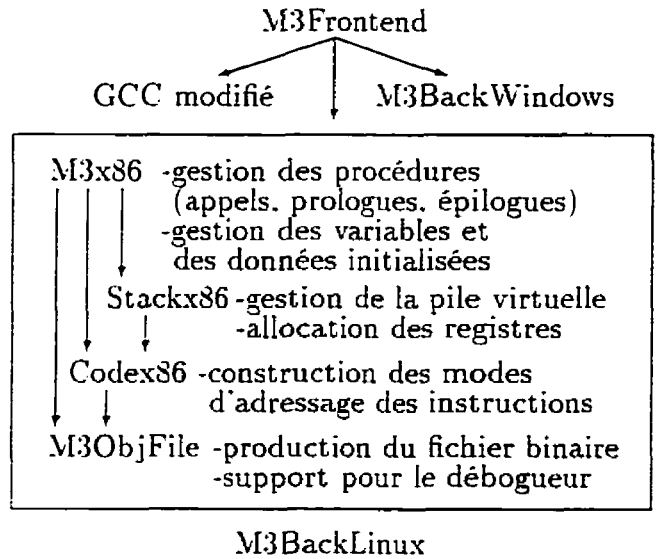


Figure 3.3: Structure du générateur de code.

celui basé sur GCC, ou celui pour Windows-NT. Le générateur de code intégré est constitué de 4 modules principaux. Généralement, chaque module met en oeuvre une seule interface qui elle-même décrit un seul objet T. Ainsi, le module Stackx86.m3, met en oeuvre l'interface Stackx86.i3 et un objet Stackx86.T. D'ailleurs, M3x86.T est un objet dérivé de M3CG.T, l'objet décrivant le langage intermédiaire. La figure 3.3 montre également les principales opérations mises en oeuvre dans chacun des modules. Les flèches, quant à elles, montrent les relations entre les objets principaux. Par exemple, M3x86.T a des champs de types Stackx86.T, Codex86.T et M3ObjFile.T.

L'exemple présenté plus haut sur la traduction en langage intermédiaire d'une procédure Modula-3 permet également d'illustrer le fonctionnement général du générateur de code. Ainsi, on peut retrouver la manière dont est généré le code pour les

prologue et épilogue de procédures dans M3x86.m3, puisque la gestion de celles-ci y est effectuée. Par contre, au moment de générer chacune de ces instructions machine, des méthodes de Codex86.T sont appelées pour former les modes d'adressage appropriés. Pour construire le code nécessaire à l'addition, des méthodes de Stackx86.T sont directement appelées depuis M3x86.m3 puisqu'il faut effectuer des opérations sur la pile virtuelle et disposer de registres du processeur. Par contre, encore ici, des méthodes de Codex86.T seront appelées depuis Stackx86.m3 pour régler les détails du mode d'adressage de chacune des instructions générées. Éventuellement, chacune des instructions formées dans Codex86.m3 sera inscrite dans le fichier binaire relocalisable en appelant des méthodes de M3ObjFile.T. Le fichier binaire doit également contenir d'autres renseignements que les instructions machines. C'est pourquoi certaines méthodes de M3ObjFile.T sont directement appelées à partir de M3x86.m3. Ces renseignements concernent principalement des instructions à fournir à un débogueur pour opérer convenablement, une table de symboles à fournir à l'éditeur de liens ainsi qu'une liste des données nécessaires à l'exécution du programme.

Le générateur de code de Tanenbaum et al. [9] a une structure très différente. Leur générateur de code est portable. Cette portabilité est assurée par la traduction du code intermédiaire via des tables contenant les instructions machines correspondantes à générer. Ces tables changent selon le processeur cible. Le générateur de code proposé ici a une structure n'offrant aucun support direct pour la portabilité. Il s'agit donc d'une approche plus directe de partie finale.

3.2.4 Modifications effectuées

Cette sous-section présente les changements apportés au compilateur pour opérer avec le système d'exploitation Linux. Le but n'est pas d'être exhaustif étant donné que certaines modifications n'ont impliqué que quelques lignes de code dans certaines procédures précises alors que d'autres ont été majeures. L'objectif est plutôt d'expliquer certains détails de fonctionnement du générateur de code dont la structure a été présentée à la section précédente, et de souligner les modifications les plus importantes au passage.

3.2.4.1 Formats de fichiers objets

Le générateur de code pour Windows-NT utilisait le format standardisé COFF (*Common Object File Format*) [30] pour les fichiers objets alors que sous Linux, le format ELF [34, 39, 37] doit être employé. La génération du fichier objet relocalisable a donc dû être refaite entièrement, ce qui implique une reprogrammation du module qui met en oeuvre l'interface M3ObjFile. Plus précisément, l'information brute que doit contenir le fichier objet est accumulée dans des structures de données de ce module. Lorsque la compilation se termine, cette information est mise en forme pour être inscrite dans le fichier. Le format ELF est beaucoup plus flexible et plus puissant que le format COFF.

La figure 3.4 montre la structure générale d'un fichier objet relocalisable ELF. On retrouve au début du fichier un en-tête ELF donnant des renseignements généraux

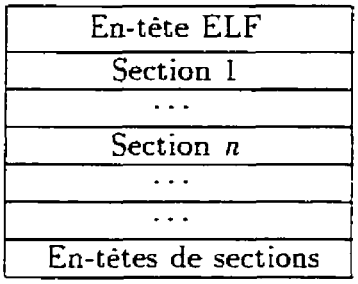


Figure 3.4: Fichier objet relocalisable ELF.

comme le type de processeur utilisé. Le coeur du fichier est constitué d'un vecteur de sections, chaque section regroupant un type d'information particulière. Pour chaque section, un en-tête correspondant est nécessaire. Dans cet en-tête de section, on retrouve des renseignements comme la position (mesurée en nombre d'octets depuis le début du fichier) du début de la section, sa longueur et certaines caractéristiques. L'ensemble des en-têtes de sections peut être placé n'importe où dans le fichier puisqu'un champ dans l'en-tête ELF donne sa position. On les place souvent à la fin tel que montré sur la figure 3.4. Par rapport au format COFF, ELF permet un nombre illimité de sections, d'où sa souplesse. De plus, on peut facilement établir des relations entre des sections, au besoin en ajustant certains champs dans les en-têtes de section. Au contraire, avec COFF, on a une structure rigide préétablie entre les sections.

Parmi les sections possibles, certaines sont d'un intérêt particulier. La section nommée ".text" contient le code binaire exécutable. La section ".data" contient les données initialisées du module ou de l'interface ce qui peut être, par exemple, de l'espace réservé pour des variables globales initialisées, des adresses (relatives au début

de la section) de procédures, des chaînes de caractères, etc... Les variables globales non initialisées sont placées dans la section ".bss". Ce qui change entre ELF et COFF pour ces sections est surtout les en-têtes de section. Par contre, ELF permet une variante de la section .data appelée ".rodata" qui permet de stocker des variables protégées contre l'écriture par le système, ce qui est parfois utile.

Une section particulièrement importante pour l'éditeur de liens est celle de la table des symboles (section ".symtab"). Un symbole représente une adresse mémoire qui sera éventuellement fixée par l'éditeur de liens. Par exemple, il arrive fréquemment que des instructions de la section .text accèdent des données de la section .data. Cependant, ces deux sections seront chargées en mémoire à des adresses virtuelles non encore déterminées. C'est pourquoi il faudra aller modifier des adresses inscrites dans la section .text lorsque l'adresse du début de la section .data sera connue. Ce processus est appelé relocalisation. L'ensemble des relocalisations à effectuer dans une section est regroupé et placé dans une section séparée. Par exemple, la section ".reloc.text" contient les relocalisations à effectuer dans la section .text. Chaque relocalisation est constituée de 3 champs soit l'endroit où doit être effectuée la relocalisation dans la section, le type de relocalisation et par rapport à quel symbole. Le début de la section .data a un symbole du nom de *MM_Main* qui représente son adresse, et qui est souvent utilisé comme référence pour les relocalisations. Il faut voir aussi que des relocalisations peuvent être nécessaires pour d'autres sections que la section .text. De plus, il est possible de relocaliser par rapport à d'autres symboles que *MM_Main*.

Les relocalisations avec ELF sont de plusieurs types, certaines étant assez compliquées. Cependant, elles sont plus puissantes qu'avec COFF et permettent parfois de simplifier les mécanismes de génération de code.

3.2.4.2 Support pour le débogueur

Pour qu'un débogueur puisse suivre le déroulement d'un programme, il faut lui fournir des informations particulières car il y a généralement une perte de représentation de la logique entre la description présente dans le code source original et celle binaire. Ainsi, une variable appelée *var_1* dans le fichier source n'est souvent plus qu'une adresse dans le fichier binaire. Donc, si on désire demander au débogueur la valeur de *var_1*, il doit pouvoir associer un nom à une adresse. Le débogueur aura aussi besoin de savoir les portions de code binaire associées à chaque ligne de code source (tel que le suggère la figure 3.1). La connaissance des types, des procédures et de leurs paramètres sera aussi nécessaire.

Cette information est souvent décrite suivant une syntaxe appelée "Stabs" [12] et qui, avec ELF, est contenue dans deux sections, ".stab" et ".stabstr". Le fait de placer cette information dans deux sections séparées et indépendantes permettrait d'utiliser complètement une autre syntaxe sans modifier les structures ELF. Avec COFF, la table des symboles contient aussi de l'information utile au débogueur. Comme aucun support pour le débogueur n'était offert avec le générateur de code pour Windows-NT, ce support a dû être ajouté pour le nouveau générateur de code. Un fait important à

noter est que Modula-3 utilise une variante non standard du format Stabs qui n'est comprise entièrement que par le débogueur m3gdb, une version modifiée du débogueur GNU gdb [10]. Cette variante vient probablement du fait que les ajouts nécessaires à Stabs pour comprendre les types des langages orienté objet n'étaient pas encore en place lorsque le compilateur a été mis au point. La différence la plus significative par rapport au Stabs standard vient du fait que les noms de types et de variables contenus dans certains champs sont étendus pour inclure des informations sur le type lui-même.

3.2.4.3 Librairies partagées

Tel que mentionné dans la première section de ce chapitre, il est souhaitable que le code d'une librairie soit partagé par plusieurs processus pour éviter qu'il s'en trouve plusieurs copies en mémoire. Ce partage est possible avec le support du noyau (*kernel*) du système d'exploitation, de l'éditeur de liens et du chargeur (*loader*). Le format ELF précise que le compilateur doit aussi participer au processus.

Le problème à la base de la réalisation d'un système de librairies partagées est la position du code en mémoire. Généralement, une portion de la mémoire virtuelle est réservée au code des librairies partagées. Si la librairie occupe une position fixe en mémoire, comme c'est le cas avec COFF, l'éditeur de liens produits les fichiers exécutables et partagés à partir des fichiers relocalisables de manière à ce que toutes les références inter-modules soient déterminées avant l'exécution. La plupart du temps,

une table contenant les points d'entrée des procédures sert d'interface à la librairie. Les adresses que maintient cette table sont cependant fixes. Le problème d'une telle organisation est qu'il peut y avoir un chevauchement d'une zone de mémoire occupée par deux librairies. ce qui nécessite une repositionnement des adresses.

Une solution à ce problème avec ELF est d'avoir la possibilité de charger une librairie à n'importe quel endroit dans la zone de mémoire partagée. Ainsi. une librairie est placée par le système à une position qui évite les conflits de position en mémoire. Il peut alors arriver que pour deux exécutions différentes du même programme. la librairie partagée occupe deux positions différentes. La librairie demeure cependant à une position fixe pour la durée de son chargement en mémoire. La contrepartie à cette solution est qu'il faut généralement proscrire les références mémoire absolues dans le fichier partagé. ce qui rend le système plus flexible mais aussi légèrement plus complexe. Le compilateur devra alors générer du code indépendant de la position (*Position Independent Code ou PIC*).

En réalité. les références absolues sont conservées dans une table. la *Global Offset Table* ou *GOT*. Une librairie partagée a sa propre table indépendante de celles des processus qui lui font référence. Pour les points d'entrées de procédures. une autre table est maintenue. la *Procedure Linkage Table* ou *PLT* qui interagit cependant avec la GOT.

Les valeurs des symboles seront déterminées et placées dans la GOT pour compléter les références au moment de l'exécution juste avant que le contrôle ne soit transféré

au programme. Les références aux adresses de procédures d'une librairie partagée sont cependant complétées au moment de l'appel de la procédure. Cette édition de liens est donc dynamique (*dynamic linking*) puisqu'elle s'effectue tout au long de l'exécution d'un programme. Comme un programme n'exécute souvent qu'une partie de tout le code possible. les relocalisations évitées se transforment en un gain de performance puisque le système n'a consacré aucun temps à les effectuer.

Les adresses des GOT ne peuvent être elles-mêmes à des positions fixes en mémoire car le code leur fait référence. Pour contourner ce problème. on a recourt à un *registre de base* qui est **ebx** pour l'architecture Intel386 selon les spécifications de l'ABI. Ce registre doit toujours contenir l'adresse de la GOT courante. Une relocalisation spéciale est requise pour ajuster la valeur de **ebx** à chaque prologue de procédure. L'usage du mode d'adressage indexé par rapport à ce registre permet de faire référence aux entrées dans la GOT.

Des modifications mineures ont été effectuées au générateur de code pour lui permettre de générer du code indépendant de la position au besoin. Par exemple. les appels de procédures ont été modifiés dans M3x86.m3 pour tenir compte de la PLT. L'algorithme d'assignation des registres dans Stackx86.m3 a subi des modifications pour tenir compte du rôle particulier du registre **ebx**. Les modes d'adressages dans Codex86.m3 ont été retravaillés pour permettre des déplacements par rapport à **ebx** au besoin. Bien d'autres portions de code ont été reprogrammées pour des détails semblables.

```
movl 0x44,%esi          movl 0x44(%ebx),%esi        movl 0x0(%ebx),%eax
                                                    movl 0x44(%eax),%esi
```

(a) (b) (c)

Figure 3.5: Accès aux variables globales externes

Il y a cependant un aspect des références aux variables globales qui a demandé des modifications majeures. La figure 3.5 montre trois cas différents d'un exemple d'une variable statique qui se situe 68 octets (0x44) du début d'une section .data et qui est transférée dans un registre (`esi`). Le premier cas (a) illustre du code pouvant faire des références absolues en mémoire avec un mode d'adressage direct. Une simple relocalisation effectuée par l'éditeur de liens permettra d'ajouter au déplacement 0x44 la valeur du début de la section .data pour avoir l'adresse exacte avant l'exécution. Le second cas montre que pour du code PIC, il faut utiliser un mode d'adressage indexé avec le registre **ebx** qui détient l'adresse de la GOT. Comme la GOT fait parti de la section des données, un simple ajustement de la valeur 0x44 par une relocalisation particulière permettra d'atteindre la position mémoire de la valeur désirée. Aucun mode d'adressage direct et aucune référence directe à la mémoire ne sont utilisés donc le code est PIC. Le cas (b) permettra facilement à un code d'atteindre les variables statique de son module à l'aide du registre de base **ebx**. Par contre, pour rejoindre des variables globales d'un module différent (ce qui est semblable au cas des variables "extern" en langage C), il faut procéder autrement. L'adressage dans ce cas se fait en deux temps (cas (c)). Tout d'abord, l'adresse de la section des données où se situe

cette variable est inconnue et doit être obtenue de la GOT et placée dans un registre (ici **eax**). Il faut encore ici utiliser un mode d'adressage indexé avec le registre de base **ebx**. Par la suite seulement, on peut procéder par déplacement par rapport au registre **eax** en mode d'adressage indexé pour rejoindre le champ de la variable désirée.

Les modifications effectuées au générateur de code pour supporter le cas (c) ont été importantes. Il faut passer l'adresse de la section des données sur la pile virtuelle, de manière à obtenir l'assignation d'un registre supplémentaire (**eax** dans ce cas-ci). De plus, la grande majorité des séquences d'instructions à générer sont susceptibles de manipuler des variables globales.

Chapitre 4

Évaluation de la vitesse de compilation

Les deux derniers chapitres ont exposé les techniques utilisées pour transformer un compilateur en un serveur de compilation et pour générer rapidement du code machine. Ce chapitre présentera les résultats obtenus lors de l'évaluation de leur performance. Les vitesses mesurées pour le serveur et pour le générateur de code feront l'objet de deux sections. Cependant, la méthode suivie pour mesurer les temps de compilation sera d'abord détaillée.

4.1 Méthodes d'évaluation de la performance

Pour avoir une idée de la vitesse du nouveau compilateur, ce dernier a été soumis à une série de programmes à compiler. La liste des programmes de test retenus sera

détaillée plus bas. La plupart du temps, les résultats obtenus seront comparés à ceux produits par le compilateur SRC-Modula-3 standard puisqu'il n'existe pas d'autres compilateurs Modula-3 pouvant servir de point de comparaison. Néanmoins, quelques facteurs d'accélération globaux seront tout de même comparés à leur équivalent dans quelques systèmes de compilation, lorsque les cas seront pertinents.

Pour mesurer les temps de traitement, le compilateur est doté de chronomètres intégrés. Ces derniers mesurent le temps réellement écoulé pour effectuer une étape. Autrement dit, il s'agit du temps passé par l'usager à attendre un résultat. C'est ce temps qui a de l'intérêt dans le cadre de cette étude et non, par exemple, le temps pris uniquement par le processeur.

Entre une requête de compilation et la fin de la production d'un fichier exécutable, le système de compilation aura consacré la majorité de son temps à effectuer 5 grandes tâches. Il y a d'abord le temps pris par le système de recompilation intelligent pour déterminer quels fichiers doivent être recompilés. Les trois étapes suivantes sont celles consacrées à la production des fichiers relocalisables. Il y a donc le temps pris par la partie frontale pour produire un fichier en langage intermédiaire (L.I. dans les tableaux) à partir du code source en langage Modula-3 (M3⇒L.I.) La partie finale de GCC effectue ensuite le passage du langage intermédiaire vers le langage d'assemblage (L.I.⇒ass.) Un assembleur permet dans une dernière phase d'obtenir les fichiers relocalisables (ass.⇒reloc.) Avec le générateur de code intégré, les parties frontales et finales ne forment qu'une seule phase. Dans ce cas, ces trois dernières étapes sont

Tableau 4.1: Ensembles compilés retenus pour l'évaluation de performance

ensemble compilé	taille	lignes	lignes*	interf.	modules	interfaces importées	memoire requise
columns	47.17K	1553	1306	6	7	30	2.26M
netobjd	3.87K	151	128	1	2	27	1.12M
webscape	11.72K	374	347	0	1	69	2.96M
m3browser	210.58K	7005	6312	12	13	67	1.93M
webvbt	194.76K	6254	5364	21	20	120	3.63M
m3tohtml	83.08K	3035	2656	8	9	35	1.26M
m3front	1.367M	45827	39789	175	171	38	3.06M
postcard	341.81K	10418	9575	12	11	161	3.93M
ps2html	315.85K	13468	9197	30	30	111	3.57M

regroupées en une seule notée M3⇒reloc. La dernière étape est celle de l'édition de liens.

D'autres étapes sont nécessaires au processus complet. Le temps de leur exécution sera regroupé sous la mention "autre" dans les tableaux. Ces étapes sont généralement beaucoup plus courtes que les précédentes. On peut y retrouver, par exemple, le temps pris pour détruire les fichiers intermédiaires entre les passes de compilation, ou le temps consacré à faire une vérification de la cohérence des types une fois les fichiers relocalisables générés. Dans le cas d'une librairie, on peut y inclure le temps nécessaire pour l'archiver. Pour un programme, le compilateur doit générer un fichier relocalisable du nom de _m3main.mo pour tenir compte de l'initialisation particulière des programmes Modula-3. Cette étape est généralement aussi très courte.

Le tableau 4.1 montre les ensembles compilés retenus pour les mesures ainsi que leurs caractéristiques. Ces caractéristiques sont d'abord le nombre d'octets du code

source ainsi que le nombre de lignes qui le composent. Le paramètre ligne* donne ce même nombre mais excluant les lignes vides. Les deux colonnes suivantes présentent respectivement les nombres de modules et d'interfaces qui constituent l'ensemble compilé. Le nombre d'interfaces, directement ou indirectement, importées ainsi que la quantité de mémoire requise par le serveur pour les représenter sous forme d'arbres syntaxiques abstraits sont également fournis. Les trois premiers ensembles compilés peuvent être considérés comme étant de petite taille, les trois suivants comme étant de moyenne grosseur et les trois derniers comme étant assez imposants. La diversité des ensembles quant à leur caractéristiques a été d'une importance de premier plan dans la sélection. Tous ces ensembles sont disponibles avec la distribution du compilateur SRC-Modula-3, sauf ps2html qui a été développé par Poirier et Dagenais [23].

Les 9 ensembles retenus ont été compilés sous diverses conditions. Cependant, à moins d'indication contraire, le code produit par un compilateur est toujours avec des instructions pour le débogueur (option "-g") et est toujours indépendant de la position (option "-fPIC"). De plus, les compilateurs ont été eux-mêmes compilés avec le compilateur SRC-Modula-3 standard version 3.6 dont le générateur de code, appelé m3cgc1, est basée sur celui de GCC. L'option d'optimisation "-O2" a été employée pour les compiler, ce qui représente la meilleure possible avec ce compilateur. L'ordinateur utilisé pour les tests est basé sur un processeur Pentium® de 75 MHz ayant 32 Mega-octets de mémoire vive.

Tableau 4.2: Compilation avec générateur de code m3cgc1

ensemble compilé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
columns	1.19	4.21	8.65	4.44	0.85	0.32	19.66
netobjd	0.78	0.87	1.50	0.85	0.43	0.32	4.75
webscape	3.48	2.56	4.06	1.51	1.50	0.91	14.02
m3browser	1.04	10.08	36.09	11.88	1.11	0.81	61.01
webvbt	4.80	14.17	37.52	16.83	0.97	2.03	76.32
m3tohtml	0.79	4.00	16.49	6.39	0.85	0.61	29.13
m3front	1.62	69.56	220.36	108.03	5.84	17.32	422.73
postcard	2.79	22.38	55.34	16.43	4.16	0.74	101.84
ps2html	3.43	28.24	84.38	27.34	6.02	1.48	150.89

Le tableau 4.2 présente les temps de compilation qui serviront de référence pour le reste de l'étude. Ils ont été obtenus avec un compilateur standard légèrement modifié. Ces modifications, au nombre de deux, ont d'ailleurs été également ajoutées au compilateur-serveur et au compilateur avec générateur de code intégré. La première a consisté à appeler directement l'éditeur de liens *ld* au lieu de passer indirectement par GCC, ce qui évite le chargement de ce dernier. Il s'agit d'une modification assez simple à réaliser. La seconde modification permet de faire générer le fichier _m3main.mo mentionné précédemment par le générateur de code directement. Auparavant, un fichier _m3main.c en langage C était généré et compilé par GCC. Le chargement de ce dernier est une fois de plus évité. Cette modification a été réalisée par un étudiant de l'École Polytechnique de Montréal, M. Louis-D. Dubeau.

4.2 Performance du générateur de code

La performance du générateur de code est abordée avant celle du serveur parce que la génération de code par m3cgc1 est l'étape qui demande généralement le plus de temps pour s'exécuter. Comme le démontreront les résultats des tableaux qui suivent, l'utilisation du générateur de code intégré a un impact considérable sur la performance globale du système de compilation.

Le générateur de code a été évalué sous quatre aspects différents. Le premier est évidemment la vitesse de compilation par rapport à celle de m3cgc1. La production de code indépendant de la position et la génération d'instructions pour le débogueur ont potentiellement un impact sur la vitesse de compilation. Ces facteurs ont aussi été évalués. En dernier lieu, la qualité du code produit a également été l'objet de mesures appropriées.

Le tableau 4.3 montre les temps obtenus pour les ensembles compilés retenus lorsque le générateur de code intégré est utilisé à la place de m3cgc1. La réduction des temps totaux de compilation est très importante dans tous les cas. On peut remarquer que pour un ensemble compilé qui demande environ 20 secondes et plus pour être compilé avec m3cgc1, le temps est au moins réduit de moitié avec le générateur de code intégré. Pour des ensembles compilés de taille plus importante comme m3front et ps2html, on atteint même une réduction des deux tiers. Dans ces derniers cas, la production des fichiers relocalisables par m3cgc1 à partir du langage intermédiaire domine largement le temps total de compilation. Le générateur de code intégré peut

Tableau 4.3: Compilation avec générateur de code intégré

ensemble compilé	Temps (en seconces)				
	recomp. intel.	M3 ⇒ reloc.	Éd. de liens	autre	total
columns	1.16	6.46	0.85	0.61	9.08
netobjd	0.77	1.24	0.43	0.39	2.83
webscape	3.98	2.93	1.69	1.14	9.74
m3browser	0.97	14.91	0.85	0.46	17.19
webvbt	4.52	21.60	1.01	2.02	29.15
m3tohtml	0.94	5.83	1.06	0.39	8.22
m3front	2.45	95.54	8.05	16.88	122.92
postcard	3.07	31.75	8.51	1.09	44.42
ps2html	3.62	35.73	4.77	1.28	45.40

effectuer le même travail en seulement quelques secondes, d'où l'importance de la réduction. Pour l'ensemble webscape, le temps pris par le système de recompilation intelligent occupe une proportion plus importante du temps total de compilation, d'où une réduction moins importante. Ceci s'explique par le fait que webscape est une très petit ensemble compilé (un seul module) mais importe un nombre relativement important d'interfaces (69), dont certaines appartiennent à des librairies graphiques de taille importante.

Tanenbaum et al. [9] ont évalué leur générateur rapide de code dans un contexte différent. Plutôt que de comparer leur compilateur amélioré par rapport à celui qu'il remplace, ils l'ont comparé à deux disponibles commercialement. De plus, l'évaluation était pour la compilation en langage C et sans la présence d'un système de recompilation intelligent. Néanmoins, ils avaient observé que le temps de compilation total pouvait être de deux à trois fois plus rapide avec leur compilateur, ce qui est semblable

Tableau 4.4: Compilation sans l'option "-g"

ensemble compilé	Temps (en seconces)				
	recomp. intel.	M3 ⇒ reloc.	Éd. de liens	autre	total
columns	1.11	5.30	0.64	0.75	7.80
netobjd	0.77	0.83	0.43	0.38	2.41
webscape	3.30	3.11	1.48	1.04	8.93
m3browser	0.92	10.97	1.06	0.44	13.39
webvbt	3.48	15.59	0.60	1.23	20.90
m3tohtml	0.85	4.45	0.65	0.78	6.73
m3front	2.15	76.10	3.46	7.20	88.91
postcard	2.69	21.18	1.49	0.93	26.29
ps2html	3.68	29.69	4.14	1.35	38.86

à ce qui est obtenu ici.

Ils ont aussi observé que la compilation de petits codes source peut être dominé par les temps de chargement et non par le processus de compilation lui-même. C'est ce qui peut expliquer les résultats un peu plus faibles pour un ensemble compilé comme netobjd.

L'expérience suivante est identique à la précédente sauf que l'option "-g" n'a pas été utilisée avec le générateur de code intégré. Par conséquent, aucune instruction n'est offerte dans les fichiers générés pour l'utilisation d'un débogueur. Tel qu'attendu, en évitant de générer ces instructions, les temps de compilation sont réduits, comme le montre le tableau 4.4, comparés à ceux du tableau 4.3. Ce qui est étonnant est que cette réduction est tout de même importante, soit entre 10% et 30% sur le temps total de compilation. Dans le cas de postcard, on atteint même 40%. On note cependant ici une réduction de 8.51 secondes à 1.49 seconde du temps de l'édition de liens. Des

tests effectués sur un ordinateur ayant 64 Mega-octets de mémoire vive au lieu de 32 ont montré que cet écart ne se produit pas. Il est normal que le temps d'édition de liens soit réduit puisqu'il y a moins de sections dans les fichiers relocalisables. Cependant, l'édition de liens elle-même en tant que traitement dépasse rarement les trois secondes. Par contre, réunir tous les fichiers nécessaires à cette opération demande beaucoup d'entrées/sorties sur un ordinateur. Donc, le seul fait de réduire la taille des fichiers relocalisables en ne produisant pas les sections "Stabs" a permis d'économiser suffisamment de mémoire pour avoir une réduction drastique du temps d'édition de liens pour ce cas particulier. Ce phénomène se reproduira pour le serveur de compilation comme il sera montré à la section suivante. Produire ces instructions pour le débogueur est néanmoins nécessaire puisque ce nouveau générateur de code s'emploiera surtout dans un contexte de mise au point d'un logiciel où un débogueur est couramment employé.

Comme il a été mentionné précédemment, le code généré par les compilateurs dans cette étude est indépendant de la position. Pour voir si du code à position fixe peut être généré plus rapidement, l'option "-fPIC" a été retirée en plus de l'option "-g". Les temps de compilation sont présentés au tableau 4.5. En les comparant à ceux du tableau 4.4, on remarque que les écarts sont faibles et se confondent pratiquement avec la variabilité du test. On note une légère réduction des temps de compilation sauf pour postcard. Donc, la production de code indépendant de la position n'amène pas un temps de compilation beaucoup plus important.

Tableau 4.5: Compilation sans les options "-g" et "-fPIC"

ensemble compilé	Temps (en seconces)				
	recomp. intel.	M3 ⇒ reloc.	Éd. de liens	autre	total
columns	1.18	5.48	0.64	0.36	7.66
netobjd	0.77	0.83	0.42	0.36	2.38
webscape	3.30	2.66	1.49	1.22	8.67
m3browser	0.86	11.00	0.85	0.39	13.10
webvbt	3.71	14.84	0.41	1.20	20.16
m3tohtml	0.83	4.57	0.64	0.53	6.57
m3front	2.25	73.61	3.72	6.70	86.28
postcard	2.75	21.52	1.48	1.46	27.21
ps2html	3.58	28.78	3.51	1.31	37.18

Le dernier point dans l'évaluation du générateur de code a concerné la qualité du code produit. Pour se faire une idée de cet aspect, on sélectionne généralement un programme à compiler avec et sans optimisation pour comparer les temps d'exécution de ce programme dans chaque cas. On cherche souvent un programme qui effectue un travail intense et varié. Devant les programmes Modula-3 disponibles, le compilateur lui-même a été retenu comme programme-test. Le compilateur utilisé pour générer les résultats du tableau 4.3 a été compilé avec un compilateur utilisant m3cgc1 avec l'option "-O2". Pour comparer, le compilateur-test avec générateur de code intégré a été recompilé avec un compilateur utilisant m3cgc1 mais sans l'option "-O2". Les résultats obtenus avec ce compilateur-test non optimisé pour la compilation des ensembles retenus sont fournis au tableau 4.6. Évidemment, le compilateur-test a aussi été recompilé avec le générateur de code intégré et utilisé pour produire les résultats du tableau 4.7. Comme le montrent les valeurs obtenues, le générateur de code intégré

Tableau 4.6: Compilation avec compilateur compilé sans optimisation

ensemble compilé	Temps (en seconces)				
	recomp. intel.	M3 ⇒ reloc.	Éd. de liens	autre	total
columns	1.53	7.17	0.85	0.49	10.04
netobjd	1.01	1.01	0.42	0.45	2.89
webscape	4.55	3.34	1.69	1.37	10.95
m3browser	1.11	17.23	0.85	0.50	19.69
webvbt	6.55	24.74	1.15	2.58	35.02
m3tohtml	1.11	7.01	0.64	0.47	9.23
m3front	2.78	112.43	9.28	18.74	143.23
postcard	4.32	37.13	8.40	1.52	51.37
ps2html	5.76	41.63	4.98	1.61	53.98

produit un code qui s'exécute en un temps qui est à peu près à mi-chemin entre ceux produits par m3cgc1 avec et sans optimisation. Donc, même en ayant comme objectif de produire un code rapidement et en sacrifiant une phase d'optimisation, on peut obtenir un code de qualité très raisonnable du point de vue de la vitesse d'exécution. Quant à la compacité du code, celui produit par m3cgc1 sans optimisation est environ 32% plus gros que celui produit avec optimisation. Ce chiffre est de 14% pour celui produit par le générateur de code intégré toujours par rapport à celui produit par m3cgc1 avec optimisation.

En résumé, le générateur de code intégré permet de réduire le temps de compilation dans sa partie la plus lente, soit celle de la génération de code. Il faut accepter un temps un peu plus long pour générer les instructions pour un débogueur mais non pour générer du code indépendant de la position. De plus, le code produit de manière rapide peut tout de même s'exécuter relativement rapidement.

Tableau 4.7: Compilation avec compilateur compilé avec générateur de code intégré

ensemble compilé	Temps (en seconces)				
	recomp. intel.	M3 ⇒ reloc.	Éd. de liens	autre	total
columns	1.43	6.58	0.85	0.42	9.28
netobjd	0.86	0.91	0.42	0.41	2.60
webscape	4.10	3.17	1.69	1.27	10.23
m3browser	0.92	15.95	0.85	0.58	18.30
webvbt	4.99	23.13	1.27	2.07	31.46
m3tohtml	0.98	6.41	0.64	0.38	8.41
m3front	2.48	104.66	8.28	19.24	134.66
postcard	3.36	33.98	7.52	1.21	46.07
ps2html	4.10	36.17	3.69	2.85	46.81

4.3 Performance du serveur de compilation

Le générateur de code intégré a permis de faire en sorte que la partie finale ne soit plus celle qui demande le plus de temps dans le processus de compilation. L'attention se porte alors vers la réduction du temps de l'analyse effectuée par la partie frontale à l'aide d'un serveur de compilation. Deux contextes présentent un intérêt particulier avec le serveur: celui où tous les fichiers d'un code source sont recompilés, et celui où quelques-uns seulement le sont.

Le contexte de recompilation de tous les fichiers d'un ensemble présente un peu moins d'intérêt puisqu'il se produit plus rarement. Son analyse peut tout de même révéler des points intéressants. Le tableau 4.8 montre les résultats obtenus pour la recompilation de tous les fichiers des ensembles sélectionnés. Pour réaliser cette expérience, l'ensemble a d'abord été compilé une première fois pour permettre au ser-

Tableau 4.8: Compilation avec compilateur-serveur et communication TCP/IP

ensemble compilé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
columns	0.63	1.96	9.04	4.45	0.86	0.26	17.20
netobjd	0.46	0.28	1.52	1.09	0.44	0.25	4.04
webscape	3.00	1.58	4.06	1.55	1.71	0.65	12.55
m3browser	0.63	8.90	37.19	12.24	1.55	0.40	60.91
webvbt	4.46	12.04	39.80	17.41	1.29	2.31	77.31
m3tohtml	0.41	3.29	16.40	6.90	0.86	0.37	28.23
m3front	1.39	77.84	235.90	121.02	34.70	31.18	502.03
postcard	2.48	21.67	57.73	18.35	7.32	0.72	108.27
ps2html	3.12	24.73	89.48	30.44	11.54	1.59	160.90

veur de placer dans la cache tous les arbres syntaxiques abstraits correspondant aux interfaces utilisées. Les fichiers relocalisables et exécutables ont alors été détruits manuellement et la compilation reprise. Cette façon de procéder permet normalement de tirer profit de la puissance du serveur. Les résultats obtenus montrent que les temps de compilation sont inférieurs à ceux du tableau 4.2 pour les petits ensembles mais que l'écart se rétrécit pour ceux de moyenne taille. Pour les plus gros, le serveur prend même plus de temps à s'exécuter.

Pour tenter d'expliquer ce problème, l'expérience avec le serveur a été reprise mais en évitant de communiquer avec ce dernier via les objets réseaux (et le protocole TCP/IP (*Transmission Control Protocol Internet Protocol*)). Le serveur a plutôt été exécuté directement et contrôlé par une petite interface texte simple. Les résultats présentés au tableau 4.9 montrent que la situation s'améliore quelque peu. Néanmoins, les ensembles de grande taille sont toujours défavorisés par le serveur de compilation.

Tableau 4.9: Compilation avec compilateur-serveur et interface texte

ensemble compilé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
columns	0.63	1.92	8.74	4.55	0.85	0.30	16.99
netobjd	0.55	0.27	1.50	0.88	0.67	0.25	4.12
webscape	2.45	1.32	4.26	1.74	1.94	0.62	12.33
m3browser	0.57	8.17	36.59	11.81	1.70	0.65	59.49
webvbt	3.16	11.16	39.29	16.88	1.42	2.23	74.14
m3tohtml	0.50	3.55	16.16	6.69	0.85	0.16	27.91
m3front	1.59	70.32	228.64	111.53	23.60	18.85	454.53
postcard	1.95	21.40	57.36	17.20	6.59	0.72	105.22
ps2html	2.17	24.16	86.16	27.43	8.79	1.34	150.05

Plusieurs points doivent être considérés ici.

D'abord, le compilateur standard SRC-Modula-3 est déjà doté d'une cache des arbres syntaxiques abstraits. La cache se construit au fur et à mesure de la compilation d'un ensemble. Une interface est donc compilée au plus une fois même avec ce compilateur. Avec les systèmes de persistance décrits dans la littérature, on parvient à conserver certains résultats de compilation en compilation évidemment mais on évite aussi de recompiler plus d'une fois un fichier d'en-tête au cours de la compilation d'un ensemble. Cette dernière caractéristique était donc déjà présente avec le compilateur standard. Le fait de maintenir certains résultats de compilation en compilation, la contribution de cette étude, sera donc davantage marquée dans des situations où le temps de compilation des interfaces importées a une importance par rapport au temps de compilation des fichiers du code source. Ainsi, pour un ensemble comme m3front, le temps de compilation des 38 interfaces qu'il importe est peu important par rapport

Tableau 4.10: Compilation avec générateur de code m3cgc1 et 64Mo de mémoire vive

ensemble compilé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
m3front	0.63	22.49	96.58	75.49	1.02	2.61	198.82
postcard	0.73	4.60	16.38	6.93	0.42	0.24	29.30
ps2html	0.88	5.59	29.37	14.72	0.63	0.30	51.49

à celui des 175 fichiers du code source. Par contre, les arbres syntaxiques abstraits des 30 interfaces du petit programme *columns* ont avantage à être conservés en mémoire de compilation en compilation car on peut épargner 1.75 secondes sur un total de 20.

Un autre point important concerne une fois de plus la mémoire. La recompilation des gros ensembles implique beaucoup d'entrées, sorties sur de nombreux fichiers. Le fait de maintenir une cache de quelques Mega-octets continuellement en mémoire a des conséquences puisque certains gros ensembles prennent plus de temps à compiler avec le serveur. Cette observation n'est pas celle de Onodera [41] qui mentionne pour le serveur en langage C orienté objet COB:

> "One might think that retaining all the data structures built would lead to a slower performance in terms of the working set theory, and that it would be preferable to keep a minimal set of data structures by using a separate compiler. However, as we have seen, given the size of real memory in recent workstation, the compilation server outperforms the seperate compiler even for a medium-sized program containing tens of thousands of lines."

Selon les observations de la présente étude, il est raisonnable de croire que cet effet bénéfique est surtout dû au fait qu'avec le serveur COB, on évite de recompiler plus

Tableau 4.11: Compilation avec compilateur-serveur et 64Mo de mémoire vive

ensemble compilé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
m3front	0.56	21.13	98.38	75.46	1.45	2.53	199.51
postcard	0.48	5.89	17.17	7.37	0.42	0.27	31.60
ps2html	0.61	5.32	29.81	14.93	0.64	0.28	51.59

d'une fois des interfaces et non au fait de maintenir des informations précompilées de compilation en compilation. une distinction qui n'est pas établie dans la littérature. D'ailleurs. on remarque en comparant les tableaux 4.2 et 4.9 que le serveur tend à faire augmenter le temps d'édition de liens. et même celui de m3cgc1 et de l'assembleur. pour les gros ensembles. Les tableaux 4.10 et 4.11 présentent respectivement les résultats pour la même expérience mais avec l'utilisation d'un ordinateur Pentiun-Pro® avec 64 Mega-octets de mémoire vive. On peut remarquer que cet effet est alors moins perceptible. Cependant. le serveur n'offre toujours pas un avantage pour la compilation de ces gros ensembles.

Les résultats avec une combinaison du serveur et du générateur de code intégré sont présentés au tableau 4.12. En les comparant à ceux du tableau 4.3. on remarque que. de manière générale. les effets bénéfiques du serveur et du générateur de code intégré s'additionnent pour le cas des ensembles de petite et moyenne taille. Par contre. encore ici. l'effet souhaité du serveur sur la compilation des gros ensembles est perdu par une augmentation du temps d'édition de liens dû à la mémoire restreinte.

Tableau 4.12: Compilation avec compilateur-serveur et générateur de code intégré

ensemble compilé	Temps (en seconces)				
	recomp. intel.	M3 ⇒ reloc.	Éd. de liens	autre	total
columns	0.64	3.39	0.85	0.69	5.57
netobjd	0.64	0.38	0.43	0.40	1.85
webscape	2.39	1.46	1.70	1.29	6.84
m3browser	0.50	12.42	1.07	0.32	14.31
webvbt	3.19	16.44	1.52	2.13	23.28
m3tohtml	0.52	4.78	1.06	0.19	6.55
m3front	1.56	92.65	19.73	17.33	131.27
postcard	1.99	27.89	6.90	0.98	37.76
ps2html	3.23	30.90	7.09	1.11	42.33

Selon les observations précédentes, le contexte de recompilation de seulement quelques unités d'un ensemble compilé devrait permettre d'obtenir de meilleurs résultats. En effet, les interfaces importées n'ont pas à être recompilées, ce qui peut représenter une part importante du temps total de compilation, car moins de fichiers source sont recompilés. De plus, il y a moins d'entrées/sorties et d'opérations sur disque dans cette situation.

Le tableau 4.13 montre les résultats obtenus dans un tel contexte. Les tests ont été effectués sur un ordinateur possédant 64 Mega-octets de mémoire vive pour éviter l'augmentation importante du temps d'édition de liens avec le serveur. L'expérience a consisté à recompiler 2 fichiers du code source (UnixMail.i3 et UnixMail.m3) qui font au total 2217 lignes. Les compilateurs utilisés sont indiqués. L'essai avec le compilateur-serveur suit celle effectuée avec celui standard. On peut également voir les résultats selon le type de générateur de code utilisé soit celui intégré ou m3cgc1.

Tableau 4.13: Recompilation de deux fichiers de l'ensemble postcard

compilateur utilisé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
standard avec m3cgc1	0.73	0.97	2.73	0.86	0.42	0.22	5.93
serveur avec m3cgc1	0.76	0.62	2.73	0.84	0.42	0.15	5.52
standard avec intégré	0.74	1.84			0.42	0.24	3.24
serveur avec intégré	0.64	1.36			0.42	0.20	2.62

Les chiffres montrent que le serveur a un effet intéressant sur la réduction du temps total de compilation. Le gain est plus faible dans le cas de l'utilisation du générateur de code m3cgc1 (6.7%), ce qui est tout à fait normal car le temps d'exécution de la partie frontale (0.97 seconde) prend seulement 1 5 du temps total de compilation (5.93 seconde). Avec le générateur de code intégré, cette proportion est plus importante donc le gain l'est également (19.1%).

Le tableau 4.14 donne les résultats pour une expérience similaire mais avec la recompilation de quatre fichiers (UnixMail.i3, UnixMail.m3, NI.i3 et NI.m3) qui font au total 3639 lignes. Les gains précédents sont maintenants de 11.4% et 21.3% respectivement. Cette augmentation vient du fait que la production des fichiers relocalisables est la seule étape du traitement qui nécessite plus de temps pour s'accomplir et qu'elle bénificie de l'effet du serveur de compilation.

Tableau 4.14: Recompilation de quatre fichiers de l'ensemble postcard

compilateur utilisé	Temps (en seconces)						
	recomp. intel.	M3 ⇒ L.I.	L.I. ⇒ ass.	ass. ⇒ reloc.	Éd. de liens	autre	total
standard avec m3cgc1	0.75	1.89	4.83	1.68	0.42	0.20	9.77
serveur avec m3cgc1	0.44	1.09	4.85	1.68	0.43	0.17	8.66
standard avec intégré	0.73	2.66			0.43	0.27	4.09
serveur avec intégré	0.51	2.05			0.42	0.24	3.22

En comparant les résultats obtenus à ceux trouvés dans la littérature, on remarque que ces derniers sont meilleurs. Il faut encore ici souligner que la contribution de la présente étude se situe au niveau de la persistance de la cache du serveur ainsi que de sa validation, le compilateur standard étant déjà doté d'un mécanisme de cache. Il devient donc difficile d'établir une base de comparaison de performance entre la présente réalisation et celles de la littérature. Reprogrammer une partie du compilateur standard SRC-Modula-3 pour lui retirer ses capacités à utiliser sa cache demanderait un effort important. Ce serait néanmoins le seul moyen d'obtenir un compilateur de référence pour juger de l'influence complète de la cache et de son maintien de compilation en compilation. Les comparaisons par rapport aux approches rencontrées dans la littérature deviendraient alors plus significatives. Les résultats présentés dans cette section montrent cependant que le système de persistance développé pour cette étude permet de réduire le temps de compilation de manière non négligeable.

Il est tout de même approprié de mentionner que Litman [36] obtient des améliorations du temps de compilation se situant entre 25% et 65% pour un système de fichiers d'en-tête précompilés pour le langage Objective-C. Encore ici, il est permis de croire qu'une bonne partie de ce pourcentage provient du fait que le nouveau système évite la recompilation d'une même portion de code plus qu'au fait d'en maintenir des résultats précompilés de compilation en compilation. Tout comme Litman, Horspool [24] a noté que le facteur de réduction du temps de compilation dépend considérablement de l'ensemble compilé évalué. Horspool parvient cependant, avec son serveur pour le langage C, à réduire le temps de compilation de près des deux tiers en moyenne dans un contexte de compilations répétitives.

Donc, en résumé, le serveur est peu utile dans un cadre de recompilation totale d'ensembles constitués de plusieurs fichiers. Son effet se fait plutôt sentir dans le contexte de la recompilation de quelques fichiers source. Cette situation est cependant plus courante et donc plus intéressante. On peut aussi remarquer que la quantité de mémoire vive influence le déroulement du processus de compilation, particulièrement avec le serveur dans le cas d'ensembles de taille importante.

Conclusion

L'objectif de cette étude était d'étudier, de mettre en oeuvre et de caractériser les techniques d'accélération de compilation par la génération rapide de code et par serveur persistant. Pour démontrer la pertinence de ces deux concepts, des améliorations ont été apportées au compilateur SRC-Modula-3. L'évaluation de la vitesse de ce nouveau système de compilation par rapport à l'original a montré que le temps de compilation peut être réduit sensiblement avec les techniques proposées. Le but ultime de réduire le cycle modification-compilation-mise-au-point se trouve donc aussi atteint, ce qui permettra aux programmeurs de recevoir plus rapidement les résultats d'une compilation.

En réduisant le nombre de passes, et en adoptant des méthodes simples de génération de code, le temps total de compilation peut ne plus être que de 30% à 50% de ce qu'il était. Cette intégration avec la partie frontale d'un générateur de code rapide est donc très importante. Cette remarque est d'autant plus justifiée que la qualité du code produit n'en est pas aussi affectée que l'on aurait pu s'y attendre.

Le système de persistance par serveur de compilation a surtout démontré son utilité dans le contexte de recompilation de quelques fichiers. L'étude laisse cependant entrevoir que le fait d'éviter de recompiler certaines parties d'un ensemble compilé en plaçant des informations précompilées dans une cache au cours d'une même compilation est plus important que de conserver ces résultats de compilation en compilation. Cette situation défavorise quelque peu le compilateur-serveur dans le contexte de recompilation totale d'un ensemble compilé de grande taille. Si seulement une partie de cet ensemble est recompilée, la situation change et le serveur permet de gagner de 10% à 20% en plus des gains obtenus avec le générateur de code intégré. L'utilisation du serveur est donc intéressante dans le cadre de la mise au point de logiciel.

Quelques points pourraient encore être améliorés dans le serveur pour réduire le temps de compilation. Actuellement, une petite interface simple permet de contrôler efficacement le serveur et de faire des requêtes de compilation. Les objets réseaux permettent cependant d'établir une véritable relation client-serveur. Le problème est qu'ils ajoutent une surcharge de traitement qui ralentit légèrement le temps de compilation. Une meilleure stratégie de communication avec le serveur serait souhaitable. Cette stratégie pourrait être établie dans le cadre du développement d'un environnement de compilation qui permettrait par exemple la communication du compilateur, d'un éditeur, d'un débogueur, etc...

Comme plusieurs autres serveurs, le compilateur aurait peut-être avantage à être utilisé dans un contexte multi-usagers. Cet aspect soulève évidemment des questions

de sécurité. De plus, la cache pourrait croître au-delà d'une limite raisonnable ce qui nécessiterait une politique de retrait des interfaces qui y sont présentes.

Dès le début de cette étude, il a été exclus d'avoir recours à du matériel dédié ou des techniques parallèles pour accélérer la compilation. Le but était de s'en tenir à un compilateur pouvant opérer sur un ordinateur peu dispendieux et disponible commercialement. De plus, le contexte de recompilations répétitives présentait un attrait car plus fréquent. Si on regarde ce qui peut être amélioré tout en restant dans ce cadre, quelques voies sont encore possibles.

Le chapitre précédent a démontré que le temps d'édition de liens peut devenir important pour la compilation de gros codes source. Linton et Quong [19] avaient vu qu'avec les processeurs rapides tels que ceux disponibles aujourd'hui, le temps d'édition de liens lui-même serait passablement réduit. Ils justifiaient cependant l'utilisation de leur éditeur de liens incrémental pour d'autres raisons que la vitesse de traitement. D'abord, l'effort fourni par l'éditeur de liens est toujours proportionnel à la taille du code ce qui représente toujours un problème pour les programmes de plus en plus gros. Plus important, ils soulignaient que le problème pourrait en devenir un d'entrées/sorties. C'est ce qui s'est produit à quelques reprises lors de l'évaluation de la performance. La présente recherche s'est attardée à accélérer la production des fichiers relocalisables à partir des fichiers source. Il faudrait cependant s'attendre à ce que le temps d'édition de liens domine éventuellement le temps total de compilation. Il serait intéressant de voir le comportement d'un éditeur de liens incrémental avec

l'utilisation d'un système de compilation tel que celui développé ici.

Évidemment, il existe des algorithmes d'analyses lexicale et syntaxique plus rapides que ceux employés par la partie frontale de Modula-3. En y regardant bien, on pourrait aussi trouver à améliorer certaines structures de données pour rendre la compilation plus rapide. Abman [1] dresse d'ailleurs une liste de détails susceptibles d'améliorer la vitesse, certains étant valables pour n'importe quelle application. Il est cependant évident que ces modifications ne pourront amener un gain aussi appréciable que celui obtenu par exemple par l'utilisation du générateur de code intégré. De plus, il est toujours tentant de sacrifier la vérification de certaines erreurs par le compilateur ou de développer un langage adapter à la compilation rapide pour obtenir une performance supplémentaire. Cependant, pour des langages, comme Modula-3, qui ont été développés pour la programmation de logiciels de taille importante et pour la programmation au niveau du système d'exploitation, ce compromis est peu intéressant et même dangereux. La vitesse de compilation est certes un point intéressant mais d'autres aspects du langage ou du compilateur doivent souvent prédominer pour former un ensemble efficace.

Présentement, à chaque compilation, les fichiers "m3x" nécessaires au système de recompilation intelligent doivent être relus. Comme ces fichiers sont rarement modifiés dans le cas des librairies importées, il serait possible de conserver certaines informations en mémoire de compilation en compilation un peu comme on l'a fait pour les interfaces avec le serveur. Cette stratégie serait rentable particulièrement dans les cas

où le temps pris par le système de recompilation intelligent est important. Cependant, les informations tirées des fichiers "m3x" sont liées à celles fournies par *Quake* pour établir des structures de données utiles au reste de la compilation. Il faudrait évaluer la façon de garder la cohérence des informations dans ces structures. Une telle approche pourrait aussi permettre de réduire les entrées/sorties. Cet aspect est non négligeable puisque les processeurs sont de plus en plus rapides. Les solutions permettant de réduire des accès au disque demeurent valables à plus long terme.

Bien sûr, l'accélération de la compilation n'est qu'une facette des langages et des compilateurs. Dès le début de l'élaboration du langage Modula-3, il avait été établi que l'ouvrage en serait un de consolidation. En d'autres mots, on chercherait à réunir les meilleurs caractéristiques de programmation dans un langage simple. En regardant le travail effectué sur le compilateur, on peut constater qu'il s'agit aussi d'un travail de consolidation. Le compilateur SRC-Modula-3 est porté sur la plupart des plates-formes et possède un système de recompilation intelligent. *Quake* est un aspect unique et efficace du système de compilation. Le travail effectué pour cette étude a permis d'ajouter des capacités de serveur et de génération rapide de code. Déjà, des travaux sont en cours pour permettre d'effectuer du chargement dynamique de modules. Le compilateur est donc en train de devenir la réunion de plusieurs techniques de compilation. D'ailleurs, la cohabitation de ces techniques est un aspect important. Par exemple, l'algorithme de validation de la cache du serveur a grandement bénéficié de certains aspects du système de recompilation intelligent. La contribution de cette

étude s'inscrit donc également dans une perspective plus globale qui est celle d'un compilateur réunissant des techniques de compilation efficaces.

Références

[1] WERNER ABMAN. A short review of high speed compilation. *Proceedings of the Second Compiler, Compilers and High Speed Compilations Workshop*, pages 1–10, (1988).

[2] YAOHAN CHU. Application-specific coprocessor computer architecture. *Proceedings of the International Conference on Application Specific Array Processors*, pages 653–664, (1990).

[3] INTEL CORPORATION. *Architecture and Programming Manual*, volume 3 of *Pentium User's Manual.* Intel Literature Sales, Mt. Prospect, IL, (1994).

[4] M. K. CROWE. Dynamic compilation in the Unix environment. *Software–Practice and Experience*, 17(7):pages 455–467, Juillet (1987).

[5] DEAN ELSNER, JAY FENLESON, and FRIENDS. The GNU Assembler. Free Software Foundation Inc., 59 Temple Place–Suite 330, Boston, MA, 01307,USA, Janvier (1994). http://www.ns.utk.edu/gnu/binutils/as_toc. html.

[6] SANJAY KHANNA, ARIF GHAFOOR et AMRIT GOEL. A parallel compilation technique based on grammar partitioning. *Proceedings of the ACM Annual Computer Science Conference*, pages 385–391, (1990).

[7] JOHN F. BEETEM et ANNE F. BEETEM. Incremental scanning and parsing with Galaxy. *IEEE Transactions on Software Engineering*, 17(7):pages 641–651, Juillet (1991).

[8] ROLF ADAMS, WALTER TICHY et ANNETTE WEINERT. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):pages 3–28, Janvier (1994).

[9] ANDREW S. TANENBAUM, M. FRANS KAASHOEK, KOEN G. LANGENDOEN et CERIEL J.H. JACOBS. The design of very fast portable compilers. *SIGPLAN Notices*, 24(11):pages 125–131, Novembre (1989).

[10] RICHARD M. STALLMAN et CYGNUS SUPPORT. Debugging with GDB–The GNU Source-Level Debugger. Free Software Foundation Inc., 59 Temple Place–Suite 330, Boston, MA, 01307,USA, Janvier (1994). http://www.cygnus.com/library/gdb/gdb_toc.html.

[11] JACK W. DAVIDSON et DAVID B. WHALLEY. Quick compilers using peephole optimisation. *Software–Practice and Experience*, 19(1):pages 1151–1162, Janvier (1989).

[12] JULIA MENAPACE, JIM KINGDON et DAVID MACKENZIE. The "Stabs" debug format. Revision 2.124, Free Software Foundation Inc., 59 Temple Place–Suite 330, Boston, MA, 01307,USA, (1993). http://www.ns.-utk.edu/gnu/gdb/stabs_toc. html.

[13] URS HÖZLE et DAVID UNGAR. A third-generation SELF implementation: Reconciling responsiveness with performance. *Ninth Annual Conference on Object-*

Oriented Programming Systems, Languages and Applications, pages 229–241, (1994).

[14] ANDREW BIRRELL, GREG NELSON, SUSAN OWICKI et EDWARD WOBBER. Network Objects–Report 115. Systems Research Center Digital Equipment Corporation 130 Lytton Avenue Palo Alto, Californie, U.S.A., 94301, Février (1994). ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-115.ps.Z.

[15] BILL KALSOW et ERIC MULLER. SRC Modula-3 Version 3.3. Systems Research Center Digital Equipment Corporation 130 Lytton Avenue Palo Alto, Californie, U.S.A., 94301, Janvier (1995). ftp://www.vlsi.polymtl.ca/m3/document/src-m3-doc/SRCm3-3.3.ps.gz .

[16] PAOLA INVERARDI et FRANCO MAZZANTI. Experimenting with dynamic linking with Ada. *Software–Practice and Experience*, 23(1):pages 1–14, Janvier (1993).

[17] LUCA CARDELLI, JAMES DONAHUE, LUCILLE GLASSMAN, MICK JORDAN, BILL KALSOW et GREG NELSON. Modula-3 Report (revised). Systems Research Center Digital Equipment Corporation 130 Lytton Avenue Palo Alto, Californie, U.S.A., 94301, Novembre (1989). ftp://www.vlsi.polymtl.ca/m3/document/src-m3-doc/Modula3.ps.gz.

[18] AFRED AHO, RAVI SETHI et JEFFREY ULLMAN. *Compilers: Principales, Technics and Tools.* Addison-Wesley Publishing Company, Reading, MA, (1986).

[19] RUSSELL W. QUONG et MARK A. LINTON. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems*, 13(1):pages 1–20, (1991).

[20] RUSS ATKINSON, ALAN DEMERS, CARL HAUSER, CHRISTIAN JACOBI, PETER KESSLER et MARK WEISER. Experiences creating a portable Cedar. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 322–329, (1989).

[21] MARYS P. BIVENS et MARY LOU SOFFA. Reuse of compiler analysis in a programming environment. *Proceedings of the Seventeenth Annual ACM Computer Science Conference*, pages 368–373, (1989).

[22] DAVID B. WORTHMAN et MICHAEL D. JUNKIN. A concurrent compiler for Modula-2+. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, (1992).

[23] BENOIT POIRIER et MICHEL DAGENAIS. Outil d'extraction et de reconnaissance de la structure de documents. *Quatrième colloque national sur l'écrit et le document*, pages pages 179–184, (1996).

[24] BRAIN KOEHLER et NIGEL HORSPOOL. CCC: A Caching Compiler for C. *À paraître dans: Software–Practice and Experience*, (1997).

[25] DAVID TARDITI et PETER LEE. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):pages 161–177, Juin (1992).

[26] DANIEL M. YELLIN et ROBERT E. STROM. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):122–131, Avril (1991).

[27] WADE H. SHAW, Jr., JAMES W. HOWATT et ROBERT S. MANESS. A software science model of compile time. *IEEE Transactions on Software Engineering*, 15(5):pages 543–549, Mai (1989).

[28] MARK A. LINTON et RUSSEL W. QUONG. A macroscopic profile of program compilation and linking. *IEEE Transactions on Software Engineering*, 15(4):pages 427–436, Avril (1989).

[29] S. I. FELDMAN. Make–a program for maintaining computer programs. *Software–Practice and Experience*, 9(3):pages 255–265, Mars (1979).

[30] GINTARA R. GIRCYS. *Understanding and Using COFF.* Nutshell Series. O'Reilly & Associates, Sebastopol, CA, (1988).

[31] JURG GUTKNECHT. Separate compilation in Modula-2: An approach to efficient symbol files. *IEEE Software*, 3(11):pages 29–38, Novembre (1986).

[32] SAMUEL P. HARBISON. *Modula-3.* Prentice Hall, Englewood Cliffs, NJ, (1992).

[33] MICK JORDAN. An extensible programming environment for Modula-3. *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 66–76, (1990).

[34] UNIX SYSTEM LABORATORIES. Executable and Linkable Format (ELF). ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz.

[35] UNIX SYSTEM LABORATORIES. *System V Application Binary Interface–Intel386 Architecture Processor Supplement.* Unix Press, (1993).

[36] ANDY LITMAN. An implementation of precompiled headers. *Software–Practice and Experience*, 23(3):pages 341–350, Mars (1993).

[37] HONGJIU LU. ELF: From the Programmer's Perspective. NYNEX Science and Technology Inc., 500 Westchester Avenue, White Plains, NY 10604, USA, Mai (1995). ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz.

[38] B.J. MCKENZIE. Fast peephole optimisation techniques. *Software–Practice and Experience*, 19(12):pages 1151–1162, Décembre (1989).

[39] MARY LOU NOHR. *Understanding ELF Object Files and Debugging Tools.* Programmer Collection. Prentice Hall, Englewood Cliffs, NJ, (1994).

[40] JACQUES NOYÉ. The KCM system: Speeding-up logic programming through hardware support. *Proceedings of the Logic Programming and Automated Reasoning International Conference*, pages 496–498, (1992).

[41] TAMIYA ONODERA. Reducing compilation time by a compilation server. *Software–Practice and Experience*, 23(5):pages 477–485, Mai (1993).

[42] MICHAEL P. PLEZBERT. Continuous compilation for software development and mobile computing. Master's thesis, Washington University, Mai (1996).

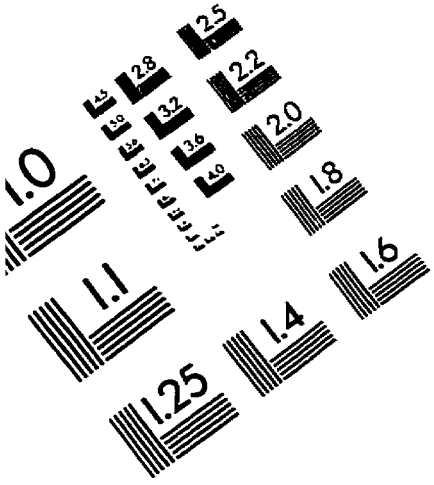
[43] MAYER D. SCHWARTZ. Incremental compilation in Magpie. *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 122–131, (1984).

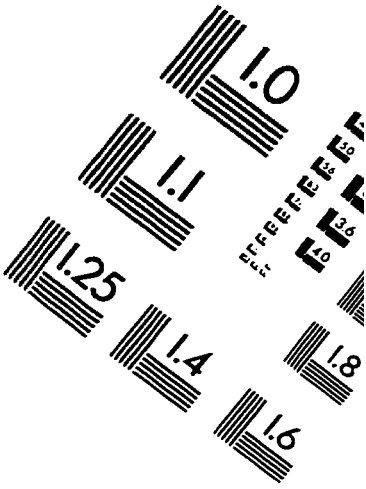
[44] RICHARD M. STALLMAN. GCC–the GNU C and C++ compiler. Free Software Foundation Inc., 59 Temple Place–Suite 330, Boston, MA, 01307,USA, Novembre (1995). http://www.cygnus.com/library/gcc/usegcc_toc.html.

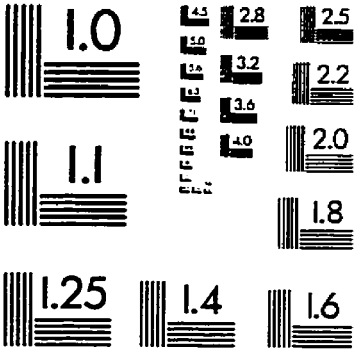
[45] BJARNE STROUSTRUP. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, (1992).

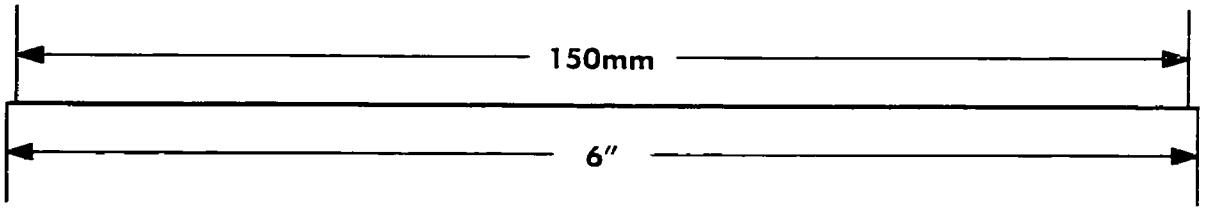
[46] WALTER F. TICHY. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):pages 273–291, Juillet (1986).

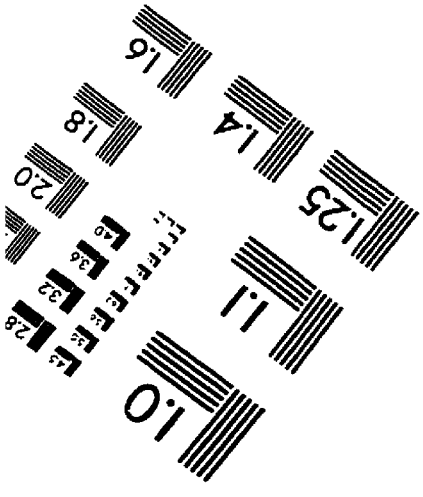
IMAGE EVALUATION TEST TARGET (QA-3)

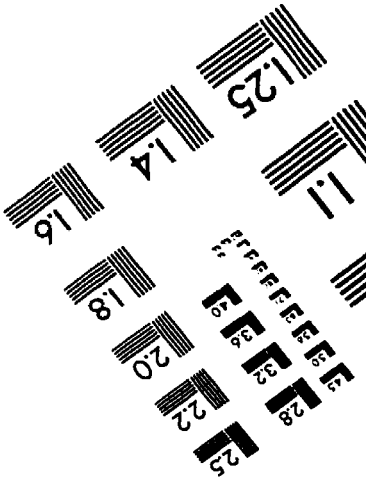












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved