UNIVERSITÉ LOUIS PASTEUR STRASBOURG

Thèse présentée pour obtenir le grade de
Docteur de l'Université Louis Pasteur
Strasbourg I

Discipline : Informatique

par Bénédicte Ramelie KENMEI YOUTA

Génération de programmes modèles pour la représentation et l'analyse de profils d'exécution : le modèle périodique-linéaire

soutenance publique le : 27/06/2006

membres du jury :

Directeur de thèse :
Philippe Clauss, Professeur à l'Université Louis Pasteur, Strasbourg
Président et rapporteur interne :
Jerzy Korczak, Professeur à l'Université Louis Pasteur, Strasbourg
Rapporteurs externes :
Christine Eisenbeis, Directeur de recherche INRIA Futurs, Orsay
Patrice Quinton, Professeur à l'Université de Rennes I
Examinateur :
Maurice Tchuente, Professeur à l'Université de Yaoundé I

Apprendre à réfléchir, c'est agir pour son propre bien ;
s'appliquer à comprendre mène au bonheur.
Salomon

Soli Deo Gloria

Reconnaissance

Je voudrais exprimer toute ma gratitude aux personnes et institution sans lesquelles ce travail n'aurait pu voir le jour.

Mon directeur Philippe Clauss a initialement suscité en moi un vif intérêt pour le présent domaine de recherche, et a été ensuite à l'origine de mon premier séjour au sein de son équipe à Strasbourg ; c'est là que j'ai effectué mon stage de DEA, en prélude à ce travail de thèse. Durant ces années, j'ai pu apprécier sa merveilleuse disponibilité, son investissement et tous ses conseils éclairés pour mon apprentissage. Pour tout cela je tiens à lui exprimer mes profonds remerciements.

Je remercie également le Centre National de la Recherche Scientifique qui a fourni tout le soutien financier nécessaire au bon déroulement de la thèse.

Je remercie les membres de l'équipe ICPS au sein de laquelle je travaille, pour l'ambiance conviviale qui est entretenue par les uns et les autres.

Je remercie le Dr Emmanuel Kamgnia de l'université de Yaoundé I, non seulement pour la formation reçue de lui, mais aussi pour l'intérêt qu'il a toujours manifesté à mon évolution scientifique.

Je ne saurai omettre Claude Tadonki qui m'a encouragée depuis ma première année d'université, et qui répond toujours présent lorsque je sollicite son aide.

Je remercie ma famille pour le soutien indéfectible dont je bénéficie de leur part, mon frère Emmanuel Youta dont les questions régulières sur l'état d'avancement de mon travail m'ont maintenue en alerte, mon époux Yves qui m'a réconfortée durant les temps difficiles de la thèse, Lisette, Patricia, Rabelais, Carine, Hilaire, Valéry, ainsi que tous les autres.

Enfin je remercie mes amis dont la présence ou les courriers électroniques m'ont accompagnée ces années durant, il s'agit de Henri, Calvin, Yolande, Eric, Rodrigue, Madeleine, Edwige, Etienne, la famille Zouzou, Adelphine, Léonard, Charles, et tous ceux qui se reconnaîtront d'après cette liste.

Dédicaces

A Bernard et Marthe YOUTA
A Yves Saturnin Djiki
A tous mes frères et soeurs,
ainsi que mes petits neuveux et nièces

Table des matières

Chapitre 1

Introduction

Les nouvelles architectures de processeurs ont bénéficié d'un ensemble de mécanismes novateurs ayant eu pour effet un formidable accroissement de leurs performances. Ces mécanismes concernent entre autres l'introduction du pipeline avec des étages de plus en plus longs, le parallélisme d'instructions, l'augmentation de la fréquence des horloges, etc. D'un autre point de vue, il s'agit également de procédés assez complexes qui rendent un peu plus ardues les interactions entre les composantes de la machine. De plus, dans la pratique, il reste avéré que les performances effectives lors de l'exécution des programmes restent encore assez éloignées des performances crêtes (fournies par les constructeurs).

L'essor des systèmes embarqués ou enfouis a introduit de nouveaux objectifs de performances, et ce particulièrement en terme de minimisation de la consommation électrique, minimisation de la consommation mémoire, maximisation de la sureté de fonctions, et respect des contraintes temps-réel. Il ne s'agit pas uniquement ici de minimiser les temps d'exécution, mais plutôt de respecter un contrat plus ou moins strict, fixant des limites de coût et de fiabilité concernant aussi bien le matériel que le logiciel. La maîtrise du comportement du système est donc essentielle dans ce cadre. Elle passe par la compréhension des distorsions entre un comportement attendu et le comportement effectif.

Pour bien comprendre ces différents phénomènes et réduire les écarts de performance résultants et qui peuvent se révéler préjudiciables à plus d'un titre, notamment sur un plan financier, il faut s'intéresser à leurs causes. En général, celles-ci se situent à quatre niveaux, à savoir le niveau de l'application, du compilateur, du système d'exploitation et de l'architecture.

Au niveau de l'application, il arrive souvent que les codes ne soient pas assez bien structurés pour permettre au compilateur d'exploiter les ressources du processeur (parallélisme d'instructions, etc), et le compilateur de son côté peut générer des codes exécutables qui exploitent mal les ressources physiques de la machine cible, ou qui ne tiennent pas compte des contraintes propres au cadre d'utilisation (taille de code, temps d'exécution constant, ...).

Au niveau du système d'exploitation, une stratégie inadéquate de gestion des ressources (gestion des pages mémoire, fréquence de permutation entre processus, ...) peut amoindrir significativement les performances et provoquer de nombreux phénomènes inattendus.

Enfin pour ce qui est de l'architecture, les ressources physiques du processeur sont quelquefois inadaptées aux besoins des programmes ; en outre, la différence de cadence constatée entre le processeur et les autres composantes (mémoire, bus) entraîne régulièrement de nombreux goulots d'étranglement.

1.1 Quelques approches de solution

En relation à ces différents niveaux mentionnés, plusieurs travaux ont été menés jusqu'ici avec des objectifs bien précis à chaque fois. Par exemple, pour l'identification des goulots d'étranglement ou encore des chemins critiques (*hot paths*), on développe des outils logiciels et matériels de mesure permettant de déduire les valeurs d'un ou plusieurs facteurs influant sur les performances des programmes. Parmi ces outils on peut citer gprof [40], Vtune [1], Perfmon [2], et bien d'autres (PCL [14], SimpleScalar [18], RSIM [67], TRIMARAN [44], CVT [81]).

Une autre piste consiste à effectuer une évaluation de l'architecture et des compilateurs en construisant des modèles (mathématiques, statistiques, analytiques, etc) qui permettent de faire de la prédiction ([21, 47, 66, 76]) : prédiction du temps d'exécution, prédiction de branchements, prédiction d'accès mémoire, etc. Grâce à de tels modèles on est capable de choisir statiquement, le cas échéant, une meilleure version d'un code parmi plusieurs ou une configuration particulière de l'architecture d'un processeur.

On trouve encore une autre alternative dans l'analyse des applications, dont le but est d'en comprendre le comportement et partant de là les dégradations éventuelles des performances, et déduire au besoin des optimisations qui seront effectuées en général non pas sur la totalité de l'application, mais sur des endroits critiques à déterminer judicieusement. Ces analyses sont généralement menées sur des traces de programmes [13, 27, 56, 57] ou profils d'exécution, qui sont des élé-

ments de base considérés comme représentants des applications à étudier. C'est dans ce contexte en particulier que se situe notre travail.

Le critère que l'on contrôle le plus souvent lors de l'analyse est le temps d'exécution des programmes, cependant il est intéressant de noter que ce temps peut revêtir une signification différente suivant le niveau où on se trouve : application, système d'exploitation ou processeur.

En effet, le programmeur verra le temps d'exécution comme celui qui sépare le lancement de son programme de la réception des résultats, l'administrateur du système d'exploitation quant à lui s'intéressera surtout au débit des tâches exécutées, tandis que le concepteur du processeur considèrera principalement le flux des instructions de bas niveau.

C'est la raison pour laquelle mis à part ce temps d'exécution (variable), il paraît fort utile de considérer également d'autres métriques, nombreuses par ailleurs, en fonction desquelles on peut conduire des analyses ; c'est ainsi qu'on peut s'intéresser aux éléments suivants :

- le nombre moyen de défauts de cache, pour les programmes dont le comportement est dominé par leurs accès mémoires, et plus largement encore le comportement en mémoire des programmes ;
- la consommation d'énergie, pour le cas des programmes embarqués où le système spécialisé est alimenté par une source d'énergie extrêmement rationnée ;
- le taux d'occupation de la mémoire embarquée ;
- le pourcentage de branchements correctement prédits, et bien d'autres encore.

1.2 L'analyse de programmes

Définition 1.2.1 *L'analyse de programmes désigne l'ensemble des techniques qui permettent de déduire mécaniquement des propriétés des programmes.*

C'est un ensemble de techniques aux applications variées et ayant déjà donné lieu à de nombreuses optimisations de programmes. Par ailleurs, elle permet des avancées significatives dans la compréhension du comportement des programmes, et cette mise en évidence du comportement des programmes est de plus en plus reconnue comme une clef principale pour de nombreux développements : reconfiguration dynamique de cache pour économiser de l'énergie, simulations matérielles au niveau de l'architecture, optimisations du compilateur, *remote profiling*, choix du coeur sur lequel faire tourner un processus dans une architecture multicores, etc.

Dans le cadre des systèmes embarqués notamment, qui sont des systèmes à base de logiciels ayant de très fortes interactions avec l'environnement qu'ils doivent contrôler, l'analyse du comportement est une part essentielle tant dans la maîtrise des accès aux caches, assez difficiles à prédire, que dans celle de la consommation électrique.

S'il s'agit en plus de systèmes temps réel, devant donc satisfaire des contraintes de ponctualité et de qualité de service, il est nécessaire de fournir une estimation précise et fiable des temps d'exécution. Et pour cela, on effectue une analyse du comportement temporel du matériel, car c'est elle qui permet de déterminer les temps d'exécution des blocs de base des programmes.

Ainsi, le besoin de maîtriser le comportement des programmes est un besoin crucial, et ce besoin peut être satisfait à travers une analyse de programmes. Pour cela, il existe deux démarches principales généralement utilisées, à savoir la démarche statique et la démarche dynamique.

1.2.1 Analyse statique

Définition 1.2.2 *L'analyse statique de programmes désigne l'ensemble de techniques permettant d'obtenir de l'information sur le comportement d'un programme uniquement à l'examen de son code source.*

Elle consiste par exemple en une recherche de dépendances entre les données, une analyse de flots (données ou contôle), un calcul de durée de vie des variables, etc, et s'effectue en général à la compilation.

Ce type d'analyse a été très largement développé avec succès et possède de nombreuses applications comme par exemple la vérification et mise au point de programmes, en plus de l'optimisation. Cependant, les résultats obtenus ne sont que des approximations dans certains cas (cas de variables dont la valeur ne peut être connue à la compilation). Par ailleurs, lorsqu'on est en présence d'applications faisant grand usage de structures de données complexes (pointeurs, etc), l'analyse statique se trouve fort limitée. Or de telles applications sont de plus en plus nombreuses.

De plus, une analyse précise du comportement global d'un système doit également tenir compte de la machine d'exécution. La complexité croissante du logiciel et du matériel, mais surtout leurs interactions, rendent quasiment impossible l'analyse statique à partir de données telles que le code source et certains paramètres d'architectures. De nombreux travaux utilisent des modèles généraux de machines telles que PRAM [39], LogP [29], BSP [80] ou QSM [38] en parallélisme. Ces modèles peuvent offrir des estimations suffisamment précises dans certains contextes à faibles contraintes d'exécution, mais s'avèrent bien souvent trop généraux lorsque ces contraintes obligent à maîtriser jusqu'à des mécanismes élémentaires d'un système.

1.2.2 Analyse dynamique

Définition 1.2.3 *L'analyse dynamique de programmes désigne l'ensemble des techniques permettant d'observer le comportement à l'exécution des programmes.*

Ce faisant, elle donne des informations bien plus précises et réalistes que l'analyse statique (par exemple quels sont les lieux où le programme passe le plus de temps).

Ce type d'analyse peut s'effectuer aussi bien au niveau matériel (par des appareils greffés aux processeur) qu'au niveau logiciel. Au niveau logiciel, qui est le plus largement répandu, on introduit des points d'observations ou *sondes* [1] dans le programme pour capturer des événements intéressants : c'est l'instrumentation [63]. Ensuite le programme sondé est exécuté pour produire soit une trace à analyser (i.e. un fichier dans lequel sont stockés les résultats des observations), soit des résultats finaux de mesure. La figure 1.2.2 résume les différentes étapes de l'analyse dynamique.

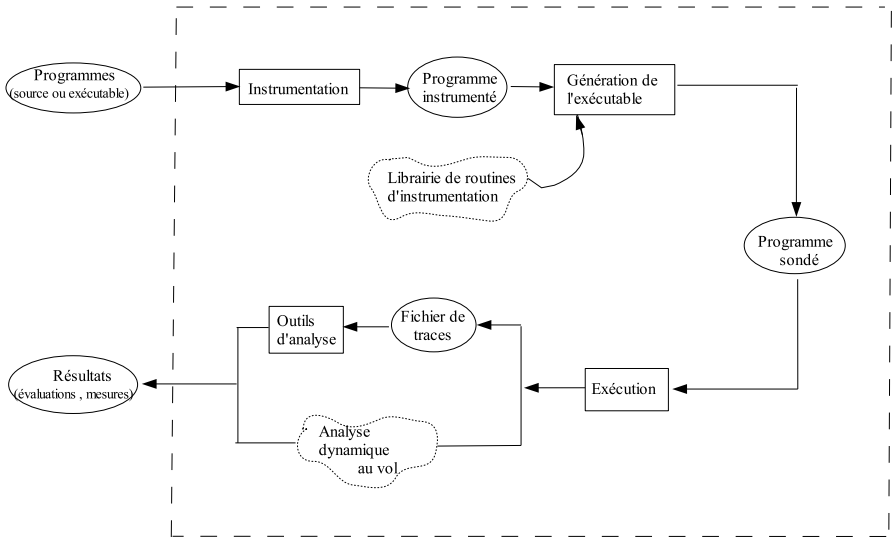


FIG. 1.1 – Etapes de l'analyse dynamique

[1] Ces sondes sont des appels à des routines spéciales d'instrumentation et servent à comptabiliser, sauvegarder, mesurer ou analyser certains événements tout au long de l'exécution du programme.

Selon les résultats en sortie lors d'une analyse dynamique de programmes, on parle de *trace driven analysis* ou de *execution driven analysis*.

Dans le premier cas, il n'y a pas de surcoût de traitement, pas d'interférence considérable avec le programme. Par contre la taille des traces peut être excessivement grande.

Dans le second cas, l'analyse est effectuée au vol (*on-the-fly*) et les informations dynamiques capturées par les routines d'instrumentation sont analysées au fur et à mesure que le programme s'exécute. Il y a donc un surcoût considérable de traitement d'informations, ce qui peut se répercuter sur les performances globales du programme. C'est pourquoi nous avons choisi de produire les différentes traces à analyser seulement en fin d'exécution des programmes.

1.3 Techniques générales de collecte et d'analyse de traces

Les traces sont partout considérées comme éléments essentiels dans l'étude du comportement des programmes, et peuvent être recueillies à 8 niveaux différents d'après Uhlig et al. [79], regroupés en deux catégories : le domaine du matériel et le domaine logiciel. Dans la production de traces assistée par le matériel, on utilise l'architecture de la machine, à l'instar des compteurs avec lesquels on évalue le nombre de défauts de cache. Au niveau logiciel par contre il s'agit surtout d'utiliser des instrumentations comme vu précédemment ou encore des simulations.

Ball et Larus proposent des techniques efficaces de collecte de traces [13, 56], et constatent qu'il n'est pas utile d'insérer des codes d'instrumentation dans tous les blocs de base. Par une méthode de profilage, ils comptent la fréquence d'exécution de tous les blocs et recueillent ensuite la trace des blocs effectivements exécutés au cours du programme. Ils présentent également une discussion sur la question du choix du lieu de placement du code d'instrumentation.

A ces techniques, Larus ajoute une exécution abstraite [55], qui est une technique semblable aux techniques de compilation, pour réduire la taille des fichiers de trace : on identifie un sous-ensemble de la trace qui est suffisant pour la générer en totalité, et lors de l'éxécution du programme on n'enregistre que les événements de ce sous-ensemble. Les principales tâches du programme devant être ciblées sont : les références mémoire et le contrôle de flots.

Dans ce dernier cas, il est fréquent que l'analyse de la trace se fasse à travers une représentation sous forme de graphe. Nous en avons un exemple dans [57], où les traces analysées comportent les séquences de blocs de base successivement exécutés au cours du programme, et qui sont encore appelés chemins.

Au cours de l'analyse, on cherche à identifier les régularités présentes dans la trace et pour cela, on effectue une compression basée sur l'algorithme de compression hiérarchique SEQUITUR de Nevill-Manning et Witten [64, 65] ; par la suite,

le résultat de cette compression est traduit par un graphe acyclique orienté, qui représente entièrement le flux de contrôle du programme étudié.

L'ensemble de ces travaux produit donc des traces au niveau logiciel, et les résultats d'analyse obtenus permettent effectivement de mieux saisir le comportement des programmes et de dériver des optimisations significatives. De part la méthode de collecte de traces nous nous rapprochons de ces travaux, étant donné que nous procédons également de manière logicielle. Toutefois ce qui fait nous distingue réside au niveau de la nature et du format des traces, puisque nous nous intéressons non pas au contrôle de flots, mais plutôt aux références effectuées par les programmes.

Dans notre travail, les traces étudiées concernent des références mémoires, obtenues à la suite d'instrumentations de codes. Après avoir identifié au préalable les fonctions les plus coûteuses à l'aide d'outils comme *gprof*, nous déterminons les instructions à tracer et le résultat de leur instrumentation est enregistré dans un fichier texte, correspondant au fichier de trace.

A la figure 1.2 nous présentons une trace obtenue après enregistrement des valeurs entières de trois pointeurs (`parray, parray2, parray3`), relativement à une adresse de base ; la fonction instrumentée provient du programme `fir2dim` tiré des bancs d'essai DSPstone [85].

0	1	2	4	5	6	8	9	10	1	2	3	5	6	7	9	10	11	4
5	6	8	9	10	12	13	14	5	6	7	9	10	11	13	14	15		

FIG. 1.2 – Exemple : une trace d'accès pointeurs

Selon les objectifs de l'analyse, les éléments et le format des traces peuvent varier. Si l'on s'intéresse à la prédiction de branchements par exemple, on peut constituer une trace de valeurs binaires, la valeur 1 étant enregistrée à chaque fois qu'un branchement est pris et la valeur 0 indiquant les cas où le branchement n'est pas effectué.

Dans le cas des systèmes embarqués, si l'on désire optimiser la consommation d'énergie on peut analyser par exemple la variation de températures du processeur ; dans ce cas la trace est constituée de différentes valeurs de températures, prélevées à intervalles de temps fixe à l'aide d'un matériel adéquat.

Bien d'autres caractéristiques encore peuvent être *tracées* et analysées, comme nous le verrons par la suite.

1.4 Objectifs de la thèse et développement d'un modèle d'analyse

Dans cette thèse, l'on s'intéresse à l'analyse et la modélisation des traces de programmes issues d'une instrumentation de codes. Les traces par lesquelles nous sommes concernés sont constituées en particulier de références mémoires et non pas d'informations sur le contrôle de flots, comme c'est le cas ailleurs. Nous nous attacherons donc à concevoir et développer un modèle de représentation de tels profils d'exécution, ayant essentiellement les deux caractéristiques suivantes :

- permettre d'améliorer la compréhension et la maîtrise du comportement du programme source ;
- se prêter ensuite aisément à différentes sortes d'analyses ayant pour but de déduire des transformations ou des optimisations de codes.

Pour réaliser pleinement ces objectifs, ce modèle de représentation des traces doit être sous une forme proche des fonctions génératrices, et peut être par exemple sous forme de fonctions statistiques ou autres fonctions mathématiques. Comme les traces que nous étudions sont constituées de valeurs numériques entières et s'assimilent par conséquent assez bien à des séries temporelles, le modèle recherché peut profiter des méthodes utilisées dans le domaine de l'économétrie.

D'une manière plus générale, ces traces peuvent aussi être considérées simplement comme des informations ou des données, c'est pourquoi les méthodes de fouilles de données (*ang. Data Mining*) présentent également un certain intérêt.

Dans une première approche d'analyse, nous investissons les deux communautés que sont l'économétrie des séries temporelles et la fouille de données, à la recherche d'un modèle de représentation et d'analyse. Il apparaît que ces méthodes générales d'analyse de données fournissent plusieurs informations intéressantes, mais une étude approfondie montre que ces informations sont justement bien trop générales pour améliorer la compréhension du comportement des programmes analysés, ou encore pour permettre des optimisations dans notre contexte. Or, cette compréhension du comportement des programmes est un objectif essentiel de nos analyses, pour les différentes raisons déjà mentionnées précédemment.

Nous effectuons une seconde approche d'analyse et proposons alors une solution de modélisation qui consiste à exprimer le comportement à l'exécution d'un programme par un autre programme : il s'agit donc d'un programme *modèle*. Ce dernier programme n'exprime pas la fonctionnalité, "le pourquoi", mais la manière, "le comment", et cela est rendu possible par le fait que les langages informatiques offrent un pouvoir d'expressivité où la fonctionnalité peut être transposée au comportement.

La génération du programme modèle qui exprime la manière, le comportement du programme initial, se fait sous forme d'une séquence de nids de boucles dans lesquelles :

- les instructions des boucles les plus internes sont des fonctions polynomiales,
- les bornes des boucles sont soit des constantes, soit des fonctions affines des indices de boucles.

Les fonctions de niveau les plus internes expriment les valeurs de la trace d'entrée à partir des indices de boucles. On obtient ainsi et de manière automatique une représentation du profil d'exécution du programme de départ.

Le modèle proposé est basé sur une méthode d'interpolation périodique et linéaire, d'où l'appelation de *modèle périodique-linéaire*. L'approche de modélisation que nous développons représente donc une trace d'entrée provenant d'une analyse dynamique par une séquence de nids de boucles générée automatiquement. Chaque séquence de nids de boucles correspond à une définition particulière d'une *phase* de programme, une phase étant un ensemble d'intervalles dépendants les uns des autres, identifiés à travers une interpolation périodique-linéaire.

Le programme modèle obtenu peut être par la suite analysé de manière à mettre en exergue des caractéristiques intéressantes du comportement du programme étudié, et le cas échéant de déduire des transformations optimisantes. En nous aidant d'outils tels que le modèle polyédrique qui se prête de manière adéquate à un grand nombre d'analyses, nous effectuons une étape d'analyse "statique" sur le programme modèle ; ce faisant, nous mettons en œuvre une certaine collaboration entre les deux principaux types d'analyse que sont l'analyse statique et l'analyse dynamique.

Partant de la prémisse selon laquelle notre représentation reflète davantage le comportement du programme analysé, nous voulons établir et vérifier que les informations obtenues à l'issue de notre modélisation sont effectivement plus précises, mieux adaptées, et plus exploitables qu'avec des méthodes générales d'analyse de données, dans l'objectif majeur de maîtrise du comportement des programmes.

La suite de ce document est organisée comme suit.

Le chapitre 2 présente quelques techniques d'analyse rencontrées dans le domaine de la fouille de données et des séries temporelles, de même que les limites de l'usage que nous avons pu en faire. Il est également question de quelques méthodes d'analyse de traces plus spécifiques au domaine de l'analyse de performances, notamment de l'idenfication des phases de programmes. Nous terminons ce chapitre par une motivation de la recherche de méthodes plus spécifiques et mieux adaptées aux traces que nous analysons.

Dans le chapitre 3 nous introduisons notre modèle périodique-linéaire (*Periodic Linear Model*) pour l'analyse de traces, modèle basé sur une interpolation périodique-linéaire. Le concept de *phases* ou de phases d'intervalles tel que nous

l'exploitons est plus longuement précisé à ce niveau, de même que les différentes configurations du modèle auxquelles l'on peut aboutir. Ces configurations sont directement liées au nombre de comportements décelables dans la trace, à leur entrelacement et à leur rapport au temps. De nombreux exemples viennent illustrer les étapes de construction du modèle.

Le chapitre 4 met en exergue le lien étroit qui existe entre le modèle périodique-linéaire et les transformations de programmes, de par la reécriture de la trace sous forme de nids de boucles imbriquées. A partir des applications issues de bancs d'essai, nous illustrons l'utilisation de notre modèle, et les optimisations qui peuvent en découler.

Nous montrons également, à l'aide du modèle polyédrique, différentes analyses du programme en boucles issu de l'interpolation périodique-linéaire, et comment ces analyses permettent de mieux comprendre, au besoin d'optimiser le programme de départ.

Nous donnons au chapitre 5 une vue d'ensemble du logiciel *PLI (Périodic-Linear Interpolation)* qui a été implémenté au cours de ce travail, ainsi qu'un bref manuel d'utilisation, et nous mentionnons également quelques pistes envisagées pour son évolution.

Le dernier chapitre présente les différentes perspectives générales qui se profilent pour la suite de nos travaux de recherche, et se termine par une conclusion sur l'ensemble de ce qui a été réalisé, et une évaluation par rapports aux objectifs fixés.

Chapitre 2

Techniques d'analyse de données : la fouille de données et les séries temporelles

Avant de se lancer dans la proposition d'une nouvelle approche d'analyse et de modélisation de traces de programmes, il semble judicieux d'examiner les techniques employées par les spécialistes de la fouille de données et de l'économétrie des séries temporelles. Cette activité nous a occupés au début de ce travail de thèse, et même auparavant lors du stage de DEA [48], durant lequel quelques-unes des approches décrites ci-dessous telles que les modèles de séries temporelles, et plusieurs autres ont été essayées sur des traces d'accès mémoire.

Ce chapitre est donc destiné à faire un tour d'horizon des principales techniques d'analyse de données, afin de mettre en évidence l'intérêt d'une méthode spécifique de modélisation périodique-linéaire du comportement des programmes.

2.1 La fouille de données

2.1.1 Contexte et objectifs

Comment trouver un diamant dans un tas de charbon sans se salir les mains ? tel est le slogan de nombreux logiciels de fouille de données largement répandus.

D'un point de vue historique, le développement de la fouille de données [10] (prospection, *ang. Data Mining*) fait suite au développement des moyens informatiques de calcul et de stockage. En effet, les entreprises qui archivaient des volumes de données considérables ont dès lors eu l'opportunité de valoriser ces informations, par des techniques d'analyse et d'exploration, afin d'améliorer leurs stratégies de marketing ou encore de disposer d'aides précieuses pour les processus de décision.

Actuellement, en réponse à l'accroissement continuel de la taille des bases de données (taille qui a été multipliée par un facteur de l'ordre de 10^6 au cours de ces dernières années), et à la complexité de leurs nouvelles architectures (passage des simples fichiers aux bases réparties dans des environnements hétérogènes, etc), les techniques d'exploration au cours de la fouille de données sont de plus en plus sophistiquées.

Généralement, la fouille de données se présente comme un domaine d'analyse exploratoire de données intégrant des outils de statistique et d'intelligence artificielle, de bases de données, de visualisation, au cours d'un processus d'extraction de connaissances. La fouille de données fait donc partie d'un processus général appelé Extraction de Connaissances dans les Données (**ECD** ou **KDD** : *Knowledge Discovery in Databases*), défini de la manière suivante [35].

Définition 2.1.1 *L'Extraction de Connaissances dans les Données (***ECD***) est un processus non trivial dont le but est d'extraire, à partir d'une grande masse de données souvent hétérogènes et structurées, des motifs (pattern) valides, nouveaux, potentiellement utiles et, à terme, compréhensibles.*

Les données en question sont des faits ou des observations et le terme *motifs* est à comprendre ici comme *propriétés décrivant un sous ensemble des données*, ou encore comme *modèles applicables aux données*. Extraire des motifs signifie donc découvrir des propriétés à partir des données ou encore trouver des modèles correspondant à ces données.

Selon Fayyad [35], le processus d'ECD comprend en fait plusieurs étapes, à savoir :

1. Une bonne compréhension du domaine d'application, étape qui consiste à expliciter les connaissances acquises *à priori*, ainsi que les buts du commanditaire.
2. La création d'un sous ensemble *cible* des données sélectionnées à partir de l'entrepôt de données.
3. Le nettoyage et le prétraitement : il faut éliminer les erreurs, déterminer les stratégies de traitement des données manquantes et des valeurs atypiques (*ang. outliers*).
4. La transformation des données : linéarisation, normalisation, compression, lissage, décomposition (ondelettes, Fourier) de courbes, etc.
5. L'harmonisation des buts (identifiés à l'étape 1) avec une méthode de fouille de données.
6. L'analyse exploratoire (analyse factorielle discriminante, analyse en composantes principales, etc), le choix des méthodes et des algorithmes en privilégiant interprétabilité ou prédictabilité.

7. La fouille de données, qui consiste à rechercher les motifs (*patterns*) intéressants.

8. L'exploitation des résultats : interprétation des motifs extraits (visualisations graphiques, etc). Parvenu à ce point, il est possible de reitérer le processus à partir de l'étape 1.

9. La diffusion des résultats pour aider à la prise de décision.

La fouille de données est donc l'étape de l'ECD qui consiste à découvrir automatiquement des motifs à partir des données. Ces motifs ou plus généralement les connaissances à extraire sont mises en exergue grâce à des techniques d'apprentissage automatiques, extrêmement variées et dépendant fortement des tâches à résoudre. Parmi ces tâches, on peut citer la prévision de l'évolution des marchés en analyse financière, la décision d'octroyer ou non des prêts dans le secteur bancaire, l'aide au diagnostic en médecine, l'analyse du panier de la ménagère en marketing, etc [51, 52].

Les principaux objectifs de la fouille de données consistent en :

- la description des données, qui se décompose elle-même en
  - l'exploration des données,
  - le classement et la classification (*clustering*) des observations,
  - la découverte des cas atypiques,
  - la découverte de hiérarchie et de dépendances entre les données,
  - l'analyse de séquences de données ;
- la prédiction, à travers la modélisation de variables cibles par un ensemble de variables explicatives.

C'est ainsi qu'on mettra en oeuvre, de manière explicite ou non :

- des méthodes de l'intelligence artificielle (réseaux de neurones, systèmes inductifs [54]),
- des méthodes de reconnaissance de formes (algorithmes évolutifs [53]),
- des méthodes statistiques (régression linéaire ou logistique, k plus proches voisins, arbres de classification et de regression [17], etc).

Il existe de nombreux algorithmes pour le déploiement des méthodes de fouille de données. Cependant, on reconnaît presque unanimement qu'il n'existe pas de *meilleur* algorithme, ni d'ailleurs de *meilleure* méthode, car il faut à chaque fois tenir compte de la nature de la tâche à résoudre. Afin de déterminer quelle méthode adopter et quel algorithme utiliser, il nous faut auparavant en faire un tour d'horizon.

2.2 Tour d'horizon des méthodes de fouille de données

On rencontre des méthodes de classement (discrimination, régression), dont le but est d'identifier des classes correspondant à des objets à partir de certains de leurs traits descriptifs. Cette identification découle d'un apprentissage qui s'effectue soit à partir de :

- bases d'instances : une instance est la description d'un objet,
- bases d'exemples : ensemble d'objets et des classes correspondantes,
- bases de cas : on prend une décision en partant d'un ou deux cas similaires déjà résolus et gardés en mémoire.

Dans tous ces cas de figure, la procédure générée doit classifier correctement les exemples de l'échantillon mais surtout avoir un bon pouvoir prédictif pour classifier correctement de nouvelles descriptions. Les termes utilisés pour désigner ces méthodes varient avec le domaine d'application considéré ; on parle de :

- *classification* en reconnaissance de formes ;
- *discrimination* en statistique ;
- *apprentissage de concepts* ou *apprentissage inductif* en apprentissage automatique ;
- *estimation* ou *régression* quand, au lieu de déterminer une classe on veut estimer une valeur continue.

Remarque 2.2.1 *Le terme* classement *est couramment remplacé dans la littérature francophone par* classification*, alors qu'il correspond en réalité au terme anglais* classification *qui se traduit bien par* classement *ou encore* discrimination*. L'amalgame vient du fait que le terme anglais correspondant pour* classification *est le mot* clustering *qui lui, correspond à une autre stratégie de fouille de données. C'est la raison pour laquelle on introduit souvent une autre appellation en français pour parler du* clustering*, il s'agit de l'expression* classification non supervisée.

Les algorithmes pour la mise en oeuvre de ces méthodes de classement reposent essentiellement sur les arbres de décision, les réseaux de neurones, la règle de Bayes, les k plus proches voisins, les classifieurs évolutifs.

Il existe également des méthodes connues sous le nom de *règles d'association* [11]. Leur but est de trouver des similarités ou des règles entre les données, régissant des associations ; en d'autres termes, elles décrivent et interprètent des modèles pour fournir de nouvelles connaissances.

Les algorithmes pour ces méthodes sont basés sur les règles d'association. On utilise le terme *sequencing* lorsque l'association doit se faire dans le temps.

Comme autre méthode, on a la classification (*ang. clustering*), encore appelée segmentation ou regroupement. Elle a pour but d'identifier un ensemble fini de classes dans les données, ou encore de découvrir des groupes dans ces données, et ces différentes classes peuvent être mutuellement exclusives ou plus complexes.

Afin de répartir les observations en classes ou catégories (*cluster*), on procède généralement par optimisation d'un certain critère visant à regrouper les données dans des classes aussi homogènes et distinctes entre elles que possible. Les principaux algorithmes utilisés en classification sont les cartes de Kohonen et les k-moyennes.

Contrairement aux méthodes de classement, les méthodes de classification (*clustering*) ne disposent pas au préalable d'exemples étiquettés ni de cas. C'est pourquoi on parle de **classification supervisée** pour désigner les méthodes de classement, et de **classification non supervisée** ou plus simplement **classification** pour le *clustering*.

2.2.1 Discussion et choix des méthodes

Définition 2.2.1 *On parle d'apprentissage supervisé lorsqu'il existe un échantillon (ou support) d'apprentissage.*

Les méthodes de discrimination, régression ou prédiction sont des méthodes d'apprentissage supervisé. Elles tentent d'élaborer un modèle qui explique le lien entre les données d'entrée et les résultats en sortie grâce à l'induction, associant une classe cible à partir d'un ensemble d'attributs descriptifs.

Or, dans notre étude, la trace d'exécution issue de l'analyse dynamique constitue le seul exemple (ou support d'apprentissage) dont nous disposons pour sa propre analyse. Cela signifie que de prime abord, on ignore complètement quelles sont les classes potentiellement identifiables. Par conséquent, l'apprentissage supervisé ne peut être appliqué.

Définition 2.2.2 *On parle d'apprentissage non supervisé lorsqu'il n'existe pas d'échantillon d'apprentissage.*

Etant donné que les méthodes d'apprentissage non supervisé permettent de **découvrir** des régularités sous forme de classes, de ressemblances, elles sont celles qui se rapprochent le plus de nos objectifs. Leurs tâches principales reposent sur :

- l'analyse d'associations,
- la découverte de classes,

- l'organisation en hiérarchie,
- la recherche d'anomalies.

Nous avons eu à développer au cours de notre travail quelques algorithmes basés sur la découverte de motifs fréquents et des règles d'association, dont l'application à l'analyse de traces est publiée en [25]. Cependant, dans le domaine de la fouille de données de manière fondamentale, c'est le *clustering* qui nous semble le plus à même de découvrir le maximum de régularités (sous forme de classes), en raison de la complexité et de la taille des traces à analyser.

Remarque 2.2.2 *Il existe aussi une autre forme d'apprentissage dite par renforcement, présentée notamment dans [62] avec plusieurs aspects algorithmiques de l'apprentissage automatique.*

2.3 Le clustering

Le clustering consiste à partitionner un ensemble d'individus ou d'éléments en classes (groupes, catégories ou clusters) homogènes, aussi distinctes les unes des autres que possible [46, 84]. La découverte des classes doit se faire automatiquement et sans exemple : il s'agit donc d'un apprentissage non supervisé.

Pour découvrir les classes, il faut auparavant établir une notion de proximité entre les éléments à classer, et chaque élément est généralement représenté sous forme d'un vecteur ligne de s colonnes, où chaque colonne est un attribut ou une caractéristique de l'individu.

2.3.1 Mesures de proximité

Soit $\mathcal{I}$ un ensemble d'individus. La proximité entre deux éléments de cet ensemble est soit une mesure de ressemblance ou similarité soit une mesure de dissemblance ou dissimilarité, définie de $\mathcal{I} \times \mathcal{I}$ vers $\mathbb{R}_+$.

Indices de ressemblance

Soit r l'indice de ressemblance et R un élément de $\mathbb{R}_+$; on a les caractéristiques suivantes :

$$r(i,j) = r(j,i),\ \forall (i,j) \in \mathcal{I} \times \mathcal{I} \ :\ \text{symétrie}\,; \tag{2.1}$$

$$r(i,i) = R > 0,\ \forall i \ :\ \text{ressemblance d'un élément avec lui-même}\,; \tag{2.2}$$

$$r(i,j) \leq R,\ \forall (i,j) \ :\ \text{la ressemblance est majorée par } R. \tag{2.3}$$

Le coefficient de corrélation est un exemple de mesure de similarité. Si on considère un ensemble de n éléments représentés par une matrice X de taille $n \times t$

$(n, t \in \mathbb{N})$, le coefficient de corrélation peut être calculé entre deux éléments ou plus souvent entre deux caractéristiques comme suit :

$$\rho(j,k) = \frac{\sum_{i=1}^{n}(x_{ij} - m_j)(x_{ik} - m_k)}{\left[\sum_{i=1}^{n}(x_{ij} - m_j)^2(x_{ik} - m_k)^2\right]^{1/2}} \tag{2.4}$$

où m_j et m_k sont les moyennes des vecteurs attributs j et k respectivement.

Indices de dissemblance

Soit d l'indice de dissemblance et D un élément de $\mathbb{R}_+$; d est caractérisé par les relations suivantes :

$$d(i,j) = d(j,i),\ \forall (i,j) \in \mathcal{I} \times \mathcal{I} \ :\ \text{symétrie}\,; \tag{2.5}$$
$$d(i,i) = 0,\ \forall i \ :\ \text{pas de dissemblance d'un élément avec lui-même}\,; \tag{2.6}$$
$$d(i,j) \leq D,\ \forall (i,j) \ :\ \text{la dissemblance est majorée par } D. \tag{2.7}$$

Il existe une correspondance triviale entre indice de ressemblance et indice de dissemblance, puisque $d(i,j) = R - r(i,j)\ \forall (i,j) \in \mathcal{I} \times \mathcal{I}$.

Métriques de Minkowsky

Ce sont des mesures de proximité assez souvent utilisées ; elles mesurent la dissemblance entre deux éléments $\mathbf{x_i} = (x_{i1}, ...x_{it})^T$ et $\mathbf{x_k}$ par :

$$d(i,k) = \left(\sum_{j=1}^{t} \left|x_{ij} - x_{kj}\right|^r\right)^{1/r} \quad \text{avec } r \geq 1$$

Ces métriques satisfont en plus les relations suivantes :

$$d(i,k) = 0,\ \text{ssi } \mathbf{x_i} = \mathbf{x_k} \tag{2.8}$$
$$d(i,k) \leq d(i,m) + d(m,k)\ \forall (i,k,m) \tag{2.9}$$
$$\tag{2.10}$$

Cas particuliers :

- $r = 2$: on a la distance euclidienne :

$$d(i,k) = \left[\sum_{j=1}^{t}\left(x_{ij} - x_{kj}\right)^2\right]^{1/2} = [(\mathbf{x_i} - \mathbf{x_k})^T(\mathbf{x_i} - \mathbf{x_k})]^{1/2}$$

- $r = 1$: on a la distance de manhattan

$$d(i,k) = \sum_{j=1}^{t} |x_{ij} - x_{kj}|$$

- $r \longrightarrow \infty$: distance sup

$$d(i,k) = \max_{j \leq 1 \leq t} |x_{ij} - x_{kj}|$$

De manière générale, les indices de proximités sont aussi représentés sous forme de matrice appelée **matrice de proximité**.

Remarque 2.3.1 *Bien qu'il existe également de méthodes de définition de distance entre valeurs qualitatives, nous nous limiterons pour notre travail aux distances entre valeurs numériques.*

2.3.2 Algorithmes de clustering

Le clustering étant un processus de partitionnement d'un ensemble d'individus en classes ou *cluster*, on peut, relativement à la distance entre valeurs numériques, définir un *cluster* comme suit :

Définition 2.3.1 *Un cluster est un ensemble de points tel que la distance entre deux points est inférieure à la distance entre un point du cluster et un autre qui ne lui appartient pas.*

Définition 2.3.2 *Un cluster est un ensemble d'entités semblables, tel que les entités des autres clusters leur soient dissemblables.*

Pour former les différentes partitions, les méthodes de clustering doivent mesurer non seulement la distance entre deux éléments, mais aussi la distance entre deux classes ou entre un élément et une classe ; elles doivent aussi définir un critère d'homogénéité des classes et en préciser le nombre.

Si on note $S(n,k)$ le nombre de partitions d'un ensemble de n éléments en k clusters, on remarque directement que $S(n,k)$ croît exponentiellement avec n $\left(S(n,k) = \frac{1}{k!}\sum_{i=1}^{k}(-1)^{(k-i)}(\mathrm{C}_k^i)(i)^n\right)$. C'est la raison pour laquelle une recherche exhaustive de toutes les partitions n'est pas possible ; à la place, les algorithmes de clustering procèdent par itérations successives jusqu'à obtenir une *bonne* partition, qui correspond en fait à un optimum local. Selon les stratégies de partitionnement utilisées, on distingue plusieurs méthodes de classification, à savoir : la classification ascendante, la classification descendante, la réallocation dynamique. Les deux premières méthodes sont dites hiérarchiques et construisent une série de partitions imbriquées ; elles se distinguent en cela de la troisième méthode, dite de partitionnement, qui ne produit qu'une seule partition. Il existe plusieurs variantes de toutes ces différentes méthodes dont une description peut être trouvée en [46].

La Classification Ascendante Hiérarchique (CAH)

On transfome la matrice de proximité en une séquence de partitions imbriquées [45, 46], en utilisant le procédé de l'agglomération : on met chaque élément dans un cluster individuel et ensuite on joint les clusters, pour aboutir finalement à un seule classe finale. Les regroupements successifs sont représentés sous la forme d'un arbre binaire ou *dendogramme*, et le nombre de classes est choisi à postériori, à la vue du dendogramme. L'algorithme est donné à la table 2.1.

1. **Initialisation.** Les classes initiales sont les singletons. Calculer la matrice de leurs distances deux à deux.
2. **Itération.** Répéter les deux étapes suivantes, jusqu'à l'agglomération en une seule classe :
   - regrouper les deux classes les plus proches au sens de la distance entre groupes choisie
   - mettre à jour le tableau de distance, en remplaçant les deux classes par la nouvelle et en calculant sa distance avec les autres classes

TAB. 2.1 – Algorithme de CAH

Exemple 2.3.1 *Soit $\mathcal{A}$ la matrice de dissimilarité pour les éléments x_1, x_2, x_3, x_4, x_5 :*

$$\mathcal{A} = \begin{array}{c} \\ x1 \\ x2 \\ x3 \\ x4 \\ x5 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \end{array} \\ \begin{bmatrix} 0 & 6 & 8 & 2 & 7 \\ 6 & 0 & 1 & 5 & 3 \\ 8 & 1 & 0 & 10 & 9 \\ 2 & 5 & 10 & 0 & 4 \\ 7 & 3 & 9 & 4 & 0 \end{bmatrix} \end{array}$$

Si on choisit pour distance entre deux groupes la distance minimale entre leurs éléments, on a le dendogramme de la figure 2.1, et si on choisit la distance maximale, on obtient celui de la figure2.2.

Le dendogramme donne une vue simple et claire des différentes classes identifiées, mais il devient impraticable au delà de quelques centaines d'éléments ; de plus, la nécessité de stocker en mémoire la matrice de proximité limite le nombre d'éléments qu'on peut traiter par la CAH.

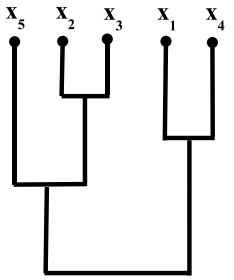


FIG. 2.1 – saut minimum

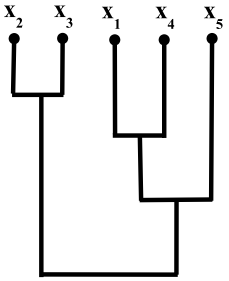


FIG. 2.2 – saut maximum

La Classification descendante

Le principe consiste à diviser l'ensemble initial des données pour arriver à des sous-classes finales qui soient aussi denses et aussi distinctes les unes des autres que possible. Il faut pour cela trouver un critère de scission qui minimise la dispersion des sous-classes résultantes [49]. En général, la classification descendante est très peu utilisée, car elle est considérée comme moins efficace que la **CAH** [33].

La réallocation dynamique

Dans cette méthode de classification, il faut fixer à priori k, le nombre de classes, car le procédé consiste à choisir k centres de classes et à affecter à chaque classe les élements les plus proches au sens de la distance choisie. Initialement, les k centres peuvent être tirés aléatoirement ou choisis grâce à des heuristiques ; par la suite, les nouveaux centres seront déterminés en calculant le barycentre de chaque classe. On itère ce procédé jusqu'à la convergence vers un minimum local ou jusqu'à un nombre maximum fixé d'itérations (table 2.2).

L'algorithme des k-moyennes (*kmeans*) [69], qui est très souvent utilisé, est une variante dans laquelle les centres de gravité des classes sont recalculés à chaque allocation d'un individu à une classe.

Remarque 2.3.2 *Il est possible de combiner les deux méthodes de classification dans une approche mixte. Par exemple, les k centres de classes de la réallocation dynamique peuvent être choisis grâce à une CAH. Cette combinaison permettra de classer les grands volumes de données (ce qui est impossible avec la CAH) et de sélectionner k, le nombre de classes.*

2.3.3 Application du clustering à l'analyse des traces d'exécution

Comme nous l'avons dit dans le chapitre précédent, les traces d'exécution que nous avons à traiter sont constituées essentiellement de valeurs entières, et sont

1. **Initialisation.** Choisir une partition initiale de k classes, en sélectionnant k points (appelés centres ou noyaux) dans l'espace des individus.
2. **Itération.** Répéter les deux étapes suivantes, jusqu'à la stabilisation des classes (le critère de variance inter classes ne croît plus de façon significative)
   - assigner chaque individu à la classe de centre le plus proche au sens de la distance choisie. On obtient ainsi, à chaque étape, une classification en k classes ou moins, si une des classes devient vide.
   - calculer le barycentre de chaque classe, qui devient le nouveau noyau. Si une classe s'est vidée, on peut éventuellement retirer aléatoirement un noyau complémentaire.

TAB. 2.2 – Algorithme de réallocation dynamique

d'assez grande taille. Quelques unes d'entres elles sont disponibles à l'adresse [7]. Pour de telles données ordinales, la classification ascendante hiérarchique pourrait assez bien convenir, mais il se trouve que la taille des traces constitue un handicap non négligeable pour ce type de méthode, notamment pour la manipulation des dendogrammes.

La méthode de réallocation dynamique quant à elle nécessite de définir une distance sur les données à classifier, et convient donc assez bien lorsque celles-ci sont sous forme de vecteurs. Cette méthode, et plus précisément l'algorithme des k-moyennes, est appliquée avec succès dans certains travaux portant sur l'analyse de programmes. En guise d'illustration, nous présentons dans la partie suivante la démarche mise en œuvre par Brad Calder *et al.* dans l'identification des phases de programmes et des points de simulation [41, 73]. Cette identification est basée sur la décomposition du programme en vecteurs de blocs de bases (BBV) et se déroule brièvement comme suit.

Définition 2.3.3 *Un bloc de base est une portion de code avec une seule entrée et une seule sortie, s'exécutant du début à la fin sans aucun branchement.*

On divise le temps d'exécution du programme en intervalles de taille fixée (un intervalle doit être compris ici comme un ensemble d'instructions). Pour chacun de ces intervalles, on construit un vecteur de blocs de base (BBV), c'est à dire un tableau à n entrées, n étant le nombre de blocs de base du programme entier. Chaque

entrée du vecteur indique la fréquence d'exécution du bloc de base correspondant, i.e le nombre d'exécutions de ce bloc multiplié par le nombre d'instructions qu'il contient.

Etant donné que la distribution des blocs de base pendant un intervalle est une indication du comportement du programme, on dit que deux BBV similaires correspondent à deux intervalles durant lesquels le comportement du programme est identique. Dans ce contexte précis, une phase se définit comme un ensemble d'intervalles, contigus ou non, durant lesquels une métrique est stable. Par conséquent, deux BBV similaires correspondent à deux intervalles faisant partie d'une même phase du programme.

Dans ces travaux, l'index de similarité utilisé entre deux vecteurs est la distance euclidienne, et les distances sont stockées dans une matrice de similarité. Une fois l'algorithme de classification appliqué sur les BBV, on obtient un regroupement des intervalles en k classes, correspondant aux phases du programme.

La sélection initiale de k (nombre de classes) se faisant aléatoirement, on teste plusieurs valeurs de k, et on retient à la fin la valeur qui donne le meilleur critère d'information de Bayes (BIC) [68].

Cet exemple montre bien comment on peut utiliser le *clustering* pour l'analyse des traces, quoique l'algorithme utilisé relève davantage des méthodes statistiques de classification plutôt que des méthodes neuronnales ou autres. Pour notre part, dans l'application de ce type de méthodes, nous avons été confrontés à une limitation située dans le format ou la structure des éléments à classifier.

En effet, les éléments des traces que nous traitons sont des scalaires et pas des vecteurs. Nous avons tout de même essayé de reformater les traces, en rajoutant des attributs (dimensions supplémentaires) à chaque observation pour les transformer en vecteurs. Comme attribut supplémentaire, il y a par exemple la position de l'élément dans la trace. Mais la notion de proximité reste assez malaisée à définir dans ce cas : deux valeurs identiques séparées par plusieurs observations vont apparaître plus ou moins distantes, à cause du nombre d'observations qui les séparent, et ne sont donc plus trouvées identiques en raison de leur valeur proprement dite.

Bien que des méthodes de classification différentes comme les méthodes neuronnales n'aient pas pu être testées, nous pouvons tout de même faire remarquer que la nécessité de définir un index de proximité approprié, requis pour la réallocation dynamique, constitue une difficulté considérable pour le cas de nos traces. D'autre part, en se limitant juste à un regroupement des éléments de la trace (c'est à dire aux clusters résultants), on n'obtient pas de fonction représentative de la trace analysée, et dans ces conditions, on n'atteint pas l'objectif d'analyses ultérieures dont le but est de transformer ou optimiser le programme de départ.

C'est pourquoi nous avons recherché d'autres techniques d'analyse, qui puissent opérer sur des données entières et scalaires. Ces données, étant des suites d'observations recueillies successivement au cours du temps, s'apparentent assez aux séries chronologiques encore appelées séries temporelles.

2.4 Les séries temporelles

2.4.1 Présentation

La démarche qui consiste à extraire d'un ensemble de données ce qui est reproductible et peut fonder une prédiction est une démarche assez largement répandue ; c'est ainsi qu'on la retrouve dans des domaines aussi variés que la géographie, l'économie, la climatologie, la démographie, etc. Le plus souvent, ces données en question sont des observations successives réalisées au cours du temps, et constituent une série temporelle [20, 42].

Définition 2.4.1 *Une série temporelle est une suite d'observations* $(x_t,\ t \in \mathcal{T})$ *d'une variable* x *à différentes dates* t. $\mathcal{T}$ *est dénombrable et* $t = 1, \ldots, T$.

En guise d'exemples, nous pouvons citer la série temporelle des orbites des planètes : grâce aux nombreuses données précises recueillies sur les orbites, Johannes Kepler a pu formuler la loi du mouvement des planètes qui porte son nom. Les économistes pour leur part utilisent les séries temporelles de prix, de taux d'intérêts et de masse monétaire pour étudier les "cycles d'affaires", les fluctuations d'activités commerciales, etc, et déterminer la périodicité des crises commerciales. Dans le domaine de l'industrie, la série temporelle de consommation électrique d'une société permet, connaissant la consommation pour une période donnée, de prédire celle du jour suivant.

2.4.2 Objectifs

L'étude des séries temporelles s'intéresse donc à la dynamique d'une variable observée, avec notamment trois objectifs majeurs qui sont la modélisation, la prédiction et la caractérisation [15] :

- la modélisation cherche à trouver une description précise du comportement à long terme de la variable observée. Elle peut se faire sous forme descriptive en identifiant les composantes de la série, ou alors se baser sur la notion de processus stochastique ;

- la prédiction cherche à évaluer précisément l'évolution à court terme de la variable observée en fonction de ses valeurs passées, partant de l'hypothèse que le futur sera comme le passé ;

- la caractérisation a comme objectif de déterminer les propriétés fondamentales du système sous-jacent.

En rapport avec l'analyse des traces d'exécution, c'est l'objectif de modélisation qui se trouve être le plus intéressant. Cette modélisation des séries temporelles peut être abordée soit par une approche descriptive, soit par un approche basée sur les processus aléatoires [42].

2.4.3 Approche descriptive

Elle consiste à mettre en lumière les différentes composantes de la série.

Composantes d'une série temporelle

Ce sont la tendance, le cycle, la saisonnalité et le bruit (composante accidentelle). Elles apparaissent dans la décomposition de la manière suivante :

$$x_t = T_t + C_t + S_t + u_t \tag{2.11}$$

- T_t est la composante tendancielle. Elle représente une variation lente de la série dans un sens donné et correspond le plus souvent à une fonction monotone croissante ou décroissante.

- S_t est la composante saisonnière ou périodique, elle correspond à l'évolution à court ou moyen terme de la série.

- C_t est la composante cyclique, beaucoup plus liée à la notion de "cycle d'affaires" en économie.

- u_t la composante conjoncturelle i.e. la partie purement aléatoire de la série, résultant des perturbations.

On dit que la tendance et la saisonnalité expliquent l'espérance de la série, tandis que la composante conjoncturelle en exprime la variabilité (variance et covariance). Les composantes tendancielle et saisonnière doivent donc être séparées pour permettre de mieux comprendre la série, et ensuite éliminées pour permettre d'observer plus clairement le phénomène aléatoire sous-jacent.

Identification et élimination des composantes

La détermination de la tendance se fait grâce aux méthodes de moyenne mobile, médiane mobile, regréssion linéaire (ajustement de la tendance). Pour l'éliminer, une fois qu'elle a été identifiée on la soustrait à la série initiale et on a alors $x_t - T_t$. On peut aussi utiliser les différences d'ordre 1 ou plus. Soit Δ_t l'opérateur de différence. La différence d'ordre 1 correspond à $\Delta_t = x_t - x_{t-1}$ et la différence d'ordre 2 correspond à

$$\Delta_t^{'} = \Delta_t - \Delta_{t-1} = (x_t - x_{t-1}) - (x_{t-1} - x_{t-2}) = x_t - 2x_{t-1} + x_{t-2}$$

La composante saisonnière pour sa part peut être modélisée analytiquement par des fonctions trigonométriques comme $sin(wt + \phi)$, si on pense que le modèle correpond à une fonction périodique connue. Elle peut aussi être directement identifiée à la vue des coefficients de corrélation, car l'ordre de la valeur maximale de ce coefficient indique la longueur de la saison (période). Enfin, la saisonnalité peut être identifiée à travers une analyse spectrale (construction du périodogramme). L'élimination de cette composante se fait par soustraction, ou en appliquant à la série un filtre différence d'amplitude s, s étant la longueur de la saison : $\Delta t_s = x_t - x_{t-s}$

L'approche descriptive permet donc d'identifier les composantes d'une série temporelle. Toutefois, outre ces composantes, il existe des caractéristiques permettant d'en appréhender la modélisation sous un autre aspect.

2.5 Caractéristiques des séries temporelles

Avant de donner les différentes caractéristiques d'une série temporelle, il est intéressant de noter le lien fondamental qui existe entre une série temporelle et un processus stochastique. En effet, la modélisation d'une série temporelle consiste essentiellement à établir une formulation statistique représentant le processus stochastique (inconnu) générateur de ladite série.

2.5.1 Processus stochastiques

Définition 2.5.1 *Un processus stochastique est une séquence de variables aléatoires* $\ldots X_{-2}, X_{-1}, X_0, X_1, X_2, \ldots$ *ordonnées dans le temps et définies à des points du temps qui peuvent être discrets ou continus.*

D'après cette définition, une série temporelle peut être assimilée à un processus stochastique, à la différence près que les inférences statistiques pour la série sont basées sur un seul jet fini de données (*finite strech data*). Pour cette raison, les observations d'une série sont considérées comme des réalisations de variables aléatoires d'un processus stochastique générateur. Pour modéliser sa distribution (inconnue en pratique), on s'intéresse à sa distribution conditionnelle via sa densité

$f(x_t|X_{t-1})$, ce qui revient à exprimer x_t en fonction de ses valeurs passées.

Ainsi, dans le traitement d'une série temporelle, l'identification de la structure du processus stochastique générateur est une étape clef. Ces processus peuvent être de plusieurs natures, selon qu'on considère des modèles linéaires (ARMA pour Auto-Regressive et Moving Average), conditionnels (ARCH pour Auto-Regressive Conditional Heteroskedasticity), ou encore dynamiques (processus de Markov). Afin de pouvoir déterminer ces différentes structures, il faut étudier les caractéristiques de la série considérée.

2.5.2 Stationnarité

La stationnarité est une notion fondamentale dans l'étude des séries temporelles, en l'absence de laquelle plusieurs méthodes d'inférence se trouveraient d'ailleurs invalidées.

Définition 2.5.2 *(stationnarité de premier ordre)*
Une série temporelle x_t est dite strictement stationnaire si le processus stochastique associé l'est aussi, i.e. la distribution conjointe de $(X_{t_1}, \ldots, X_{t_k})$ est identique à celle de $(X_{t_1+h}, \ldots, X_{t_k+h})$, quelque soit h, $k \in N$. En d'autres termes, la distribution conjointe des variables aléatoires dans la séquence est constante pout tout décalage de temps.

En pratique, très peu de séries sont stationnaires et on définit une stationnarité de second ordre, déterminée par les moments d'ordre un (espérance) et deux (variance et covariance). Intuitivement, une série stationnaire est un mouvement qui fluctue autour d'une valeur moyenne constante.

Définition 2.5.3 *Un processus stochastique $(X_t, t \in \mathbb{Z})$ est dit stationnaire d'ordre deux si*

- $E(X_t) = m,\ m$ *constante*, $\forall t$
- $E(|X_t|^2) < \infty, \quad \forall t$
- $COV(X_t, X_{t+k}) = E[(X_t - m)(X_{t+k} - m)] = \gamma_k, \ \forall t, \ \forall k$

Cela revient à dire que les espérances $E(X_t)$ des variables X_t sont égales, les moments de second ordre (et donc les variables X_t) ne doivent pas prendre des valeurs infiniment grandes, et la corrélation entre deux variables X_t et X_{t+k}) ne dépend que de l'intervalle k entre leurs instants respectifs. La stationnarité est une caractéristique qui se conserve par combinaison linéaire.

Propriété 2.5.1 *Si X_t est un processus stationnaire et si a_i est une suite de nombres réels absolument sommables i.e $\sum_{i=-\infty}^{\infty} |a_i| < \infty$, alors le processus linéaire Y_t suivant est stationnaire*

$$Y_t = \sum_{i=-\infty}^{\infty} a_i X_{t-i}$$

2.5.3 Autocovariance

Soit $(X_t, t \in \mathbb{Z})$ un processus stochastique. La fonction d'autocovariance de X_t pour deux instants r et s est la covariance des variables X_r et X_s notée $\gamma(r,s) = COV(X_r, X_s)$. Pour la série x_t, cette fonction est donnée empiriquement par

$$\gamma_k = \frac{\sum_{t=1}^{n-k}(x_t - m)(x_{t+k} - m)}{n-k},\ k \in \mathbb{Z} \tag{2.12}$$

où n est le nombre d'observations de la série et m sa moyenne.

2.5.4 Autocorrélation

Pour faciliter l'interprétation de l'autocovariance, on utilise la fonction d'autocorrélation, qui indique le degré de relation entre une série et elle même décalée dans le temps ; pour la série x_t et pour un décalage k, cette fonction est donnée par

$$\rho_k = \frac{\gamma_k}{\gamma_0} = Cor(x_{t+k}, x_t),\ \ k \in \mathbb{Z} \tag{2.13}$$

Il existe aussi une fonction d'autocorrélation partielle notée τ_k et définie comme suit.

Définition 2.5.4 *On appelle coefficient d'autocorrélation partielle de retard k la mesure de la corrélation directe qui existe entre x_t et x_{t-k} une fois éliminées les influences de $x_{t-1}, x_{t-2}, \ldots, x_{t-k+1}$ sur x_t.*

Le tracé de la fonction d'autocorrélation représente le *corrélogramme*, lequel permet entre autres de déceler la présence de la composante saisonnière ou périodicité (voir figure 2.3), et aussi d'identifier les différents modèles linéaires des processus. Dans la suite nous nous limiterons à ces modèles linéaires des séries temporelles, appelés **ARMA**.

2.6 Modélisation ARMA

Un processus ARMA est composé de deux processus de type AR (*AutoRegressive*) et MA (*Moving Average*) respectivement, chacun faisant intervenir un processus particulier appelé bruit blanc.

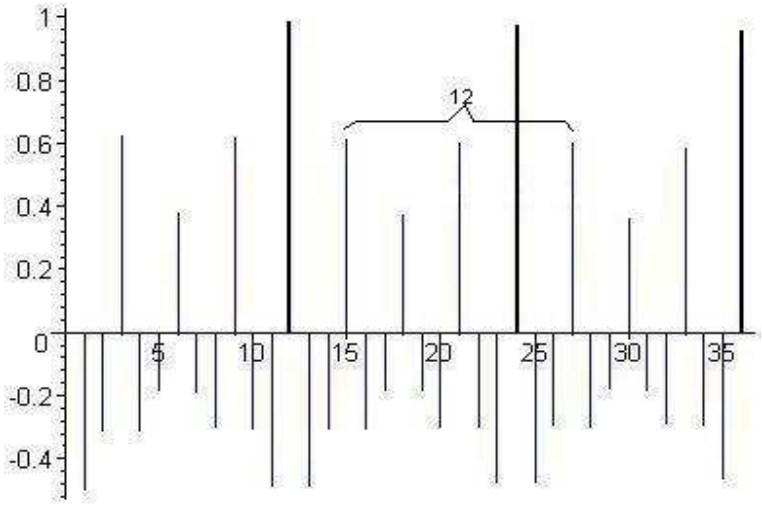


FIG. 2.3 – Corrélogramme indiquant une périodicité de longueur 12

Définition 2.6.1 *Un bruit blanc est un processus dont les espérances sont nulles, les variances constantes et les variables non autocorrélées.*

Etant donné qu'on ne sait pas grand-chose de la loi de distribution de ces variables aléatoires, on choisit de dire qu'elles sont indépendantes et identiquement distribuées. Pour une écriture simplifiée des processus ARMA, on définit l'opérateur retard noté L par :

$$Lx_t = x_{t-1} \tag{2.14}$$

$$L^2 x_t = L(Lx_t) = L(x_{t-1}) = x_{t-2} \tag{2.15}$$

$$L^n x_t = x_{t-n}, n \in \mathbb{Z} \tag{2.16}$$

2.6.1 Processus MA

Un processus $(X_t, t \in \mathbb{Z})$ suit un modèle MA d'ordre q s'il vérifie une relation du type :

$$X_t = \epsilon_t - \theta_1 \epsilon_{t-1} - \theta_2 \epsilon_{t-2} - \ldots - \theta_q \epsilon_{t-q} \tag{2.17}$$

avec $\theta_1, \theta_2, \ldots \theta_q \in \mathbb{R}$, ϵ_t est un bruit blanc. Un processus MA(q) est stationnaire par définition, car c'est un cas particulier de processus MA(∞) qui sont des séries linéaires causales, et donc stationnaires d'après la propriété 2.5.1. Avec l'opérateur retard, un $MA(q)$ sécrit

$$X_t = (1 - \theta_1 L - \theta_2 L^2 - \ldots - \theta_q L^q)\epsilon_t = \Theta(L)\epsilon_t \tag{2.18}$$

Processus MA(1)

$$X_t = \epsilon_t - \theta\epsilon_{t-1} \tag{2.19}$$

Le coefficient d'autocorrélation est

$$\rho_k = \begin{cases} \frac{-\theta}{1+\theta^2} & \text{si } k = 1 \\ 0 & \text{si } k > 1 \end{cases} \tag{2.20}$$

Le coefficient d'autocorrélation partielle est donné par

$$\tau_k = \frac{-\theta^k(1-\theta^2)}{1-\theta^{2(k+1)}} \tag{2.21}$$

Corrélogramme

Le corrélogramme partiel d'un processus MA(q) se caractérise par une décroissance géométrique de ses termes vers 0, tandis que son corrélogramme simple s'annule au delà des q premiers termes.

2.6.2 Processus AR

Un processus $(X_t, t \in \mathbb{Z})$ suit un modèle AR d'ordre p s'il vérifie une relation du type

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \epsilon_t \tag{2.22}$$

où $\phi_1, \phi_2, \ldots \phi_p \in \mathbb{R}$, ϵ_t est un bruit blanc. Un processus $AR(p)$ peut être transformé en $MA(\infty)$ sous certaines conditions, et est stationnaire dans ces cas là. La stationnarité repose ici sur l'invertibilité du polynôme retard, étant donné que

$$(1 - \phi_1 L - \phi_2 L^2 - \ldots - \phi_p L^p)X_t \;=\; \Phi(L)X_t \;= \epsilon_t \tag{2.23}$$

L'équation caractéristique associée à $\Phi(L)$ est $1 - \phi_1 z - \phi_2 z^2 - \ldots - \phi_p z^p = 0$ et $\Phi(L)$ est inversible si les modules des racines de cette équation sont supérieurs à 1.

Processus AR(1)

$$X_t = \phi X_{t-1} + \epsilon_t \tag{2.24}$$

On peut encore écrire $X_t - \phi L X_t = (1 - \phi L)X_t = \epsilon_t$.
Considérons la série de terme général $(\phi L)^n, n \in N$. On a la propriété

$$\sum_{i=0}^{\infty} \phi^i L^i = \frac{1}{1-\phi L}$$

si $-1 < \phi < 1$. Dans ce cas l'AR(1) s'écrit

$$X_t = \sum_{i=0}^{\infty} \phi^i \epsilon_{t-i}$$

et le processus est stationnaire.

Si $|\phi| = 1$ le processus est non stationnaire et ses variations $X_t - X_{t-1}$ sont imprévisibles : ce processus est appelé *marche aléatoire*.

Si $|\phi| > 1$ le processus est explosif.

Corrélogramme

De manière générale, le corrélogramme d'un processus AR(p) se caractérise par une décroissance géométrique de ses termes vers 0, tandis que son corrélogramme partiel s'annule au delà des p premiers termes, ce qui est le contraire pour un $MA(q)$.

2.6.3 Processus ARMA

Un processus $ARMA(p, q)$ est la combinaison d'un processus $AR(p)$ et d'un processus $MA(q)$. Il se note

$$\Phi(L)X_t = \Theta(L)\epsilon_t$$

Les processus ARMA sont représentatifs des séries stationnaires, mais il en existe plusieurs séries qui ne sont pas stationnaires.

2.6.4 Processus ARIMA

Un processus ARIMA (*AutoRegressive Integrated Moving Average*) résulte de l'application de la non stationnarité au processus ARMA. Cette non stationnarité peut provenir de la présence d'une tendance ou d'une composante saisonnière.

Dans le premier cas, on a $x_t = f(t) + z_t$, où $f(t)$ est une fonction du temps et z_t un processus stochastique stationnaire. On dit alors que la série est *stationnaire en tendance*, de type TS.

Dans le second, la série s'écrit$(1 - L)^d x_t = \beta + \epsilon_t$, ϵ_t étant un processus stationnaire et $\beta \in \mathbb{R}$. On dit qu'elle est intégrée d'ordre d ou encore *stationnaire en différence*(DS). Ce processus suit donc un modèle ARIMA et d est appelé l'ordre de différentiation.

De manière générale on note $(1 - L)^d(1 - L^s)\Phi(L)X_t = \Theta(L)\epsilon_t$ un processus intégré de type ARIMA(s, p, d, q), s étant l'ordre de la saisonnalité.

2.7 Etapes de la modélisation

La modélisation d'une série temporelle basée sur la notion de processus comporte plusieurs étapes. Celles-ci consistent à déterminer, dans la famille des processus ARIMA, le modèle le plus représentatif du phénomène exprimé par la série étudiée. L'algorithme le plus utilisé est celui de Box et Jenkins [15] qui, dans les années 70, ont proposé une modélisation en trois étapes principales :

a) Identification du modèle
- on choisit la valeur de d pour la désaisonnalisation, i.e le nombre de fois qu'il faut différencier la série pour obtenir une série stationnaire. On utilise pour cela des test de stationnarité type Dickey-Fuller [32], et la lecture des autocorrélogrammes ;
- on choisit les valeurs de p et q, ordres respectifs des composantes AR et MA, en respectant le principe de parcimonie.

b) Estimation des paramètres du modèle à l'aide des méthodes statistiques (moindres carrés, maximum de vraisemblance, etc), on estime les paramètres ϕ_i et θ_i des composantes AR et MA respectivement.

c) Test et validation
- on effectue des tests d'hypothèse portant sur la nullité ou la significativité des coefficients estimés précédemment ;
- on analyse les résidus pour valider l'hypothèse de bruit blanc (tests Q, Q' [16, 60]).

  Si l'hypothèse de nullité d'un paramètre est acceptée, il convient d'invalider l'ordre correspondant et d'effectuer une nouvelle estimation. Si les résidus ne sont pas bruit blanc, on déduit que la spécification du modèle est incomplète et qu'il manque au moins un ordre au processus.

Cette dernière étape (test et validation) peut nécessiter un retour à la première (identification), et ainsi de suite.

2.8 Application de l'analyse des séries temporelles à l'analyse des traces

2.8.1 Exemples d'application

Nous avons étudié certains accès mémoire du programme Tsp (Traveling Salesman Problem) issu des *olden benchmarks* [19, 70], qui calcule une approximation du meilleur circuit hamiltonien pour le problème du *voyageur de commerce* ; ce programme utilise un algorithme de partitionnement et une heuristique pour identifier les points les plus rapprochés, et les villes à parcourir sont représentées par un arbre binaire balancé [9, 50]. Pendant que le programme s'exécute, nous enregistrons dans

un fichier l'ensemble des emplacements mémoires visités. Nous présentons ici les résultats obtenus pour $n = 148$ et $n = 530$, n étant le nombre de noeuds de l'arbre des villes. Les traces ont été analysées sous TSP (Time Series Processing), qui est entre autres un logiciel de traitement de séries temporelles [3].

Exemple 1 : $n = 148$

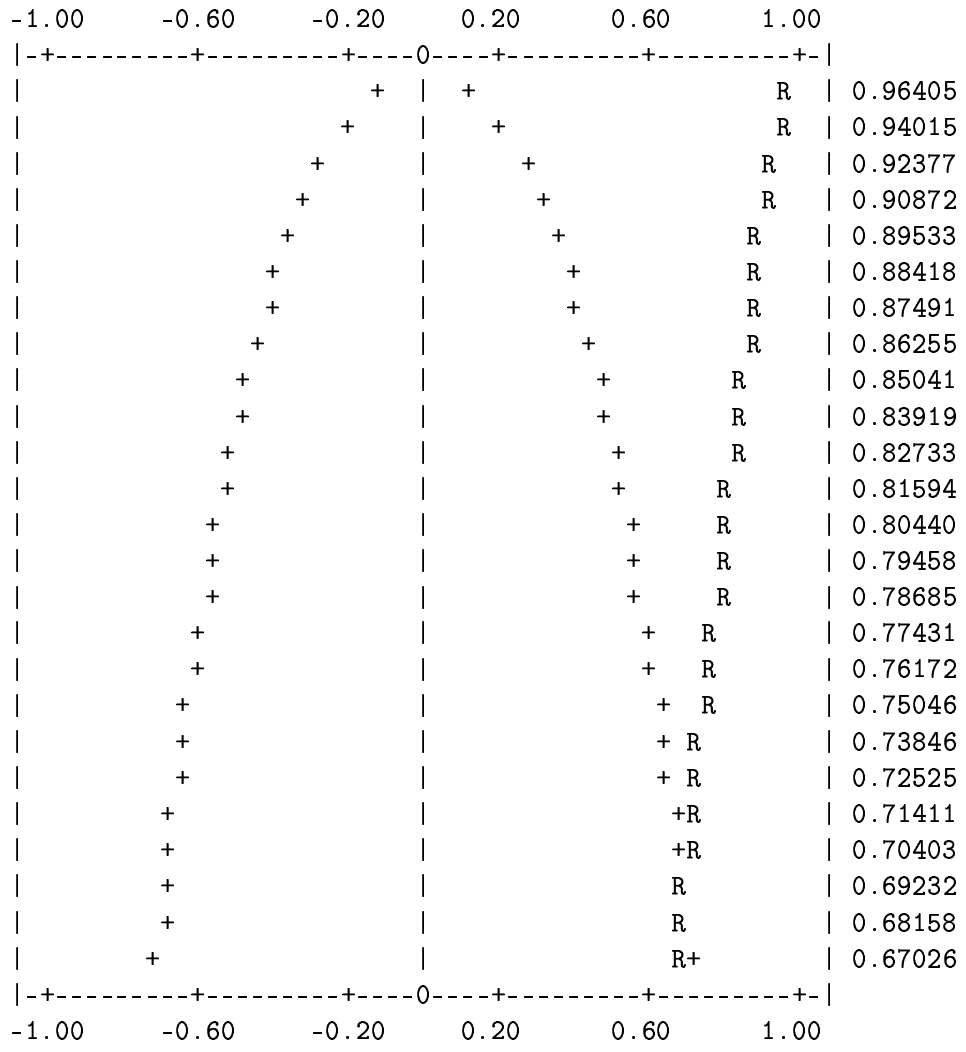
Remarque 2.8.1 *Avec le logiciel TSP utilisé, la lecture du graphique d'un corrélogramme s'effectue comme suit :*

- *les signes + + délimitent les bornes d'un intervalle de confiance ; chaque coefficient à l'extérieur de ces intervalles est donc significativement différent de 0 au seuil de 5%, et tous les coefficients à l'intérieur des intervalles sont significativement nuls ;*
- *les nombres listés à gauche des corrélogrammes représentent les ordres d'autocorrélation, tandis que ceux de droite représentent les valeurs calculées du coefficient d'autocorrélation ;*
- *les signes R correspondent à $\rho(k)$, valeur d'autocorrélation pour un ordre k donné.*

Dans ce premier exemple, la série étudiée comporte 255 observations. Son corrélogramme simple présente une lente décroissance vers 0 tandis que son corrélogramme partiel s'annule au delà du premier retard, comme on peut le voir sur les graphiques suivants.

```
-1.00     -0.60     -0.20     0.20      0.60      1.00
    |-+---------+---------+----0----+---------+---------+-|
 1  |                       +  |  +                   R  | 0.97482
 2  |                       +  R  +                      | -0.0033356
 3  |                       +  |R +                      | 0.038157
 4  |                       + R|  +                      | -0.042138
 5  |                       +  R  +                      | -0.0029570
 6  |                       +  |R +                      | 0.032036
 7  |                       +  |R +                      | 0.051339
 8  |                       +R |  +                      | -0.082817
 9  |                       +  R  +                      | -0.0062411
 10 |                       +  R  +                      | 0.015251
    |-+---------+---------+----0----+---------+---------+-|
   -1.00     -0.60     -0.20     0.20      0.60      1.00
```

Exemple 1 : Corrélogramme partiel



Exemple 1 : Corrélogramme simple pour 255 observations (trace Tsp)

Il pourrait s'agir bien d'un modèle $AR(1)$, seulement la décroissance des autocorrélations simples n'est pas très rapide. Pour en savoir plus, on utilise une stratégie de tests de stationnarité [32] consistant à tester les trois modèles suivants :

$$(1): \quad VAR_t - VAR_{t-1} = \phi VAR_{t-1} + \epsilon_t \tag{2.25}$$
$$(2): \quad VAR_t - VAR_{t-1} = \phi VAR_{t-1} + c + \epsilon_t \tag{2.26}$$
$$(3): \quad VAR_t - VAR_{t-1} = \phi VAR_{t-1} + c + bt + \epsilon_t \tag{2.27}$$

où VAR est la série étudiée, c une constante et b le coefficient de la tendance. On commence par tester le modèle 3, i.e. l'hypothèse de la non stationnarité ($\phi = 0$), avec présence d'une constante et d'une tendance.

Propriété 2.8.1 *Lors d'un test d'hypothèse $H0$ de non stationnarité par la méthode de Dickey-Fuller, il faut comparer la statistique de ϕ aux valeurs tabulées correspondant au modèle testé; si cette statistique t_ϕ est supérieure à la valeur tabulée, alors on accepte $H0$, sinon on la rejette.*

Pour le modèle 3 testé on obient les résultats suivants :

```
           Estimated    Standard
Variable   Coefficient     Error       t-statistic   P-value
TPS        .021974       .067462       .325719       [.745]
C          3.02017       2.08691       1.44720       [.149]
VAR(-1)    -.046050      .067462       -.682606      [.495]
DICKEY-FULLER(CT,ASY.,0)  Test Statistic: -0.6826058,
                          Lower tail area: .97427
```

On compare t_ϕ à la valeur tabulée par Dickey et Fuller [37] au seuil de 5% pour le modèle 3, elle vaut -3.42. Comme $t_\phi = -0.682606$ est supérieure à -3.42, on accepte l'hypothèse de racine unitaire : la série est non stationnaire. Pour valider qu'elle suit bien le modèle avec tendance, on effectue un test de Fisher ($(c, 0, 0)$) pour la nullité de b conditionnelle à celle de ϕ. On trouve :

```
           Estimated    Standard
Variable   Coefficient     Error       t-statistic   P-value
C          -.027559      1.03480       -.026632      [.979]
F(2,251)   Test Statistic: 1.574162, Upper tail area: .20922
```

Propriété 2.8.2 *Lors d'un test conjoint de Fisher pour la nullité de b ou de c conditionnelle à celle de ϕ, il faut comparer la statistique de F aux valeurs tabulées par Dickey et Fuller, correspondant au cas testé ($(c, 0, 0)$ ou $(0, 0)$); si cette statistique est inférieure à la valeur tabulée, alors on accepte l'hypothèse, sinon on la rejette.*

On compare la statistique 1.574162 au seuil tabulé pour les tests conjoints, qui vaut 6,30. Comme la statistique calculée est inférieure au seuil, on accepte l'hypothèse de nullité de b conditionnelle à celle de ϕ, ce qui a pour conséquence d'invalider le modèle 3 testé. On doit recommencer le test sur le modèle 2, et on obtient ce qui suit.

```
            Estimated    Standard
 Variable  Coefficient     Error       t-statistic   P-value
 C         3.10386       2.06736       1.50136       [.135]
 VAR(-1)   -.024560      .014056       -1.74730      [.082]
 DICKEY-FULLER(CT,ASY.,0)  Test Statistic: -1.747301,
                           Lower tail area: .72960
```

On compare $t_\phi = -1.747301$ à la valeur tabulée pour ce modèle $(-2,87)$ et on accepte l'hypothèse de non stationnarité. On cherche à valider le modèle par un test de Fisher $((0,0))$ sur la nullité de c conditionnelle à celle de ϕ, ce qui revient ici à un test de Wald sur la nullité jointe de tous les paramètres du modèle 2. On trouve :

```
Parameter  Estimate        Error       t-statistic   P-value
SUM        3.07930       2.05519       1.49830       [.134]

Wald Test for the Hypothesis that the given set of Parameters
are jointly zero:
CHISQ(1) =   2.2449159     ; P-value = 0.13405

F Test for the Hypothesis that the given set of Parameters
are jointly zero:
F(1,252) =   2.2449159     ; P-value = 0.13531
```

Comme la valeur de $F = 2,2449159$ inférieure à la valeur tabulée pour ce test $(4,61)$, on accepte l'hypothèse de nullité de c conditionnelle à celle de ϕ, ce qui invalide le modèle 2. On teste le modèle 1.

```
            Estimated    Standard
 Variable  Coefficient     Error       t-statistic   P-value
 VAR(-1)   -.626627E-02  .702462E-02   -.892044      [.373]
 DICKEY-FULLER(CT,ASY.,0)  Test Statistic: -0.8920441,
                           Lower tail area: .95700
```

On compare $t_\phi = -0.8920441$ à la valeur tabulée pour ce modèle $(-1, 95)$ et on accepte l'hypothèse de non stationnarité, ce qui veut dire que $\phi = 0$ dans l'expression $VAR_t - VAR_{t-1} = \phi VAR_{t-1} + \epsilon_t$. On déduit alors que la série est intégrée d'ordre 1 (ARIMA(0,1,0)), sans constante ni tendance. On le vérifie en analysant les résidus, et leurs corrélogrammes montrent qu'il s'agit bien de bruits blancs.

L'estimation par les moindres carrés de la série VAR_t nous donne un résultat conforme à l'analyse précédente :

Variable	Estimated Coefficient	Standard Error	t-statistic	P-value
VAR(-1)	.993734	.702462E-02	141.464	[.000]

Exemple 2 : $n = 530$

La série ici comporte 1020 observations. Le corrélogramme simple laisse apparaître la présence d'une forte tendance, ce qui se voit aux valeurs élevées des autocorrélations sur plusieurs ordres.

```
 -1.00     -0.60     -0.20     0.20      0.60      1.00
   |-+---------+---------+----0----+---------+---------+-|
1  |                        + | +                    R | 0.99728
2  |                        + | +  R                   | 0.21206
3  |                        + | +  R                   | 0.19119
4  |                        + | +R                     | 0.11583
5  |                        + | +R                     | 0.10923
6  |                        + | +R                     | 0.11837
7  |                        + | +R                     | 0.13018
8  |                        + |R+                      | 0.035738
9  |                        + |R+                      | 0.054105
10 |                        + |R+                      | 0.057369
   |-+---------+---------+----0----+---------+---------+-|
   -1.00     -0.60     -0.20     0.20      0.60      1.00
```

Exemple 2 : Corrélogramme partiel

```
    -1.00     -0.60     -0.20     0.20      0.60      1.00
    |-+---------+---------+----0----+---------+---------+-|
 1  |                        + | +                     R | 0.99549
 2  |                       +  |  +                    R | 0.99175
 3  |                       +  |  +                    R | 0.98852
 4  |                      +   |   +                   R | 0.98534
 5  |                     +    |    +                  R | 0.98228
 6  |                     +    |    +                 R  | 0.97938
 7  |                     +    |    +                 R  | 0.97664
 8  |                    +     |     +                R  | 0.97360
 9  |                    +     |     +                R  | 0.97060
 10 |                   +      |      +               R  | 0.96769
 11 |                   +      |      +               R  | 0.96471
 12 |                   +      |      +               R  | 0.96177
 13 |                  +       |       +              R  | 0.95885
 14 |                  +       |       +              R  | 0.95607
 15 |                  +       |       +              R  | 0.95346
 16 |                  +       |       +              R  | 0.95036
 17 |                 +        |        +             R  | 0.94730
 18 |                 +        |        +             R  | 0.94438
 19 |                 +        |        +             R  | 0.94138
 20 |                 +        |        +            R   | 0.93831
    |-+---------+---------+----0----+---------+---------+-|
    -1.00     -0.60     -0.20     0.20      0.60      1.00
```

Exemple 2 : Corrélogramme simple d'une trace de 1020 observations

On procède aux tests de non stationnarité pour vérifier cette intuition de départ, en commençant toujours par le modèle 3. On obtient les résultats suivants :

```
            Estimated    Standard
 Variable  Coefficient     Error       t-statistic   P-value
 TPS        .477046       .029425       16.2123       [.000]
 C         1.80683        1.21653       1.48523       [.138]
 VAR(-1)   -.478423       .029349       -16.3010      [.000]
 DICKEY-FULLER(CT,ASY.,0)  Test Statistic: -16.30096,
                           Lower tail area: .00000
```

Comme -16.30096 (la statistique du coefficient ϕ) est inférieure à la valeur tabulée qui vaut -3.41 au seuil de 5% pour 1020 observations, on rejette l'hypothèse de racine unitaire. On effectue ensuite un test simple de Student pour la nullité de b. La statistique $|16.2123|$ étant supérieure à 1.96, on rejette l'hypothèse de nullité de b. Par conséquent, il existe bien une tendance dans la série et elle est donc de type TS. On estime cette tendance par la méthode des moindres carrés et on trouve :

Variable	Estimated Coefficient	Standard Error	t-statistic	P-value
TPS	.998679	.236076E-02	423.034	[.000]
C	2.16870	1.39195	1.55804	[.120]

La valeur de c n'est pas significative ($|1.55804| < 1.96$). Après extraction de la tendance, on obtient les corrélogrammes ci-dessous.

```
    -1.00     -0.60     -0.20     0.20      0.60      1.00
    |-+---------+---------+----0----+---------+---------+-|
 1  |                        + | +        R               | 0.45461
 2  |                        + | + R                      | 0.17782
 3  |                        + | R                        | 0.060150
 4  |                        + R +                        | -0.018153
 5  |                        +R| +                        | -0.059180
 6  |                        +R| +                        | -0.057708
 7  |                        + R +                        | -0.019493
 8  |                        +R| +                        | -0.037700
 9  |                        +R| +                        | -0.049634
 10 |                        +R| +                        | -0.045294
 11 |                        +R| +                        | -0.053842
 12 |                        +R| +                        | -0.055637
 13 |                        +R| +                        | -0.059364
 14 |                        +R| +                        | -0.027975
 15 |                        + |R+                        | 0.048043
 16 |                        + R +                        | 0.017889
 17 |                        + R +                        | -0.0080153
 18 |                        + R +                        | -0.0038071
 19 |                        + R +                        | -0.015600
 20 |                        +R| +                        | -0.057674
    |-+---------+---------+----0----+---------+---------+-|
    -1.00     -0.60     -0.20     0.20      0.60      1.00
```

Corrélogramme simple

```
    -1.00     -0.60     -0.20     0.20      0.60      1.00
    |-+---------+---------+----0----+---------+---------+-|
 1  |                        + | +         R             | 0.48569
 2  |                        +R| +                       | -0.038611
 3  |                        + R +                       | -0.0072310
 4  |                        +R| +                       | -0.050339
 5  |                        +R| +                       | -0.038902
 6  |                        + R +                       | -0.0099476
 7  |                        + |R+                       | 0.026805
 8  |                        +R| +                       | -0.051802
 9  |                        +R| +                       | -0.027552
 10 |                        + R +                       | -0.015300
    |-+---------+---------+----0----+---------+---------+-|
    -1.00     -0.60     -0.20     0.20      0.60      1.00

                    Corrélogramme partiel
```

On conclut que la série diminuée de la tendance est donc stationnaire de type $AR(1)$, et s'écrit $RS_t = VAR_t - \alpha t = 0.532135 RS_{t-1} + \epsilon_t$

2.9 Discussion et conclusion

En considérant les traces de programmes comme des réalisations de variables aléatoires d'un processus stochastique, il est possible de les modéliser suivant l'approche des séries temporelles. Nous avons montré sur deux exemples comment y parvenir, et les informations obtenues peuvent s'avérer pertinentes. Par exemple, lorsque nous avons traité la trace avec $n = 148$ à la recherche de motifs symboliquement constants et répétitifs [25], il nous a fallu auparavant différencier les données pour voir apparaître un caractère régulier. Or l'analyse de cette trace par la méthode des séries temporelles conclut effectivement qu'il s'agit d'une série intégrée d'ordre 1. Par conséquent, il serait bien possible d'exploiter les informations issues de ce type d'analyse, en amont pour d'autres méthodes d'analyse ou d'autres types de recherche. Cela s'illustre encore plus clairement par les travaux décrits dans [77].

Toutefois, le processus de modélisation demeure assez complexe. En outre, les estimations effectuées tout au long de ce processus fournissent en sortie des valeurs réelles (et non entières), dont la réutilisation conduit seulement à des valeurs approchées de la trace initiale. Nous avons aussi noté l'émergence d'autres types d'approches [22, 59] qui cherchent à appliquer les méthodes du *Data mining* à l'identification des motifs dans les séries temporelles.

De manière générale, les résultats obtenus ne se prêtent pas aisément à une représentation synthétisée de la trace, ce qui aurait pourtant eu comme avantage de permettre de mieux saisir le comportement des programmes analysés et d'en déduire des transformations optimisantes.

Dans le chapitre suivant nous introduisons une méthode d'interpolation dite *périodique-linéaire* qui, grâce à la combinaison de différents facteurs, répond à ce besoin d'expressivité, de clarté et de flexibilité d'utilisation pour l'analyse des traces.

Chapitre 3

Le modèle périodique-linéaire

3.1 Motivations

Ainsi que nous l'avons dit en introduction, les méthodes d'analyse statique ont une portée limitée par les structures de contrôle et de données pouvant être analysées, et plus généralement par les informations connues, ou déduites de l'analyse, à partir du code source d'un programme. Ces méthodes statiques, telles que le modèle polyédrique, offrent pourtant un environnement riche permettant de nombreuses analyses et transformations de programmes exactes et symboliques. Par conséquent il serait bien dommage de ne pas pouvoir profiter de leurs possibilités dans le cadre de l'analyse d'informations extraites au cours des exécutions de programme.

D'autre part, l'analyse dynamique révèle une extrême variabilité du comportement des programmes au cours de leur exécution, mais aussi une périodicité intéressante qui reste identique lorsqu'on considère plusieurs métriques différentes [34, 74, 71]. Pour identifier les lieux ou les régions où se produisent ces régularités, on se sert habituellement de la trace d'exécution, et le traitement qu'on y applique tient compte de la nature des informations qu'elle contient : vecteurs de blocs de base (BBV), ensembles de travail (*instruction working set*), valeurs scalaires, etc.

L'analyse de traces d'exécutions consiste classiquement à appliquer des méthodes et modèles généraux en analyse de données, telles que les méthodes de classification rappelées au chapitre 2, ou les méthodes d'interpolation ou de régression mathématiques standards. Le pouvoir d'expressivité des modèles ainsi construits est lui-même assez général, soit donnant peu d'informations utiles, soit n'ouvrant pas directement vers des transformations optimisantes de code.

La solution que nous proposons et qui a été publiée dans [24] consiste à exprimer le comportement à l'exécution d'un programme par un autre programme, qui n'exprime donc pas la fonctionnalité, "le pourquoi", mais la manière, "le comment". Les langages informatiques offrent un pouvoir d'expressivité où la fonctionnalité

peut être transposée au comportement. Et par la suite, une analyse "statique" du programme représentant le comportement permet d'obtenir des informations adaptées aux objectifs, plus précisément qu'avec des méthodes générales d'analyse de données.

Dans ce travail, nous développons cette approche en représentant une trace d'exécution par des séquences de nids de boucles dans lesquelles les instructions des boucles les plus internes sont des fonctions exprimant les valeurs de la trace d'entrée à partir des indices de boucles. Les bornes des boucles sont soit des constantes, soit des fonctions affines des indices de boucles. Cette représentation en programmes de boucles est ensuite analysée via le modèle polyèdrique d'analyse statique.

Dans toute la suite, une phase de programme correspond à un comportement périodique-linéaire constant. Chaque boucle correspond à une phase, une phase étant elle-même un ensemble d'intervalles dépendants les uns des autres, identifiés à travers une méthode d'interpolation périodique-linéaire. Les boucles étant imbriquées, celles-ci définissent également une hiérarchie de phases.

3.2 Interpolation polynomiale

L'interpolation polynomiale classique de $n+1$ points calcule le polynôme de degré au plus égal à n qui passe exactement par chacun de ces points. C'est une fonction calculable, mais également complexe du fait de son degré, et difficilement exploitable dans notre contexte.

Exemple 3.2.1 *Soit une suite de valeurs* $[3,3,7,13,11,23,15,33,19,43,23]$*, observées aux instants* $x = 0,1,\ldots,10$ *respectivement. La fonction polynomiale d'interpolation de cette suite est donnée par*

$$\frac{-4}{2835}x^{10} + \frac{197}{2835}x^9 - \frac{277}{189}x^8 + \frac{16348}{945}x^7 - \frac{16912}{135}x^6 + \frac{77408}{135}x^5 - \frac{932752}{567}x^4 + \frac{7998976}{2835}x^3 - \frac{814336}{315}x^2 + \frac{59518}{63}x + 3$$

On voit bien que d'un point de vue informationnel cette fonction n'est pas très expressive, en particulier au sujet d'une possible périodicité de la suite traitée ; cette remarque s'étend d'ailleurs à toutes les fonctions d'interpolation polynomiales en général. Par contre, si l'on observe tour à tour les éléments dont les abscisses sont distantes de 2 dans la suite de départ, on s'aperçoit que tous ces termes sont liés par la relation $[2,5]x + [3,-2]$ ($2x+3$ si x est pair et $5x-2$ sinon), ce qui laisse clairement apparaître la périodicité de cette suite qui est de 2, et une relation linéaire entre termes distants de 2. C'est pourquoi, au lieu d'une interpolation polynomiale complexe et peu expressive, nous choisissons d'effectuer sur les traces une interpolation linéaire et périodique, dont la description suit.

3.3 Interpolation périodique-linéaire

En se restreignant aux polynômes de degré 1, une interpolation linéaire assure que les résultats soient simples, faciles à manipuler et à déployer. La périodicité du polynôme, pour sa part, permet de saisir et d'expliciter le caractère répétitif pouvant exister dans une suite. Pour déployer une telle interpolation périodique et linéaire, on fait intervenir des fonctions affines particulières appelées fonctions périodiques.

3.3.1 Fonction périodique

On définit auparavant ce qu'est un nombre périodique.

Définition 3.3.1 *Un nombre périodique est une suite finie de n valeurs numériques $[a_1, a_2, ..., a_n]_y$, dont la valeur correspond à celle dont le rang est donné par $y \ mod \ n$, $y \in \mathbb{N}$, i.e.*

$$[a_1, a_2, ..., a_n]_y = \begin{cases} a_1 & si \ y \ mod \ n = 0 \\ a_2 & si \ y \ mod \ n = 1 \\ ... & ... \\ a_n & si \ y \ mod \ n = n-1 \end{cases}$$

Le nombre de valeurs contenues dans un nombre périodique est appelé sa *période*. Ainsi le nombre $[a_1, a_2, ..., a_n]_y$ est de période n. Dans la pratique, le domaine de définition de l'indice y est un sous-ensemble fini de $\mathbb{N}$, ce qui permet de limiter les répétitions des termes du nombre périodique. Pour une présentation plus détaillée des nombres périodiques on peut se référer à [61].

On a aussi la propriété suivante, qui donne quelques opérations entre deux périodiques ou entre un périodique et un scalaire.

Propriété 3.3.1 *Soit λ un scalaire, a et b deux nombres périodiques de période n. On a :*

$$\lambda = [\lambda, \lambda, ..., \lambda]_t, \forall t, \ \ t \ fixé$$

$$\lambda a = a\lambda = [\lambda a_1, \lambda a_2, ..., \lambda a_n]$$

$$a + b = [a_1 + b_1, a_2 + b_2, ..., a_n + b_n]$$

On peut à présent définir une fonction périodique de la manière suivante.

Définition 3.3.2 *Une fonction périodique f est une fonction affine de la forme $f(x) = ax + b$, où les coefficients a et b sont des nombres périodiques.*

Pour effectuer des opérations sur des fonctions périodiques, on doit réduire leurs coefficients périodiques à la même période, ce qui peut toujours se faire en utilisant le plus petit commun multiple (*ppcm*) des périodes respectives des coefficients.

3.3.2 Somme de fonctions périodiques

Soient f et g deux fonctions périodiques de période m et n respectivement, telles que

$$f(x) = [a_1, a_2, ..., a_m]x + [b_1, b_2, ..., b_m]$$

$$g(x) = [a'_1, a'_2, ..., a'_n]x + [b'_1, b'_2, ..., b'_n]$$

On désire calculer $(f + g)(x)$. Pour cela, on doit ramener les nombres périodiques a, b, a', b' à la même période $p = ppcm(m, n)$ de manière à ce que $f(x) = [s_1, s_2, ..., s_p]x + [b_1, b_2, ..., b_p]$ et $g(x) = [s'_1, s'_2, ..., s'_p]x + [b'_1, b'_2, ..., b'_p]$. On procède comme suit :

- on constitue b (respectivement b'), qui est égal aux p premières valeurs de $f(x)$ (respectivement $g(x)$), soient $f(0), \ldots, f(p-1)$;
- on constitue s grâce aux valeurs de a, multipliées par $ppcm(m, n)/m$ et répétées $ppcm(m, n)/m - 1$ fois ;
- de même, le nombre périodique s' est constitué des valeurs de a', multipliées par $ppcm(m, n)/n$ et répétées $ppcm(m, n)/n - 1$ fois.

Finalement on obtient :

$$(f + g)(x) = [s_1 + s'_1, s_2 + s'_2, ..., s_p + s'_p]x + [b_1 + b'_1, b_2 + b'_2, ..., b_p + b'_p]$$

Grâce aux fonctions périodiques et aux opérations présentées ci-dessus, nous pouvons décrire l'interpolation périodique.

3.3.3 Interpolation périodique-linéaire

Soit une suite S de m valeurs. L'interpolation périodique-linéaire consiste à décomposer la suite S en intervalles successifs disjoints tels que tout élément e_{ij} se trouvant à la position j d'un intervalle i soit linéairement dépendant de l'élément $e_{i-1,j}$ de l'intervalle précédent, c'est à dire $e_{ij} = e_{i-1,j} + a_j$, avec a_j constant (voir figure 3.1).

La fonction périodique résultante est de la forme $f(x) = ax + b$, et le nombre d'éléments dans chaque intervalle correspond au plus petit commun multiple des périodes des coefficients a et b. On le note p. L'ensemble des valeurs possibles de p est égal à $\{1, 2, \ldots, m/3\}$, afin d'éviter le cas trivial $p = m/2$ et donc d'interpoler au minimum trois points de la trace.

A priori, on ne connaît pas la valeur de p pour laquelle il existe une dépendance linéaire entre intervalles successifs. Cependant, on sait que la fonction d'autocorrélation établit la relation linéaire entre les termes d'une suite ou série temporelle

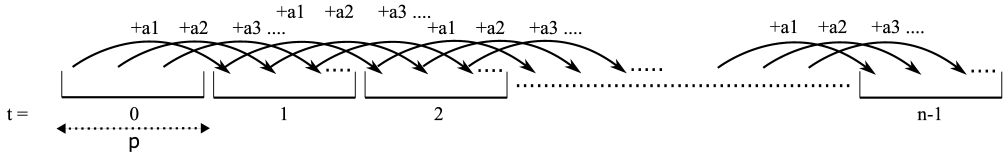


FIG. 3.1 – Intervalles adjacents de taille constante

(chapitre 2) : si la valeur du coefficient d'autocorrélation[1] à un ordre k donné (coefficient noté ρ_k) est élevée, alors il existe une relation linéaire forte entre tous les élements espacés de k.

On en déduit que l'ordonnancement par grandeur décroissante des coefficents d'autocorrélation détermine une suite optimale des ordres k (ou p) à considérer pour une interpolation périodique-linéaire.

Exemple 3.3.1 *Soit la suite* $[3, 3, 7, 13, 11, 23, 15, 33, 19, 43, 23]$.

L'ensemble des valeurs possibles de p *est* $\{1, 2, 3\}$. *On calcule donc les coefficients d'autocorrélation* $\rho_1 = 0.35$, $\rho_2 = 0.49$ *et* $\rho_3 = 0.02$. *Les périodes possibles, par ordre décroissant sont* $2, 1, 3$. *On en déduit qu'il sera plus probable de trouver une interpolation périodique-linéaire avec des intervalles de période* 2 *plutôt qu'avec une période de* 1 *ou encore une période de* 3.

Nous avons déjà vu que cette suite est effectivement interpolée par une fonction périodique de période 2, *donnée par* $[2, 5]x + [3, -2], x = 0, \ldots, 10$. *On peut encore l'écrire* $[4, 10]_y X + 3$, *avec* $X = 0, ..., 4$ *et* $y = 0, 1$, *ce qui correspond à une représentation multidimensionnelle de la suite.*

3.4 Représentation multidimensionnelle et configurations du modèle périodique-linéaire

3.4.1 Représentation multidimensionnelle

Lorsqu'on décompose une suite de valeurs en n intervalles disjoints successifs et linéairement dépendants, on associe à chaque intervalle un indice de temps t_1 ($t_1 = 0, \ldots, n-1$) et on définit un espace temps pour la modélisation. On rappelle que la fonction d'interpolation périodique-linéaire pour tous ces intervalles est de la forme $f(t_1) = at_1 + b$, où a et b sont des nombres périodiques.

Si la période de a et b est plus grande que 1, on peut s'intéresser aux séries temporelles définies par les éléments de a et b, et rechercher récursivement leurs

[1] pour être plus rigoureux, il faut considérer plutôt la valeur du coefficient de détermination ρ^2

fonctions d'interpolation périodiques-linéaires. Le coefficient a (respectivement b) sera donc représenté par une fonction périodique de la forme $f_a(t_2)$ (respectivement $f_b(t_2)$). Pour que a et b gardent toujours des périodes identiques, les coefficients de $f_a(t_2)$ et de $f_b(t_2)$ doivent être ramenés à la même période, comme nous l'avons montré au paragraphe 3.3.2.

Si la période de la fonction f_a (ou f_b) est plus grande que 1, on peut poursuivre la récursion sur ces 4 nouveaux coefficients, et ainsi de suite.

Au final, on obtient un espace temps multidimensionnel $(t_1, \ldots, t_h)$ et une arborescence binaire de fonctions périodiques (figure 3.2) pour la représentation de la suite de départ : à chaque niveau de l'arbre binaire des fonctions périodiques correspond une dimension temporelle t_i, $i = 1, \ldots, h$.

Un tel modèle multidimensionnel peut être entièrement représenté par un nid de boucles imbriquées de profondeur h, dont la forme générale est la suivante :

for $t_1 = 0$ to u_1
  for $t_2 = 0$ to u_2
    for $t_3 = 0$ to u_3
      ...
        for $t_h = 0$ to u_h
          $f(t_1, t_2, ..., t_h)$;

$f(t_1, t_2, ..., t_h)$ est la fonction périodique multi-variable finale résultante de l'application récursive de l'interpolation périodique-linéaire. C'est une fonction globalement non linéaire, mais cependant linéaire relativement à chaque variable t_i.

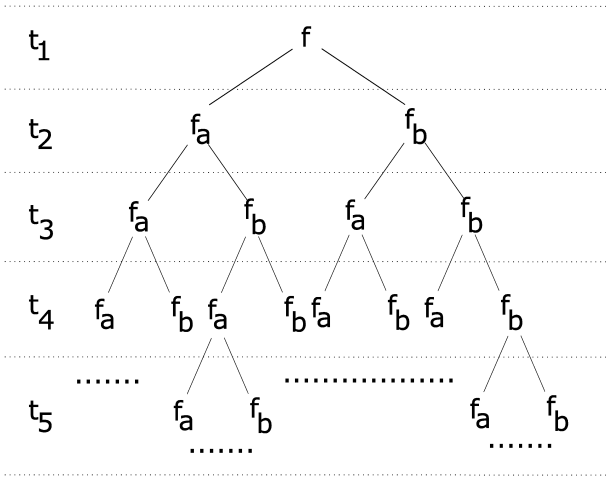


FIG. 3.2 – Arborescence de fonctions périodiques

Exemple 3.4.1 *Reconsidérons la suite* $[3, 3, 7, 13, 11, 23, 15, 33, 19, 43, 23, 53]$ *constituée de* 12 *éléments pouvant être regroupés en* 6 *intervalles comme suit :*

$$\underset{0\ 1}{[3,\ 3},\ \underset{0\ 1}{7,\ 13},\ \underset{0\ 1}{11,\ 23},\ \underset{0\ 1}{15,\ 33},\ \underset{0\ 1}{19,\ 43},\ \underset{0\ 1}{23,\ 53]}$$
$$\underset{0}{},\ \underset{1}{},\ \underset{2}{},\ \underset{3}{},\ \underset{4}{},\ \underset{5}{}$$

Suivant la première dimension de temps on trouve :

$$f(t_1) = [4, 10]t_1 + [3, 3], \quad 0 \leq t_1 \leq 5$$

Suivant la seconde dimension, pour $a = [4, 10]$ *et* $b = [3, 3]$*, on trouve*

$$\begin{cases} f_a(t_2) = 6t_2 + 4, & 0 \leq t_2 \leq 1 \\ \\ f_b(t_2) = 0t_2 + 3, & 0 \leq t_2 \leq 1 \end{cases}$$

Comme les coefficients périodiques sont tous de période 1*, le processus récursif s'arrête ici et on génère la boucle correspondante :*

for $t_1 = 0$ *to* 5
    for $t_2 = 0$ *to* 1
        $(6t_2 + 4)t_1 + 3$ *;*

Pour une suite de taille m quelconque la récursion termine, étant donné que la taille des intervalles décroît à chaque étape récursive jusqu'à ne plus trouver d'interpolation périodique-linéaire. Si la profondeur maximale h de l'arbre binaire est atteinte, il n'y a plus de coefficient périodique (c'est à dire de période >1), et le modèle a été appliqué autant de fois que possible. Toutefois, il est toujours possible de se limiter à une hauteur h_f fixée d'avance. C'est pourquoi les différentes dimensions de l'espace temps peuvent être perçues comme différents niveaux de granularité pour l'observation du comportement des programmes. De cette manière, on peut se restreindre à une analyse gros grain si l'on cherche juste à avoir une vue d'ensemble du programme, et dans le cas contraire, une analyse plus fine apportera davantage de précisions.

Pendant la recherche des intervalles linéairement dépendants, on peut considérer la durée comme facteur discriminant ou non, ce qui veut dire pour le premier cas que lorsqu'un comportement se reproduit, il doit le faire pendant la même longueur de temps que précédemment. Ces deux façons différentes de considérer la durée font aboutir à des configurations différentes des intervalles, et donc de notre modèle périodique-linéaire.

3.4.2 Configurations du modèle

Lorsqu'on décide que la durée est discriminante, on n'admet de relation qu'entre intervalles de même taille. Dans le cas contraire, on peut admettre que des intervalles de tailles différentes soient liés, c'est à dire qu'un comportement puisse se reproduire avec à chaque fois des durées de temps variables. On obtient ainsi deux configurations où les intervalles sont de taille constante ou variable. Pour chacune d'entre elles, il peut arriver que les intervalles soient séparés par d'autres intervalles relevant d'un comportement différent, si on est en présence de plusieurs comportements entrelacés. Ces combinaisons nous donnent au total 4 configurations possibles du modèle périodique-linéaire :

- intervalles adjacents de taille constante (figure 3.3),
- intervalles adjacents de taille variable (figure 3.4),
- intervalles distants de taille constante,
- intervalles distants de taille variable (figure 3.5).

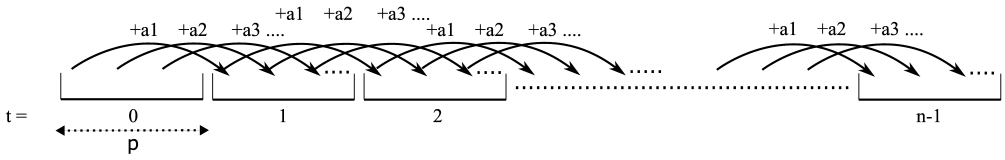


FIG. 3.3 – Intervalles adjacents de taille constante

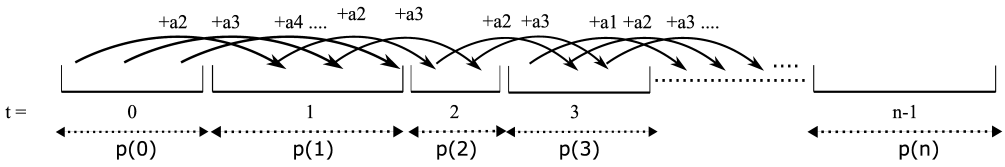


FIG. 3.4 – Intervalles adjacents de taille variable

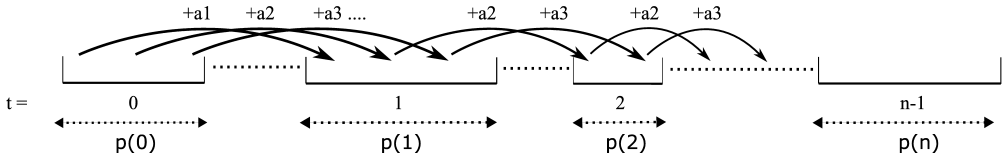


FIG. 3.5 – Intervalles distants de taille variable

Tenant compte de toutes ces différentes configurations, le modèle général de boucles associé aux différents modèles devient donc :

for $t_1 = 0$ to n
  for $t_2 = l(t_1)$ to $u(t_1)$
    for $t_3 = l(t_1, t_2)$ to $u(t_1, t_2)$
      ...
        for $t_h = l(t_1, t_2, ..., t_{h-1})$ to $u(t_1, t_2, ..., t_{h-1})$
          $f(t_1, t_2, ..., t_h)$;

où $f(t_1, t_2, ..., t_h)$ est toujours la fonction périodique multi-variable finale resultant de l'application récursive de l'interpolation périodique-linéaire, tandis que $l(t_1, t_2, ..., t_i)$ et $u(t_1, t_2, ..., t_i)$ sont des fonctions qui indiquent les tailles des intervalles considérés à l'étape $i + 1$.

Dans la suite, nous nous limitons particulièrement à des fonctions linéaires affines, afin de faciliter l'exploitation des résultats pour d'autres outils d'analyse tels que le modèle polyédrique.

Interpolation d'intervalles de taille variable

Lorsque la taille des intervalles à interpoler varie, il convient de modifier légèrement la définition de l'interpolation périodique linéaire donnée en 3.3.3. On pose i_{max} le plus grand des intervalles de taille variable interpolés, et max sa taille. Soit un intervalle quelconque i de taille s, dont les éléments sont e_{ij}, $0 \leq j \leq s-1$. Les intervalles i et i_{max} sont linéairement dépendants s'il existe $\alpha \in Z$ et s éléments consécutifs $e_{i_{max}k}$ dans i_{max}, $0 \leq k \leq max - 1$, tels que $e_{ij} = e_{i_{max}k} + \alpha a_k$.

La valeur α est associée de façon unique à l'intervalle i. Autrement dit, lorqsqu'un intervalle i est mis en relation de dépendance linéaire avec l'intervalle i_{max}, la différence entre leurs élements liés est égale à αa_k, et toute autre relation de dépendance incluant i_{max} implique une valeur de α différente.

Exemple 3.4.2 *On considère la trace suivante et on cherche une interpolation périodique-linéaire :* $[0, 5, 8, 10, 13, 16, 15, 18, 21, 24, 20, 23, 26, 29, 32]$. *Pour cette trace il existe une interpolation périodique-linéaire de* 5 *intervalles de taille variable, comme on le voit à la figure 3.6.*

Le plus grand intervalle est le dernier, et $max = 5$ *; la valeur de* α *associée au premier intervalle est* -4 *(*$0 = 20 - 4 * 5$*), la valeur associée au* 4^{ieme} *intervalle est* -1*, etc.*

En se basant sur les deux derniers intervalles, on trouve :

$$\begin{cases} 15 = 20 - 1 * 5 \rightarrow a_0 = 5 \\ 18 = 23 - 1 * 5 \rightarrow a_1 = 5 \\ 21 = 26 - 1 * 5 \rightarrow a_2 = 5 \\ 24 = 29 - 1 * 5 \rightarrow a_3 = 5 \end{cases}$$

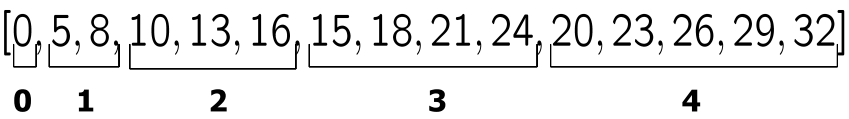


FIG. 3.6 – Intervalles adjacents de taille variable

N'ayant plus d'éléments pour le calcul de a_4 *on pose* $a_4 = 5$. *On vérifie ensuite que, si* s *est la taille d'un intervalle* i, *on a bien la relation :*

$$e_{ij} = e_{i_{max}j} + 5\alpha,\ \ 0 \leq i \leq 3,\ \ 0 \leq j \leq s-1.$$

Par extrapolation on a donc : $e_{0j} = e_{4j} - 20,\ \ 0 \leq j \leq 4,$
et on obtient la fonction d'interpolation périodique-linéaire suivante :

$$f(t_1) = 5t_1 + [0,3,6,9]_{t_2}, 0 \leq t_1 \leq 4$$

dans laquelle le deuxième coefficient périodique dépend de t_2 *à cause de la variabilité de la taille des intervalles.*
Sachant que $f_b(t_2) = [0,3,6,9] = 3t_2$, *la boucle finale s'écrit :*

for $t_1 = 0$ *to* 4
  for $t_2 = 0$ *to* t_1
    $5t_1 + 3t_2$;

Interpolation d'intervalles distants de taille constante

Le cas des intervalles distants diffère des intervalles adjacents par la présence de valeurs entrelacées qui relèvent d'un autre comportement que celui en cours d'identification. Prenons par exemple la trace suivante :

[15, 16, 17, 1, 6, 20, 23, 26, 9, 17, 25, 30, 35, 21, 12, 30, 37, 44, 0, 5, 35, 44, 53, 23, 25].

Tandis que la recherche d'une configuration d'intervalles ajacents de taille constante donnerait en résultat 4 phases, on trouve ici une seule phase d'intervalles de période 3, distants de 2 éléments à chaque fois. La boucle finale s'écrit :

for $t_1 = 0$ to 4
  for $t_2 = 0$ to 2
    $t_2 + 15 + (2t_2 + 5)t_1$;
  for $t_2 = 3$ to 4
    ;

Interpolation d'intervalles distants de taille variable

Le principe est le même que pour les intervalles adjacents de taille variable, à ceci près qu'on admet certains éléments de la trace pouvant représenter un tout autre comportement ; toutefois la taille des intervalles reste variable et ce de manière linéaire. On prend par exemple la trace suivante :

$$[0, 6, 120, 0, 12, 5, 240, 120, 0, 3, 7, 14, 360, 240, 120, 0, 5, 3, 11, 13, 480, 360, 240, 120].$$

On trouve des intervalles de taille variant de 1 à 5 et qui sont entrelacés avec d'autres intervalles de taille variable également, mais ne suivant pas la même relation. La boucle correspondante est donnée par :

$$\begin{array}{l} \text{for } t_1 = 0 \text{ to } 4 \\ \quad \text{for } t_2 = 0 \text{ to } t_1 \\ \qquad\qquad 120t_1 - 120t_2\,; \\ \quad \text{for } t_2 = t_1 + 1 \text{ to } 2t_1 + 1 \\ \qquad ; \end{array}$$

Remarque 3.4.1 *On constate bien que pour les configurations d'intervalles distants, le comportement entrelacé n'est pas modélisé. On peut cependant imaginer une extension permettant de le représenter, par exemple en générant une trace auxiliaire composée des éléments entremêlés, en y appliquant ensuite une passe de l'interpolation périodique-linéaire, pour enfin réunir les boucles obtenues pour chacune des deux analyses.*

3.4.3 Notion de phases

En général pour une trace donnée, une interpolation périodique-linéaire ne recouvre pas la totalité de la trace, et le résultat obtenu est alors constitué de plusieurs sous-ensembles appelés *phase*.

Définition 3.4.1 *Une phase est la plus grande partie d'une trace pour laquelle tous les éléments sont interpolés de manière périodique-linéaire.*

Sur le reste de la trace, il est possible d'appliquer à nouveau le modèle périodique-linéaire et d'identifier une autre phase, et ainsi de suite. On obtient donc plusieurs phases successives pour un niveau de granularité donné, à l'intérieur desquelles on peut identifier de nouvelles (plus petites) phases, définissant ainsi une hiérarchie de phases. Au niveau de la représentation, des phases successives correspondent à une succession de boucles de même niveau de profondeur, et chaque phase peut elle-même être composée de plusieurs sous-phases, ce qui définit bien une hiérarchie de phases représentée par une succession de boucles imbriquées.

Exemple 3.4.3 *Considérons la trace suivante pour laquelle on recherche une interpolation périodique-linéaire :* $[3, 4, 5, 1, 7, 14, 21, 8, 11, 24, 37, 15, 2, 5, 7, 9, 3, 5, 9, 11, 13, 4, 8, 13, 15, 17, 5, 11, 17, 19, 21, 6]$.

Au niveau de la première dimension de temps on obtient deux phases :

1. *une phase couvrant les* 12 *premiers éléments de la trace, de période* 4*, dans laquelle la fonction d'interpolation périodique-linéaire s'écrit :*

$$f(t_1) = [4, 10, 16, 7]t_1 + [3, 4, 5, 1], \quad 0 \leq t_1 \leq 2 \quad ;$$

2. *une deuxième phase couvrant tous les éléments restants, de période* 5*, dans laquelle la fonction d'interpolation périodique-linéaire s'écrit :*

$$f(t_1) = [3, 4, 4, 4, 1]t_1 + [2, 5, 7, 9, 3], \quad 0 \leq t_1 \leq 3.$$

A l'intérieur de la première phase, on interpole les coefficients périodiques et on identifie deux sous-phases de période 1*, composées de* 9 *et* 3 *éléments respectivement. La boucle correspondante s'écrit :*

for $t_1 = 0$ *to* 2
  for $t_2 = 0$ *to* 2
    $(6t_2 + 4)t_1 + t_2 + 3$ *;*
  for $t_2 = 3$ *to* 3
    $7t_1 + 1$;

A l'intérieur de la seconde phase, on interpole les coefficients périodiques et on identifie cette fois 3 *sous-phases de période* 1*, composées de* 4*,* 9 *et* 4 *éléments respectivement. La boucle correspondante est la suivante :*

for $t_1 = 0$ *to* 3
  for $t_2 = 0$ *to* 0
    $3t_1 + 2$ *;*
  for $t_2 = 1$ *to* 3
    $4t_1 + 2t_2 + 3$; $\quad /* 4t_1 + 2(t_2 - 1) + 5 */$
  for $t_2 = 4$ *to* 4
    $t_1 + 3$ *;*

En définitive on obtient la succession de boucles suivante :

for $t_1 = 0$ *to* 2
  for $t_2 = 0$ *to* 2
    $(6t_2 + 4)t_1 + t_2 + 3$ *;*
  for $t_2 = 3$ *to* 3
    $7t_1 + 1$;
for $t_1 = 3$ *to* 6
  for $t_2 = 0$ *to* 0

$$
\begin{array}{ll}
\quad 3t_1 - 7; & /*\, 3(t_1 - 3) + 2 \,*/ \\
\textit{for}\ t_2 = 1\ \textit{to}\ 3 & \\
\quad 4t_1 + 2t_2 - 9; & /*\, 4(t_1 - 3) + 2(t_2 - 1) + 5 \,*/ \\
\textit{for}\ t_2 = 4\ \textit{to}\ 4 & \\
\quad t_1; & /*\, (t_1 - 3) + 3 \,*/
\end{array}
$$

Nous faisons remarquer que la notion de *phase* telle que nous l'entendons ici est assez différente de celle que l'on peut retrouver dans d'autres travaux concernant l'analyse de programme et tout particulièrement les techniques d'identification et de détection de changement de phases [31, 30, 43, 75, 72, 74, 58]. Dans ces travaux, on définit une phase comme étant un ensemble de parties d'un programme, non nécessairement consécutives et caractérisées par la proximité ou l'identité de leurs représentants, au sens de la relation de similarité choisie. Intuitivement, une phase survient lorsque le comportement du programme est constant ou stable, et on dit qu'il y a un changement de phase lorsque ce comportement varie.

Cependant, comme le font remarquer Hind et al. [43], la notion de *phase* n'est pas définie de manière absolue et dépend fortement du choix des éléments atomiques qui la constituent, de leur représentation, ainsi que de la manière dont se calcule la similarité entre ces éléments.

Dans [75, 72, 74] par exemple, une phase est un ensemble d'intervalles dans lesquels le comportement du programme est similaire tout au long de son exécution, un intervalle étant défini comme une section d'exécution continue (ou encore une tranche de temps) du programme. Les intervalles ne se chevauchent pas, peuvent être de taille fixe ou de taille variable et ne sont pas forcément contigüs à l'intérieur d'une phase ; c'est ainsi qu'on peut avoir des phases qui se répètent au cours de l'exécution d'un programme.

On représente les intervalles par des vecteurs de blocs de base (BBV) qu'on pondère ensuite par leur fréquence d'exécution, et la similarité entre intervalles se mesure par des distances (euclidiennes, manhattan, etc) ; ainsi, deux intervalles faiblement distants sont reconnus comme faisant partie de la même phase. L'identification des phases est résolue comme un problème de classification (*clustering*) en apprentissage automatique, d'où l'utilisation des algorithmes de classification comme l'algorithme des k-moyennes.

Dans [31, 30], Dhodapkar et Smith mettent en relation la notion de phase avec celle d'*ensemble de travail* et montrent que les changements de phases surviennent au moment des changements de ces ensembles. Un *ensemble de travail* est un ensemble de segments distincts touchés dans une fenêtre (ou un intervalle) de taille donnée et un segment est une région mémoire (par exemple une page mémoire) de taille fixée ; ils définissent donc une phase comme l'intervalle maximal durant lequel l'*ensemble de travail* demeure plus ou moins constant. Ici tout comme dans les tra-

vaux cités précédemment, les intervalles ne se chevauchent pas, mais cependant les instructions comptabilisées ne sont pas pondérées par leur fréquence d'exécution.

Les changements de phases sont identifiés en comparant des *ensembles de travail* consécutifs à l'aide d'une métrique de similarité appelée *distance relative d'ensemble de travail*. Pour deux ensembles donnés, une distance relative nulle indique que ces ensembles sont identiques et une distance égale à l'unité indique qu'ils sont complètement différents. On définit alors une valeur seuil telle que toute distance supérieure implique un changement de phase et donc d'*ensemble de travail*.

En réponse à ce changement d'*ensemble de travail*, ils effectuent entre autres une reconfiguration dynamique des unités multiconfigurations (cache d'instruction, cache de données, prédicteur de branchement), et parviennent ainsi à réduire la consommation d'énergie.

3.5 Algorithmes pour le modèle périodique

Le modèle périodique-linéaire est caractérisé par :
- le nombre de niveaux de granularité ou encore la profondeur de l'arbre des fonctions ;
- le nombre de phases à chaque niveau ;
- la taille des intervalles, encore appelée période des fonctions d'interpolation.

La profondeur de l'arbre n'est pas connue d'avance, aussi peut-on envisager deux possibilités : soit on spécifie que l'analyse doit être la plus exhaustive possible (profondeur maximale), soit on entame une démarche itérative. Cette démarche itérative descendante va consister à appliquer le modèle à partir du plus haut niveau, étudier le résultat obtenu et déterminer s'il faut davantage de précision (aller au niveau suivant) ou non. Si l'on opte pour une démarche itérative, le nombre de phases peut être moins grand que dans le cas de l'exhaustivité.

Dans les deux situations, par souci de cohésion avec la représentation sous forme de boucles, on précise qu'une phase doit avoir un nombre entier d'intervalles. Ainsi, tout intervalle *incomplet*, i.e. de taille inférieure aux autres, même s'il est correctement interpolé, ne sera pas inclus dans une phase, bien que le nombre de phases doive être minimal.

En ce qui concerne la taille des intervalles, on sait déjà qu'elle vaut au maximum $n/3$, n étant la taille de la trace. Cependant, si on ne trouve pas de phase d'au moins 3 points interpolés, on pourra lever cette restriction.

Supposons que pour deux valeurs p et p' l'on identifie une même phase d'intervalles, et que l'on ait $p' > p$. Comme p représente plus de points interpolés que p' et que le nombre périodique associé est moins large que celui associé à p', on retiendra qu'à cette phase correspond des intervalles de période p.

De manière générale, les algorithmes procèdent par une stratégie descendante comme suit :

1. trouver une fonction d'interpolation périodique-linéaire, d'au moins trois intervalles, et qui représente la plus grande partie de la trace. Définir cette plus grande partie comme une phase ;
2. sur la ou les parties restantes de la trace, appliquer l'étape 1 pour définir d'autres phases ;
3. un niveau de la hiérarchie est effectué. Au besoin, définir le niveau hiérarchique suivant pour chacune des phases identifiées précédemment. On applique pour cela les étapes 1, 2, 3 aux coefficients périodiques a et b de chaque phase.

3.5.1 Intervalles adjacents de taille constante

L'ensemble des valeurs possibles de p (taille des intervalles) est connu ; on se sert des coefficients d'autoccorélation pour les ordonner et déterminer un ordre de précédence. Pour une période donnée p, la recherche démarre du début de la trace (indice 0) en considérant tout d'abord les $3p$ premiers éléments. Si une fonction d'interpolation périodique est trouvée, on a identifié une première phase. Sinon, on s'intéresse aux $3p$ éléments partant de l'indice p (de 0 à $4p-1$), et ainsi de suite. A chaque fois qu'on identifie une première phase, on vérifie que la fonction d'interpolation s'applique aussi aux p éléments suivants de manière à accroître la taille de cette phase, et on poursuit l'extension autant que possible. L'algorithme correspondant à cette configuration est l'**Algorithme 1** et procède comme suit :

- Initialement (ligne 1) on n'a pas encore trouvé de phase et $phase_{max}$ est mise à $NULL$.

- Les lignes 2 et 3 permettent de calculer les coefficients d'autocorrélation jusqu'à l'ordre $n/3$.

- Les lignes 4 à 14 quant à elles concernent la recherche de la phase maximale proprement dite, pour toutes les périodes possibles.

- Le test effectué en ligne 12 porte sur la dernière phase trouvée, afin de déterminer si elle est la plus grande ou si, ayant la même taille que la phase maximale identifiée jusqu'alors, elle est composée d'intervalles de plus petite période ; dans l'un de ces deux cas, la phase courante devient alors la nouvelle phase maximale.

- Les lignes 16 à 17 servent à trouver la dernière phase à partir de quelques éléments restant : si on n'a pas trouvé de phases avec au moins 3 intervalles, on lève cette restriction en permettant le choix d'une phase de seulement

Algorithm 1 *trouve_phase*(T)

ENTRÉES: une trace T de taille n

1: $phase_{max} = NULL$
2: **pour tout** p tel que $1 \leq p \leq n/3$ **faire**
3: calculer les coefficients d'autocorrélation r_p d'ordre p
4: **pour** le plus grand au plus petit coefficient r_p et les périodes associées p, $r_p \geq 0.1$ **faire**
5: **pour** i=1 à p **faire**
6: trouver la plus petite valeur entière $\alpha \leq \frac{n-3p}{p}$ telle que 3 éléments au moins $T[i+\alpha p]$, $T[i+(\alpha+1)p]$ et $T[i+(\alpha+2)p]$ soient liés
7: **tantque** une valeur α a été trouvée **faire**
8: **pour tout** q tel que $1 \leq q \leq p-1$ **faire**
9: vérifier que $T[i+\alpha p+q]$, $T[i+(\alpha+1)p+q]$ et $T[i+(\alpha+2)p+q]$ sont aussi linéairement dépendants
10: **si** c'est le cas **alors**
11: élargir cette séquence d'intervalles vers la droite, au maximum possible
12: **si** $(taille(phase_{courante}) > taille(phase_{max}))$ OU $(taille(phase_{courante}) = taille(phase_{max})$ ET $p_{courante} < p_{max})$ **alors**
13: $phase_{max} = phase_{courante}$; $p_{max} = p_{courante}$
14: trouver la valeur suivante de $\alpha \leq \frac{n-3p}{p}$
15:
16: **si** $phase_{max} = NULL$ **alors**
17: permettre de choisir des phases de moins de 3 élements
18: construire la fonction d'interpolation périodique-linéaire $f_{phase_{max}}$ de coefficients a et b et de période p_{max}
19: soit T_{gauche} la partie gauche de $T - phase_{max}$, et T_{droite} la partie droite de $T - phase_{max}$
20: trouve_phase(T_{gauche}) ; trouve_phase(T_{droite})

deux intervalles.

- L'algorithme se poursuit par deux appels récursifs sur les parties disjointes de la phase identifiée précédemment.

3.5.2 Intervalles adjacents de taille variable

Afin de représenter le résultat final sous forme de nids de boucles à bornes affines sur les indices, on impose que les tailles des intervalles soient linéairement dépendantes. De plus, on suppose que si deux intervalles sont liés, leurs premiers éléments le sont aussi, ce qui permet de réduire la complexité de l'algorithme de recherche. Les intervalles étant de taille variable, la période sera celle de l'intervalle le plus grand. On ne peut donc plus se servir de l'autocorrélation pour guider la recherche.

La recherche débute sur les premiers éléments de la trace, à partir de l'indice $i = 0$. On pose d la taille du premier intervalle, q la taille du deuxième intervalle, et on cherche une dépendance linéaire entre l'intervalle suivant de taille $2q - d$ et les 2 intervalles précédents. Si une telle relation existe, on a trouvé une phase et on essaie de l'étendre autant que possible. On recommence le processus avec différentes valeurs de i, d, q, en comparant à chaque fois la phase précédente à la phase courante pour conserver toujours la plus grande des deux.

A cause de la relation linéaire entre les tailles des intervalles consécutifs, la plus petite phase peut avoir au minimum 6 éléments et est obtenue lorsque $d = 1$, $q = 2$, $2q - d = 3$ ou encore $d = 3$, $q = 2$, $2q - d = 1$. Sachant que pour une valeur de i donnée on ne traite que $s = n - i$ éléments de la trace, et que s varie de n à 6, on en déduit que i varie de 0 à $n - 6$.

Pour d, la valeur minimale est 1 et la valeur maximale est atteinte lorsque $d > q$ et $2q - d = 1$, ce qui entraîne le système suivant :

$$\begin{cases} 2q - 1 &= d \\ d + q &= s - 1 \end{cases} \quad \Longrightarrow \quad \begin{cases} q &= s/3 \\ d &= 2s/3 - 1 \end{cases}$$

Quant aux valeurs de q, ont doit toujours avoir $2q - d \geq 1$, et la valeur maximale est atteinte avec $s - d - 1$.

L'algorithme correspondant est l'**Algorithme 2**.

Algorithm 2 *trouve_phase_var*(T)

ENTRÉES: une trace T de taille n

1: $phase_{max} = NULL$
2: **pour** $i = 0$ à $n - 6$ **faire**
3: **pour** $d = 1$ à $(2n - 2i - 3)/3$ **faire**
4: **pour** $q = d/2 + 1$ à $n - i - d - 1$ {On cherche une valeur de q telle que $T[i + d + q]$, $T[i + d]$ et $T[i]$ soient liés} **faire**
5: **si** $T[i + d + q] = (T[i + d] - T[i]) + T[i + d]$ {si $q - d > 0$ les tailles des intervalles croissent et le plus petit des intervalles est le premier} **alors**
6: **pour tout** k tel que $1 \leq k \leq min(d - 1, q + (q - d) - 1)$ {on veut vérifier que tous les élements du plus petit intervalle sont interpolés avec les éléments correspondant des deux autres} **faire**
7: vérifier que $T[i+k]$, $T[i+d+k]$ et $T[i+d+q+k]$ sont linéairement dépendants
8: **si** c'est le cas **alors**
9: élargir cette séquence d'intervalles de tailles $q+2(q-d)$, $q+3(q-d)$, $\ldots q + \alpha_{max}(q - d)$ vers la droite, au maximum possible, en vérifiant que tous les éléments d'un intervalle donné soient liés aux élements correpondants de l'intervalle suivant si $(q - d) > 0$ ou de l'intervalle précédent si $(q - d) < 0$
10: **si** $(q - d) < 0$ **alors**
11: $P_{courante} = d$;
12: **sinon**
13: $P_{courante} = q + \alpha_{max}(q - d)$;
14: **si** taille$(phase_{courante})$ > taille$(phase_{max})$ OU (taille$(phase_{courante})$ = taille$(phase_{max})$ ET $p_{courante} < p_{max}$) **alors**
15: $phase_{max} = phase_{courante}$;
16: $p_{max} = p_{courante}$
17: **si** $n \geq 6$ ET $phase_{max} = NULL$ {on n'a pas trouvé de phases} **alors**
18: sortir avec un message " pas d'interpolation pour cette configuration "
19: **sinon**
20: **si** $n < 6$ {on ne peut avoir des tailles variables} **alors**
21: effectuer une interpolation avec la configuration "intervalles adjacents de taille fixe"
22: **sinon**
23: construire la fonction d'interpolation périodique-linéaire $f_{phase_{max}}$ de coefficients a et b et de période p_{max}
24: soit T_{gauche} la partie gauche de $T - phase_{max}$, et T_{droite} la partie droite de $T - phase_{max}$
25: trouve_phase_var(T_{gauche}) ; trouve_phase_var(T_{droite})

3.5.3 Intervalles distants de taille constante

La recherche d'intervalles distants de taille constante est semblable à celle décrite en 3.5.1, à la seule différence qu'il peut exister un ou plusieurs phénomènes alternant avec le phénomème identifié, et dont les valeurs sont intercalées dans la sous-suite interpolée. On modifie donc l'**algorithme 1** pour tenir compte de la distance d' entre les intervalles, et on recherche des intervalles de période $p + d'$; toutefois lors de l'interpolation, l'on ne tient aucun compte des d' derniers éléments des différents intervalles.

Pour cette configuration, les coefficients d'autocorrélation permettent encore d'ordonner les valeurs de p, tandis qu'on fait varier i de 0 à $n - 3p$ et d' de 0 à $(n - i - 3 * p)/2$.

3.5.4 Intervalles distants de taille variable

L'algorithme pour cette configuration s'apparente à celui décrit en 3.5.2 et tient compte de la distance éventuelle entre les intervalles. On impose toujours qu'il existe une relation linéaire entre les durées des répétitions (tailles des intervalles), et de nouveau, les coefficients d'autocorrélation ne sont d'aucune utilité pour ordonner les périodes, à cause de la variabilité des tailles des intervalles.

3.5.5 Complexité des algorithmes

On pose $C(n)$ la complexité de l'algorithme pour une trace de longueur n. On appellera c_i le temps de calcul d'une ligne i de l'algorithme, et coût(i) le coût associé. $C(n)$ est donnée par :

$$C(n) = \sum_i c_i * \text{coût}(i), \quad \forall i$$

Cas des intervalles adjacents de taille constante

Soit c_3 le temps de calcul d'un coefficient d'autocorrélation. Etant donné que toutes les opérations effectuées pour ce calcul sont élémentaires, on peut dire que c_3 est constant ;

- le coût des lignes 2 et 3 étant de $n/3$, la complexité à ces endroits est égale à $n/3 * c_2 + n/3 * c_3 = n/3 * (c_2 + c_3)$
- à la ligne 4 on effectue un tri, d'où une complexité de $(n/3)log(n/3)$ tandis qu'à la ligne 5 on a un coût de $\sum_{i=1}^{p} i = \frac{1}{2}p(1 + p)$ avec $p = n/3$
- le temps de calcul $c6$ est borné par $\frac{n-3p}{p} + 1$, avec $p = 1$ et le coût associé est de $\sum_{i=1}^{p} i = \frac{1}{2}p(1 + p)$

- la boucle en 7 est effectuée entre $n-2$ fois au maximum ($p=1$) et 1 fois (lorsque $p=n/3$) selon les valeurs de p et son coût est donc égal à
  $(n-2)+2(\frac{(n-6)}{2}+1)+3(\frac{(n-9)}{3}+1)+\cdots+\frac{n}{3}$
  $=(n-2)+(n-4)+(n-6)+\cdots+(n-\frac{2n}{3})$
  $=n\frac{n}{3}-\frac{n}{3}(\frac{n}{3}+1)=\frac{n(2n-3)}{9}$

- à la ligne 8 tout comme en 9, on a $p-1$ vérifications à faire $\forall p$, et le côut est égal à
  $=(n-2)+(n-4)+2(n-6)+3(n-8)+\cdots+(\frac{n}{3}-1)(n-\frac{2n}{3})$
  $=(n-2)+n[1+2+3+\cdots+(\frac{n}{3}-1)]-(4+2*6+3*8+\cdots+\frac{2n^2}{9}-\frac{2n}{3})$,
  ce qui est majoré par $\frac{n^2(n-3)}{18}$

- en ligne 11 le nombre d'élargissements dépend de la valeur de p et de α, on le notera $el_{ij}, 0\leq i=p\leq n/3 \ et \ 0\leq j=\alpha\leq\frac{n-3p}{p}$. On a par exemple

$$el_{10}=n-3,\ el_{11}=n-4,\ \ldots,\ el_{1(n-3)}=0$$

$$el_{20}=\frac{n-6}{2},\ el_{21}=\frac{n-6}{2}-1,\ \ldots,\ el_{1\frac{n-6}{2}}=0$$

  Lorsqu'on fixe p et qu'on pose $\frac{n-3p}{p}=A$ on a $\sum_{j=0}^{A} el_{pj}=A+(A-1)+(A-2)+\cdots+(A-A)=A(A+1)-\sum_{i=1}^{A} i$
  $=\frac{A(A+1)}{2}$ et le coût total associé à cette ligne est de $\sum_{p=1}^{n/3} p\frac{A(A+1)}{2}$ ce qui est majoré par n^3.

Au final on a :

$$C(n)=\sum_{1}^{13} c_i * \text{coût}(i)+C(n')+C(n'')$$

où n' est la taille de la trace T_{gauche} et n'' celle de la trace T_{droite}.

Comme la somme $\sum_1^{13} c_i * \text{coût}(i)$ avec une trace de taille n est majorée par n^3 et que $n', n'' < n$, on en déduit que

$$C(n)\in\mathcal{O}(n^3)$$

Cas des intervalles adjacents de taille variable

les c_i sont constants

- le coût de la ligne 2 est de $n-5$ et celui de la ligne 3 égal à
  $\sum_{i=0}^{n-6}\frac{2n-2i-3}{3}=\frac{1}{3}((2n-3)+(2n-5)+(2n-7)+\cdots+(2n-(2n-12)-3))$
  $=\frac{1}{3}((n-5)2n-(3+5+7+\cdots+(2n-9)))=\frac{1}{3}((n-5)2n-((n-4)^2-1))$
  $=\frac{(n^2-2n-15)}{3}$

- à la ligne 4 les variations de q sont majorées par n d'où un coût de l'ordre de $= \frac{n(n^2-2n-15)}{3}$
- à la ligne 6 l'indice k est majoré par $d-1$ de sorte que, pour $i=0$ par exemple, on a un coût de $n + \sum_{d=2}^{\frac{2n-3}{3}} (d-1)n$, et le coût total est donc de $\sum_{i=0}^{n-6}(\sum_{d=1}^{\frac{2n-2i-3}{3}} dn)$, ce qui est de l'ordre de n^4
- la vérification en 7 se fait en temps constant et l'élargissement dans cette configuration n'est pas aussi fréquent que pour les intervalles de taille variable ; il n'en demeure pas moins qu'on peut en majorer le nombre par n et le coût pour cette ligne devient de l'ordre de n^5, quoique négligeable devant n^5.

Au final on a :

$$C(n) \in o(n^5)$$

3.5.6 Discussion

Pour nos algorithmes, nous avons opté pour une approche descendante. Celle-ci a l'avantage de permettre le contrôle du niveau de granularité de la modélisation périodique-linéaire. L'inconvénient majeur de cette approche est de nécessiter plusieurs parcours de la trace d'entrée afin d'identifier la période. Cela entraîne pour les traces de grandes tailles un temps de traitement important.

Une approche ascendante aurait permis d'éviter ce problème. En effet, un tel algorithme aurait tout d'abord interpolé linéairement, de manière classique non périodique, les éléments de la trace, en découpant cette trace en phases successives d'éléments interpolés. Cette succession d'interpolations linéaires aurait constitué une première compression de la trace d'entrée, réduisant ainsi sensiblement le volume de données à traiter ensuite. L'étape suivante aurait consisté à rechercher à nouveau une interpolation linéaire entre les interpolations construites précédemment, et il en aurait été de même pour toutes les autres étapes suivantes, jusqu'au modèle complet où plus aucune interpolation n'est possible.

Bien que plus efficace en temps, cette approche pose de nombreux problèmes quant à l'identification des périodes et des phases. Pour cela, plusieurs parcours seraient malgré tout nécessaires. De plus, les fonctions périodiques-linéaires d'interpolation ne pourraient être identifiées qu'à postériori, à la fin d'une modélisation complète.

3.6 Extensions du modèle

3.6.1 Modèle approximatif

Les fonctions périodiques-linéaires, tel que nous les avons présentées jusqu'ici, interpolent exactement la trace considérée. Cependant, dans certains cas, l'exactitude peut être un critère trop contraignant voire inutile. Prenons par exemple le cas où la décomposition en phases d'intervalles conduit à une multitude de petites phases de faibles tailles, à cause de la présence de quelques valeurs singulières. Il est possible qu'en ignorant ces valeurs disparates, l'on aboutisse à un nombre considérablement réduit de phases. C'est pourquoi on peut opportunément admettre un pourcentage de valeurs erronnées lors de l'interpolation, lesquelles valeurs sont alors considérées comme résultant d'événements exceptionnels différant du comportement général. Nous nous référons alors à l'objectif classique d'optimisation logicielle et matérielle : couvrir les cas les plus fréquents.

De manière classique, une interpolation approximative peut être effectuée grâce à des techniques de regression linéaire, et les valeurs calculées sont alors des approximations des termes de la trace. Pour conserver un modèle cohérent et plus adapté à la compréhension du comportement du programme, nous choisissons une approche qui garantisse que l'approximation de la fonction d'interpolation soit égale à la fonction exacte pour la plus grande partie des valeurs de la trace. Il s'agit donc de définir un pourcentage maximum d'éléments issus d'un comportement différent de celui qu'on observe. De la même façon, on peut admettre des intervalles dont la taille diffère de ce qui est attendu, dans les configurations *intervalles de taille constante* ou *intervalles de taille variable*.

Un autre contexte favorable à l'utilisation d'une interpolation approximative est celui de la prédiction dynamique. En effet, l'utilisation d'un modèle exact et complexe serait alors susceptible d'augmenter considérablement le coût de la prédiction. Pour obtenir le modèle approximatif dans ce cas, on peut partir de l'arbre binaire des phases et des fonctions d'interpolation correspondant au modèle exact, sur lequel on applique une technique d'élagage (figure 3.6.1).

3.6.2 Modèle paramétrique

Lorsque les tailles des intervalles et des phases dépendent linéairement de certains paramètres, le modèle est lui aussi paramétré. Cela permet notamment de reconnaître un comportement identique, quand il se présente, à travers l'analyse de traces différentes, obtenues avec des données d'entrée différentes du programme ou générées sur des plate-formes différentes. Pour construire ce modèle paramétrique, on a le choix entre :

- calculer les fonctions d'interpolation pour chacune des traces et les comparer ensuite pour en déduire les paramètres généraux ;

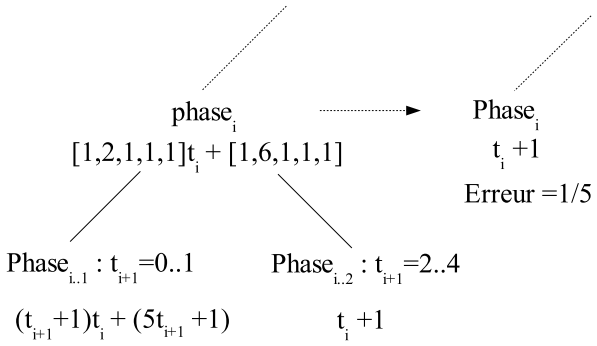


FIG. 3.7 – Elagage de l'arbre des phases et fonctions d'interpolation

– calculer la fonction d'interpolation pour une trace donnée et vérifier son adéquation avec les autres traces, en s'aidant d'éventuelles relations linéaires entre les caractéristiques des traces (taille des phases et des intervalles, valeurs des éléments, ...)

En guise d'illustration, nous présentons en figure 3.8 des boucles paramétrées obtenues à l'issue de la modélisation de la trace des accès mémoires via 3 pointeurs, pour le programme `fir2dim` qui implémente le filtre d'une image $2D$ par une matrice de convolution 3×3, et qui provient des banc d'essai DSPStone [85]. Ce programme comporte un paramètre d'entrée à savoir la dimension de l'image qu'il traite. La taille du tableau contenant cette image est un autre paramètre d'exécution.

for $t_1 = 0$ to $IMAGEDIM - 1$
  for $t_2 = 0$ to $IMAGEDIM - 1$
    for $t_3 = 0$ to 2
      for $t_4 = 0$ to 2
        $ARRAYDIM * t_1 + t_2 + ARRAYDIM * t_3 + t_4$;

FIG. 3.8 – Boucles paramétrées modélisant une trace du programme `fir2dim`

3.6.3 RPLI : Récurrences périodiques-linéaires

Le modèle périodique-linéaire, en identifiant des dépendances linéaires entre les éléments d'une trace, permet de s'affranchir de la trop grande complexité d'une

interpolation polynomiale et offre de surcroît une compréhension plus accrue du comportement du programme étudié. Cependant, il existe d'autres formes de dépendances permettant d'expliciter tout aussi clairement le comportement des programmes et ayant l'avantage de couvrir un plus large spectre de données, à l'instar des récurrences linéaires.

Exemple 3.6.1 *Soit la trace de valeurs suivante à analyser :*
$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]$.
La recherche d'une interpolation périodique-linéaire fournit en résultat 3 *phases :*

1. *une première phase allant de* 0 *à* 1 *avec une fonction d'interpolation donnée par* $f_0 = x,\ x = 0..1$
2. *une deuxième phase allant de* 2 *à* 4 *avec comme fonction d'interpolation* $f_1 = x + 1,\ x = 0..2$
3. *une dernière phase portant sur le reste de la trace, dans laquelle la restriction sur le nombre minimum d'intervalles interpolés (*3*) a été levée, ce qui signifie qu'il n'y a pas eu d'interpolation périodique-linéaire significative dans cette phase.*

Par contre, l'examen de cette même trace à l'aide des récurrences linéaires montre que chaque terme est la somme des deux termes qui le précèdent, i.e.

$$u_n = u_{n-1} + u_{n-2} \quad \forall n \geq 2,\ u_0 = 0, u_1 = 1.$$

L'extension *RPLI* a donc pour objectif de déceler des dépendances sous forme de récurrences linéaires entre les éléments d'une trace, mettant ainsi en évidence une corrélation entre un nombre fixe d'éléments.

Définition 3.6.1 *Une équation de récurrence se présente généralement sous la forme*

$$u_n = f(\{u_{n-1}, u_{n-2}, \ldots, u_{n-k}\}), \quad k, n \in \mathbb{N}$$

avec k *conditions initiales permettant de démarrer la récurrence et les* u_i *étant les valeurs d'une suite.*

Nous nous intéressons au cas où f est une combinaison linéaire à coefficients constants, et ne dépend pas seulement des u_i ; plus précisément,
$u_n = f(\{u_{n-1}, u_{n-2}, \ldots, u_{n-k}\}) + c, \quad k, n \in \mathbb{N},\ c$ est une constante.
On parle alors d'équation de récurrence linéaire non homogène d'ordre k à coefficients contants.

Puisqu'on a au total $k+1$ inconnues à rechercher, on résoud un système de $k+1$ équations de récurrence, constituées à partir des éléments de la trace comme suit :

$$\begin{cases} u_k & = & a_1u_{k-1} & + & a_2u_{k-2} & + & \cdots & + & a_{k-1}u_1 & + & a_ku_0 & + & a_{k+1} \\ u_k+1 & = & a_1u_k & + & a_2u_{k-1} & + & \cdots & + & a_{k-1}u_2 & + & a_ku_1 & + & a_{k+1} \\ \vdots & & & & & & & & & & & & \\ u_{2k} & = & a_1u_{2k-1} & + & a_2u_{2k-2} & + & \cdots & + & a_{k-1}u_{k+1} & + & a_ku_k & + & a_{k+1} \end{cases}$$

On maintient la cohérence du modèle périodique en recherchant les récurrences suivant une période p, et le système à résoudre est donc le suivant :

$$\begin{cases} u_k & = & a_1u_{k-p} & + & a_2u_{k-2p} & + & \cdots & + & a_ku_{k-kp} & + & a_{k+1} \\ u_k+p & = & a_1u_k & + & a_2u_{k-p} & + & \cdots & + & a_ku_{k-(k-1)p} & + & a_{k+1} \\ \vdots & & & & & & & & & & \\ u_{k+kp} & = & a_1u_{k+(k-1)p} & + & a_2u_{k+(k-2)p} & + & \cdots & + & a_ku_k & + & a_{k+1} \end{cases}$$

On définit ainsi un modèle de récurrence linéaire périodique :

$$u_n = [a_{11}, \cdots, a_{1p}]u_{n-p} + \cdots + [a_{k1}, \cdots, a_{kp}]u_{n-kp} + [a_{k+1,1}, \cdots, a_{k+1,p}]$$

Algorithmes

Le principe des algorithmes qui implémentent cette extension est sensiblement le même que celui des algorithmes pour PLI, avec ceci de particulier que $2k+1$ points au minimum sont nécessaires pour former un système d'ordre k (de u_0 à u_{2k} pour le cas simple où $p=1$).

Pour éviter les solutions irrationnelles, on introduit une nouvelle variable g qui est égale au *ppcm* des dénominateurs de toutes les solutions et on obtient un système à $k+2$ inconnues : il faut donc $k+2$ équations pour résoudre ce système et on impose alors que $n \geq 2k+2$, n étant la taille de la trace.

On démarre l'algorithme en recherchant les $2k+2$ premiers points distants de p pour lesquels le système d'équations récurrentes admette au moins une solution non nulle ; si une telle solution est trouvée, on vérifie que les systèmes correspondants aux autres points de l'intervalle ont également des solutions non nulles et on étend la phase identifiée au maximum possible vers la fin de la trace.

Exemple 3.6.2 *Pour la trace* $[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]$, *on cherche une récurrence linéaire d'ordre* $k=2$. *On commence donc par constituer, à l'aide des* 6 *premiers points de la trace* ($p=1$), *un système de* 4 *équations à* 4 *inconnues et on obtient le système suivant :*

$$\begin{cases} g & = & a_1+ & 0a_2+ & c \\ 2g & = & a_1+ & a_2+ & c \\ 3g & = & 2a_1+ & a_2+ & c \\ 5g & = & 3a_1+ & 2a_2+ & c \end{cases}$$

qui se résoud en

$$c = 0, a_1 = a_2 = p.$$

On pose $p = 1$ et on étend la phase identifiée à toute la trace, car tous les termes suivants vérifient bien la relation

$$u_n = u_{n-1} + u_{n-2}.$$

Les résultats finaux obtenus par l'algorithme ne conservent pas explicitement la variable ajoutée g. Puisque que le modèle de récurrences linéaires reste périodique, ces résultats consistent en la donnée de deux matrices : une matrice de coefficients A de taille $p \times (k+1)$ et une matrice de termes initiaux U de taille $(k+1) \times p$. Une ligne i de A $(1 \leq i \leq p)$ correspond aux coefficients de l'équation récurrente pour des éléments espacés de p dans la trace, et les valeurs initiales de ces mêmes éléments sont stockés en colonne i de la matrice U (les k premières lignes seulement car le dernier élément vaut toujours 1, coefficient de c). De cette manière, un élément de la trace est identifié par rapport à p et se note $u(t_1, t_2),\ 0 \leq t_1 < n/p,\ 1 \leq t_2 \leq p$.

On a en plus l'expression d'une boucle permettant de calculer tous les autres termes de la trace, selon les équations de récurrence identifiées. Pour une phase identifiée de taille n et de période p, le résultat se met sous forme de boucles comme suit :

for $t_1 = k$ to $n/p - 1$
  for $t_2 = 1$ to p
    $u(t_1, t_2) = \sum_{j=0}^{k-1} [A(t_2, (j+1)) * u((t_1 - j - 1), t_2)] + A(t_2, k+1);$

Exemple 3.6.3 *Pour la trace* $[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]$ *mentionnée précédemment pour laquelle la récurrence d'ordre* 2 *aboutit avec une période de* 1, *on a les les matrices* $A = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix}$ *, et* $U = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix}^T$ *. La boucle correspondante est suivante :*

for $t_1 = 2$ *to* 14
  for $t_2 = 1$ *to* 1
    $u(t_1, t_2) = \sum_{j=0}^{1} [A(t_2, (j+1)) * u((t_1 - j - 1), t_2)] + A(t_2, 3);$

avec $u(1, 1) = 1$ *et* $u(0, 1) = 0$.

Exemple 3.6.4 *Prenons comme autre exemple la trace :*
$[0, 2, 2, 1, 3, 3, 14, 8, 51, 16, 206, 27, 819, 41, 3278, 58, 13107, 78, 52430, 101]$.

Avec une récurrence d'ordre 2, on trouve les résultats suivants :

$p = 2, \quad A = \begin{pmatrix} 3 & 4 & -3 \\ 2 & -1 & 3 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 1 \\ 0 & 2 \\ 1 & 1 \end{pmatrix};$

la boucle correspondante est :

for $t_1 = 2$ *to* 9
  for $t_2 = 1$ *to* 2
    $u(t1, t2) = \sum_{j=0}^{1} [A(t2, (j+1)) * u((t1 - j - 1), t2)] + A(t2, 3);$

avec $u(1,1) = 2$, $u(0,1) = 0$, $u(1,2) = 1$ *et* $u(0,2) = 2$.

Complexité

Les algorithmes pour $RPLI$ sont plus complexes que ceux qui sont implémentés pour PLI, car ils font plusieurs appels à des routines de la bibliothèque *Polylib* [6] pour la décomposition LU de matrices et la résolution de systèmes homogènes qui en découlent ; en outre, il est nécessaire de stocker au moins $2p(k+1)$ éléments pour aboutir au résultat final, d'où une plus grande complexité en espace également. C'est pourquoi les solutions obtenues ne constituent pas toujours une modélisation exploitable du comportement du programme.

Cependant, on peut aussi poser une restriction visant à rejeter les solutions nécessitant un fort pourcentage de stockage d'éléments de la trace : par exemple si $2p(k+1) > n/3$, n étant la taille de la trace, on rejette le résultat et on abandonne le modèle $RPLI$ pour la trace en question.

Pour le modèle de récurrences périodiques-linéaires on peut envisager, comme pour PLI, des versions étendues, comme par exemple un modèle d'intervalles récurrents distants de taille constante, etc.

Dans tous les cas, la recherche d'une interpolation périodique-linéaire produisant une hiérarchie de phases, chaque phase et chaque sous-phase doivent être analysées à leur tour pour pouvoir aboutir au modèle final de boucles. Cette application récursive du modèle peut s'avérer longue et fastidieuse si elle se fait manuellement, d'où l'idée de produire directement, à partir de la trace initiale, la résultante des différentes boucles obtenues à tous les niveaux possibles de la hiérarchie.

3.7 Génération automatique des boucles

Lors de la construction du modèle périodique-linéaire, nous avons vu que plusieurs phases sont successivement puis récursivement extraites de la trace, partant

de la plus grande possible. Chacune de ces phases peut être représentée de façon multidimensionnelle par un nid de boucles comme suit :

for $t_1 = 0$ to n
  for $t_2 = l(t_1)$ to $u(t_1)$
    for $t_3 = l(t_1, t_2)$ to $u(t_1, t_2)$
      ...
        for $t_h = l(t_1, t_2, ..., t_{h-1})$ to $u(t_1, t_2, ..., t_{h-1})$
          $f(t_1, t_2, ..., t_h)$;

où

- $f(t_1, t_2, ..., t_h)$ est la fonction périodique multi-variable finale
- $l(t_1, t_2, ..., t_i)$ et $u(t_1, t_2, ..., t_i)$ sont des fonctions affines qui indiquent les tailles des intervalles considérés à l'étape $i + 1$.

Chaque étape de représentation résulte de l'application récursive de l'interpolation périodique-linéaire aux éléments de la phase concernée. Bien qu'il soit possible d'itérer manuellement le développement des différentes étapes jusqu'à un niveau fixé, il est clair qu'une automatisation de ce processus soit plus judicieuse encore, au regard du nombre total de phases et d'étapes qui peut être très élevé. Alors se pose une question fondamentale à résoudre, en plus de l'intégration dans la boucle finale des phases successives issues des niveaux hiérarchiques, à savoir :

comment mettre en correspondance des périodes et des sous-phases résultant de l'interpolation périodique-linéaire de coefficients périodiques différents, sachant que ces coefficients sont les termes d'une fonction d'interpolation donnée et doivent par conséquent faire partie de la même boucle ?

3.7.1 Méthode

On commence par définir une fonction notée *autb* qui prend en paramètres une trace t et un niveau de granularité d'analyse i, et produit en sortie les boucles imbriquées correspondantes. L'appel de cette fonction se fait donc par l'instruction $autb(t, i)$.

La trace de départ t de taille $n + 1$ est représentée par un nombre périodique $[x_0, \cdots, x_n]\_{t_0}$, $t_0 = 0 \ldots n$, et on y recherche une interpolation périodique-linéaire.

En résultat, on obtient un certain nombre de phases que nous noterons m, de taille tp_j, $j = 0, \ldots, m-1$, tel que $\sum_{j=0}^{m-1} tp_j = n + 1$, et de période p_j respectivement, ce qui implique un nombre d'intervalles $nb\_int_j = tp_j/p_j$ dans chaque phase. Il peut arriver que deux phases différentes possèdent des intervalles de même période ; on ajoute donc aux composantes des coefficients périodiques un indice portant sur le numéro de la phase, afin de les distinguer.

La succession de boucles correpondantes est donnée à la figure 3.9.

for $t_0 = 0$ to $nb\_int_0 - 1$
$[x_{00}, \cdots, x_{0(p_0-1)}]\_t_1 * t_0 + [y_{00}, \cdots, y_{0(p_0-1)}]\_t_1, \quad t_1 = 0 \ldots p_0 - 1$

for $t_0 = nb\_int_0$ to $nb\_int_0 + nb\_int_1 - 1$
$[x_{10}, \cdots, x_{1(p_1-1)}]\_t_1 * t_0 + [y_{10}, \cdots, y_{1(p_1-1)}]\_t_1, \quad t_1 = 0 \ldots p_1 - 1$
$\vdots$
for $t_0 = nb\_int_{m-2}$ to $nb\_int_{m-2} + nb\_int_{m-1} - 1$
$[x_{(m-1)0}, \cdots, x_{(m-1)(p_{m-1}-1}]\_t_1 * t_0 + [y_{(m-1)0}, \cdots, y_{(m-1)(p_{m-1}-1}]\_t_1,$
$t_1 = 0 \ldots p_{m-1} - 1$

FIG. 3.9 – fonction $aut(t, 0)$

Exemple 3.7.1 *Reprenons la trace de l'exemple 3.4.3 :*
$t = [3, 4, 5, 1, 7, 14, 21, 8, 11, 24, 37, 15, 2, 5, 7, 9, 3, 5, 9, 11, 13, 4, 8, 13, 15, 17, 5, 11, 17, 19, 21, 6]$, *et* $n = 32$.

Pour les étapes de granularités i on commence par $i = 0$ en appelant la fonction $aut(t, 0)$. A ce premier niveau de la hiérarchie, on obtient $m = 2$ phases :

la première phase comporte 12 éléments ($tp_0 = 12$) et est de période 4 ($p_0 = 4$), elle a donc 3 intervalles ($nb\_int_0 = 3$). On a alors la boucle suivante :

for $t_0 = 0$ *to* 2
$[4, 10, 16, 7]_{t_1} t_0 + [3, 4, 5, 1]_{t_1}, \quad 0 \leq t_1 \leq 3 \quad ;$

la deuxième phase couvre tous les éléments restants de la trace ($tp_1 = 20$) et est de période 5 ($p_1 = 5$) ; elle a donc 4 intervalles ($nb\_int_0 = 4$) et on obtient la boucle suivante :

for $t_0 = 3$ *to* 6
$[3, 4, 4, 4, 1]_{t_1} t_0 + [2, 5, 7, 9, 3]_{t_1}, \quad 0 \leq t_1 \leq 4 \quad ;$

En conclusion la trace t se modélise à un premier niveau de granularité par les boucles suivantes :

On remarque que pour chaque phase de niveau $i = 0$ la boucle s'écrit sous la forme

$$
\begin{array}{l}
\text{for } t_0 = 0 \text{ to } 2 \\
\qquad [4,10,16,7]_{t_1}\, t_0 + [3,4,5,1]_{t_1}, \quad 0 \le t_1 \le 3 \quad ; \\
\text{for } t_0 = 3 \text{ to } 6 \\
\qquad [3,4,4,4,1]_{t_1}\, t_0 + [2,5,7,9,3]_{t_1}, \quad 0 \le t_1 \le 4 \quad ;
\end{array}
$$

$$
\begin{array}{c}
\text{for } t_0 = \mathit{deb} \ \text{ to } \mathit{fin} \\
A_{t_1} * t_0 + B_{t_1}
\end{array}
$$

où A et B sont de nouvelles traces de niveau $i = 1$, auxquelles on peut donc également appliquer la fonction *autb*. L'écriture précédente devient :

$$
\begin{array}{c}
\text{for } t_0 = \mathit{deb} \ \text{ to } \mathit{fin} \\
\mathit{autb}(A,1) * t_0 + \mathit{autb}(B,1)
\end{array}
$$

Seulement la fonction *autb* retourne une liste de phases (ou de boucles), et l'écriture $\mathit{autb}(A,1) * t_0 + \mathit{autb}(B,1)$ induit le calcul d'une *union* de phases ou de boucles ; pour ce faire, les phases et les intervalles des deux listes doivent être cohérents, c'est à dire que le découpage en phases doit être similaire des deux côtés avec à chaque fois le même domaine de variation, et les périodes des intervalles doivent également être similaires en taille et en nombre.

Dans l'exemple précédent on a bien une telle cohérence :

- les coefficients $[4,10,16,7]$ et $[3,4,5,1]$ se décomposent chacun en deux sous-phases allant de 0 à 2 et de 3 à 3 respectivement :

$$
\begin{array}{c|c}
[4,10,16,7] & [3,4,5,1] \\
 & \\
0 \le t_1 \le 2 & 0 \le t_1 \le 2 \\
\quad 6t_1 + 4 \ ; & \quad t_1 + 3 \ ; \\
3 \le t_1 \le 3 & 3 \le t_1 \le 3 \\
\quad 7 \ ; & \quad 1 \ ;
\end{array}
$$

- dans les premières sous-phases on observe la même période de 1, dans les secondes sous-phases on observe également la même période ; on peut procéder à l'union des deux phases et on trouve :

$$
\begin{array}{l}
\text{for } t_1 = 0 \text{ to } 2 \\
\qquad (6t_1 + 4)t_0 + t_1 + 3\,; \\
\text{for } t_1 = 3 \text{ to } 3 \\
\qquad 7t_0 + 1 \ ;
\end{array}
$$

– les coefficents $[3,4,4,4,1]$ et $[2,5,7,9,3]$ se décomposent chacun en trois sous-phases allant de 0 à 0, de 1 à 3 et de 4 à 4 respectivement :

$[3,4,4,4,1]$	$[2,5,7,9,3]$
$0 \leq t_1 \leq 0$	$0 \leq t_1 \leq 0$
3 ;	2 ;
$1 \leq t_1 \leq 3$	$1 \leq t_1 \leq 3$
4 ;	$2(t_1 - 1) + 5$;
$4 \leq t_1 \leq 4$	$4 \leq t_1 \leq 4$
1 ;	3 ;

– dans les sous-phases qui se correspondent l'interpolation se fait avec une même période. On procède à l'union des deux phases et on trouve :

$$
\begin{array}{ll}
\text{for } t_1 = 0 \text{ to } 0 & \\
\quad 3t_0 + 2; & \\
\text{for } t_1 = 1 \text{ to } 3 & \\
\quad 4t_0 + 2t_1 + 3; & /*\, 4t_0 + 2(t_1 - 1) + 5 \,*/ \\
\text{for } t_1 = 4 \text{ to } 4 & \\
\quad t_0 + 3; &
\end{array}
$$

La cohérence observée précédemment n'existe pas toujours, comme on peut le voir dans l'exemple suivant.

Exemple 3.7.2 *Supposons qu'on ait de nouvelles traces* $A = [2,5,3,3,4,1]$ *et* $B = [8,17,35,8,5,4]$ *dans lesquelles on trouve une seule sous-phase, avec les boucles correspondantes respectives suivantes :*

for $t_1 = 0$ *to* 2	*for* $t_1 = 0$ *to* 2
$[1,-2]_{t_2} t_1 + [2,5]_{t_2},\ 0 \leq t_2 \leq 1$;	$[0,-12,-31]_{t_2} t_1 + [8,17,35]_{t_2}$; $0 \leq t_2 \leq 2$

*On a bien un même nombre de sous-phasses (*1*), le même domaine de variation pour* t_1 *(de* 0 *à* 2*), mais les périodes des coefficients périodiques sont différents (*2 *et* 3*) : on ne peut plus procéder directement à une union. Il faut auparavant ramener les périodes à leur ppcm, mais dans ce cas de figure, le ppcm est égal à* 6*, et on n'admettra pas d'interpolation d'une telle période étant donnée la taille des traces.*

Pour établir la correspondance entre deux listes de phases on a donc besoin de connaître à l'avance leurs sous-phases, et la fonction $autb(t,i)$ seule n'est plus suffisante pour ce faire. C'est pourquoi on définit une nouvelle fonction notée b_aut qui prend en paramètres deux traces A et B ainsi que leur niveau i, et qui renvoie en sortie l'union de boucles correspondantes. Cette fonction est définie en fonction

de *autb* de la manière suivante :

$$b\_aut(A, B, i) = union(autb(A, i), autb(B, i))$$

3.7.2 Union de phases

La fonction *union* a comme arguments deux listes de m_1 et m_2 phases, de taille tp_{1j} et de période p_{1j} pour la première, avec $j = 0, \ldots, m_1 - 1$, de taille $tp_{2j'}$ et de période et $p_{2j'}$ pour la seconde, avec $j' = 0, \ldots, m_2 - 1$. La cohérence ou encore la mise en correspondance de ces deux listes peut être garantie dans certaines conditions seulement, à savoir :

$$m_1 = m_2 \quad et \quad tp_{1j} = tp_{2j} \tag{3.1}$$
$$m_1 = 1 \quad et \quad p_{10} = 1 \tag{3.2}$$
$$m_2 = 1 \quad et \quad p_{20} = 1 \tag{3.3}$$

La condition 3.1 stipule que les deux listes doivent avoir le même nombre de phases et que les tailles des sous-phases de même niveau doivent aussi être égales. S'il advient que les périodes diffèrent, on les harmonise en les ramenant à leur *ppcm* si cela est possible, et sinon on lève la restriction sur le nombre minimal d'intervalles interpolés.

Les conditions 3.2 et 3.3 nuancent légèrement la première en admettant des nombres différents de phases, pourvu que l'une des traces n'ait qu'une seule phase de période 1, ce qui permettra de la mettre au diapason de la deuxième trace, en terme de découpage.

Différents cas de figure pour l'union de listes

Lorsque l'une au moins des conditions requises est satisfaite, on peut procéder à l'union des listes en considérant à chaque fois une seule phase dans chacune des listes ; soient donc les phases j et j' considérées respectivement dans la première et la seconde liste. On pose

$$nb\_int_j = min(nb\_int_{1j}, nb\_int_{2j'}).$$

La fonction *union* porte donc de manière générale sur deux boucles dont les formes sont les suivantes :

for $t_1 = 0$ to $nb\_int_{1j} - 1$	for $t_1 = 0$ to $nb\_int_{2j'} - 1$
$[x_{j0}, \ldots, x_{j(p_{1j}-1)}]_{t_2} * t_1$	$[x_{j'0}, \ldots, x_{j'(p_{2j'}-1)}]_{t_2} * t_1$
$+ [y_{j0}, \ldots, y_{j(p_{1j}-1)}]_{t_2}$	$+ [y_{j'0}, \ldots, y_{j'(p_{2j'}-1)}]_{t_2}$

1. Si $p_{1j} = p_{2j'} = 1$:
   le résultat est direct, et la boucle en figure 3.9 devient :

$$\begin{array}{l} \text{for } t_0 = \mathit{deb} \text{ to } \mathit{fin} \\ \quad \text{for } t_1 = 0 \text{ to } nb\_int_j - 1 \\ \quad\quad (x_{j0} * t_1 + y_{j0}) * t_0 + (x_{j'0} * t_1 + y_{j'0}) \end{array}$$

Exemple 3.7.3 *On prend* $A = [3, 12, 21]$ *et* $B = [9, 14, 19]$, *et on doit harmoniser les boucles suivantes :*

$$\begin{array}{l|l} \mathit{for}\ t_1 = 0\ \mathit{to}\ 2 & \mathit{for}\ t_1 = 0\ \mathit{to}\ 2 \\ \quad 9t_1 + 3\ ; & \quad 5t_1 + 9\ ; \end{array}$$

La boucle incluant le niveau supérieur est donc :

$$\begin{array}{l} \mathit{for}\ t_0 = \mathit{deb}\ \mathit{to}\ \mathit{fin} \\ \quad \mathit{for}\ t_1 = 0\ \mathit{to}\ 2 \\ \quad\quad (9t_1 + 3) * t_0 + 5t_1 + 9 \end{array}$$

2. Si $p_{1j} = 1$ le résultat devient :

$$\begin{array}{l} \text{for } t_0 = \mathit{deb} \text{ to } \mathit{fin} \\ \quad \text{for } t_1 = 0 \text{ to } nb\_int_j - 1 \\ \quad\quad (x_{j0} * t_1 + y_{j0}) * t_0 \\ \quad\quad\quad + [x_{j'0}, \ldots, x_{j'(p_{2j'}-1)}]_{t_2} * t_1 + [y_{j'0}, \ldots, y_{j'(p_{2j'}-1)}]_{t_2} \end{array}$$

ce qui revient à ajouter le terme $(x_{j0} * t_1 + y_{j0}) * t_0$ à toutes les fonctions issues de $\mathit{baut}([x_{j'0}, \ldots, x_{j'(p_{2j'}-1)}], [y_{j'0}, \ldots, y_{j'(p_{2j'}-1)}], 2)$, tout en mettant à jour l'indice t_1.

Exemple 3.7.4 *On prend* $A = [3, 12, 21, 30]$ *et* $B = [7, 6, 9, 11]$, *et on doit harmoniser les boucles suivantes :*

$$\begin{array}{l|l} \mathit{for}\ t_1 = 0\ \mathit{to}\ 3 & \mathit{for}\ t_1 = 0\ \mathit{to}\ 1 \\ \quad 9t_1 + 3\ ; & \quad [2, 5]_{t_2} t_1 + [7, 6]_{t_2}, 0 \leq t_2 \leq 1\ ; \end{array}$$

On a deux possibilités :

- *soit on travaille sur la deuxième sous-phase et on passe au niveau de granularité suivant, on obtient alors les deux fonctions périodiques* $3t_2 + 2, 0 \leq t_2 \leq 1$ *pour* $[2, 5]$ *et* $-t_2 + 7,\ 0 \leq t_2 \leq 1$ *pour* $[7, 6]$. *Les phases à harmoniser deviennent :*

$$\begin{array}{l|l} \textit{for } t_1 = 0 \textit{ to } 3 & \textit{for } t_1 = 0 \textit{ to } 1 \\ \quad 9t_1 + 3 \ ; & \quad \textit{for } t_2 = 0 \textit{ to } 1 \\ & \quad\quad (3t_2 + 2)t_1 - t_2 + 7 \, ; \end{array}$$

Puisque l'indice t_1 de la première sous-phase varie de 0 à 3, il correspondra à $2t_1 + t_2$ dans l'union d'où le résultat final suivant :

$$\begin{array}{l} \textit{for } t_0 = \textit{deb to fin} \\ \quad \textit{for } t_1 = 0 \textit{ to } 1 \\ \quad\quad \textit{for } t_2 = 0 \textit{ to } 1 \\ \quad\quad\quad (18t_1 + 9t_2 + 3)t_0 + (3t_2 + 2)t_1 - t_2 + 7 \, ; \\ \quad\quad\quad /*(9(2t_1 + t_2) + 3)t_0 + (3t_2 + 2)t_1 - t_2 + 7 */ \, ; \end{array}$$

– *soit on ramène la période de la première sous-phase à 2 (ppcm de $(1, 2)$) et les boucles deviennent :*

$$\begin{array}{l|l} \textit{for } t_1 = 0 \textit{ to } 2 & \textit{for } t_1 = 0 \textit{ to } 1 \\ \quad 18t_1 + [3, 12]_{t_2}, \ 0 \leq t_2 \leq 1 \ ; & \quad [2, 5]_{t_2} t_1 + [7, 6]_{t_2}, 0 \leq t_2 \leq 1 \ ; \end{array}$$

Le coefficient $[3, 12]$ étant interpolé par la fonction $9t_2 + 3$, $0 \leq t_2 \leq 1$, on obtient au niveau de granularité suivant les boucles ci-dessous :

$$\begin{array}{l|l} \textit{for } t_1 = 0 \textit{ to } 1 & \textit{for } t_1 = 0 \textit{ to } 1 \\ \quad \textit{for } t_2 = 0 \textit{ to } 1 & \quad \textit{for } t_2 = 0 \textit{ to } 1 \\ \quad\quad 18t_1 + 9t_2 + 3 \ ; & \quad\quad (3t_2 + 2)t_1 - t_2 + 7 \, ; \end{array}$$

d'où le résultat final de l'union :

$$\begin{array}{l} \textit{for } t_0 = \textit{deb to fin} \\ \quad \textit{for } t_1 = 0 \textit{ to } 1 \\ \quad\quad \textit{for } t_2 = 0 \textit{ to } 1 \\ \quad\quad\quad (18t_1 + 9t_2 + 3)t_0 + (3t_2 + 2)t_1 - t_2 + 7 \, ; \end{array}$$

3. Si $p_{2j'} = 1$, de la même manière que précédemment on aura :

$$\begin{array}{l} \text{for } t_0 = \textit{deb} \text{ to } \textit{fin} \\ \quad \text{for } t_1 = 0 \text{ to } nb\_int_j - 1 \\ \quad\quad ([x_{j0}, \ldots, x_{j(p_{1j}-1)}]_{t_2} * t_1 + [y_{j0}, \ldots, y_{j(p_{1j}-1)}]_{t_2}) * t_0 \\ \quad\quad\quad\quad + (x_{j'0} * t_1 + y_{j'0}) \end{array}$$

4. Si $p_{1j} = p_{2j'}$ et $p_{1j} \neq 1$ on a :

$$\begin{array}{l}\text{for } t_0 = deb \text{ to } fin\\ \quad \text{for } t_1 = 0 \text{ to } nb\_int_j - 1\\ \quad\quad ([x_{j0}, \ldots, x_{j(p_{1j}-1)}]_{t_2} * t_1 + [y_{j0}, \ldots, y_{j(p_{1j}-1)}]_{t_2}) * t_0 +\\ \quad\quad\quad [x_{j'0}, \ldots, x_{j'(p_{1j}-1)}]_{t_2} * t_1 + [y_{j'0}, \ldots, y_{j'(p_{1j}-1)}]_{t_2}\end{array}$$

c'est à dire qu'on multiplie par t_0 toutes les fonctions issues de *baut*$([x_{j0}, \ldots, x_{j(p_{1j}-1)}]$, $[y_{j0}, \ldots, y_{j(p_{1j}-1)}], 2)$ et qu'on rajoute les fonctions correspondantes de *baut*$([x_{j0}, \ldots, x_{j(p_{1j}-1)}]$, $[y_{j'0}, \ldots, y_{j(p_{1j}-1)}], 2)$; en guise d'exemple, se référer au cas 2, deuxième solution, à partir du moment où l'on ramène les périodes des coefficents périodiques à leur *ppcm*.

5. Si $p_{1j} \neq p_{2j'}$:
   on choisit comme nouvelle période $ppcm(p_{1j}, p_{2j'})$.
   Si $3 * ppcm(p_{1j}, p_{2j'}) > tp_{1j}$ alors il n'y a pas d'interpolation périodique-linéaire, et on se restreindra dans cette phase à deux intervalles de période $tp_{1j}/2$.

Il arrive parfois qu'en partant de deux listes de phases répondant aux critères 3.1, 3.2 ou 3.3, on arrive à une n-ième étape d'itération où les sous-phases ne sont plus cohérentes. Dans ces conditions, on passe à une harmonisation de ces sous-phases.

3.7.3 Harmonisation de listes de phases

On considère deux traces modélisées par deux listes de phases l_1 et l_2 de taille m_1 et m_2, m_1 différent de m_2. L'harmonisation consiste à redécomposer les traces de départ selon les intersections des phases trouvées en l_1 et l_2.

Si l'intersection concerne des phases contenant 1 seul ou 2 éléments, la nouvelle période est égale à 1 ; dans les autres cas, elle est égale au *ppcm* des périodes des phases intersectées, à condition toutefois que la taille des phases soit assez grande pour contenir au moins 3 intervalles de telle période. Sinon il n'y a pas vraiment d'interpolation périodique-linéaire et la période est mise à $taille\_phase/2$.

Exemple 3.7.5 *On considère la fonction d'interpolation suivante, à un niveau donné :* $[4, 6, 7, 8, 9, 3]_{t_4} t_3 + [6, 1, 2, 3, 4, 5]_{t_4}$, $0 \leq t_4 \leq 5$.

Pour le coefficient $[4, 6, 7, 8, 9, 3]$, *on trouve dans un premier temps* 3 *phases, tandis que pour le coefficient* $[6, 1, 2, 3, 4, 5]$ *on en trouve* 2. *Les boucles correspondantes sont les suivantes, respectivement :*

for $t_4 = 0$ *to* 1 $2t_4 + 4$; *for* $t_4 = 2$ *to* 4 $t_4 + 5$; *for* $t_4 = 5$ *to* 5 3 ;	*for* $t_4 = 0$ *to* 0 6 ; *for* $t_4 = 1$ *to* 5 t_4 ;

Les domaines de variation des sous-phases étant différents, on détermine les intersections qui conservent au mieux les fonctions d'interpolation précédentes et on trouve $0 \leq t_4 \leq 0$, $1 \leq t_4 \leq 1$, $2 \leq t_4 \leq 4$ *et* $5 \leq t_4 \leq 5$. *L'union se fait suivant ces nouveaux domaines et on obtient comme boucle :*

for $t_3 =$ *deb to fin*
  for $t_4 = 0$ *to* 0
    $4t_3 + 6$;
  for $t_4 = 1$ *to* 1
    $6t_3 + 1$;
  for $t_4 = 2$ *to* 4
    $(t_4 + 5)t_3 + t_4$;
  for $t_4 = 5$ *to* 5
    $3t_3 + 5$;

On constate qu'on peut terminer une harmonisation de phases avec un nombre important de petites sous-phases. Dans les cas où l'on choisit pour une sous-phase donnée une période de $taille\_phase/2$ (pas d'interpolation périodique linéaire au sens strict), on impose directement que toutes les étapes récursives ultérieures (basées sur les éléments de cette même phase) opèrent également sur $taille\_phase'/2$ c'est à dire $taille\_phase/4$, et ainsi de suite, ceci afin de limiter le nombre de petites sous-phases qui pourraient en résulter.

Remarque 3.7.1 *L'harmonisation de phases peut être utilisée pour la comparaison de traces, grâce à l'intersection effectuée au cours de l'analyse ; par exemple on pourrait rechercher des phases qui se correspondent mutuellement et qui proviennent de traces quelconques, en se basant juste sur leurs fonctions d'interpolation et sur les sous-phases qu'elles contiennent.*

3.8 Conclusion

Le modèle périodique-linéaire nous permet de représenter une trace de programme sous forme de boucles imbriquées, exprimant ainsi le comportement d'un

programme par un autre programme, qui traduit non pas la fonctionnalité du programme analysé, mais plutôt la manière de remplir cette fonctionnalité.

C'est un modèle assez flexible puisqu'il se décline sous plusieurs configurations : intervalles adjacents de taille constante, intervalles adjacents de taille variable, intervalles distants de taille constante et intervalles distants de taille variable. A travers ces différentes configurations, il est possible de mettre en relief des comportements consécutifs du programme ou même des comportements entrelacés, dont la durée de répétition peut varier au cours de l'exécution. De plus, les fonctions exprimant l'interpolation peuvent être soit linéaires, soit polynomiales par une composition de fonctions linéaires.

C'est un modèle extensible, car il est possible le cas échéant de se limiter à des approximations, d'introduire une paramétrisation des données ou encore d'effectuer la modélisation sous forme de récurrences linéaires périodiques.

Dans la séquence de nids de boucles résultante, les instructions des boucles les plus internes sont des fonctions, et les bornes des boucles sont soit des constantes, soit des fonctions affines des indices de boucles. Les fonctions de niveau les plus internes expriment les valeurs de la trace d'entrée à partir des indices de boucles. C'est une représentation typique qui permet ensuite d'effectuer des analyses avancées grâce à l'utilisation d'outils d'analyse statique, et le modèle d'analyse statique employé est naturellement le modèle polyédrique dédié à l'analyse des nids de boucles.

Dans le chapitre suivant nous présentons quelques unes de ces analyses, et nous verrons alors que la visualisation graphique sous forme de polyèdres, ainsi que l'arithmétique des polyèdres constituent une riche interface d'aide à l'analyse et à la compréhension. Nous montrons également comment notre approche permet de guider et de générer des optimisations statiques ou dynamiques de programmes, telles que l'optimisation de la localité spatiale des données ou le pré-chargement dynamique de données, ce qui permet, comme on le note au passage, la mise en oeuvre d'une collaboration entre analyse dynamique et analyse statique.

Chapitre 4

Utilisation et applications du modèle périodique-linéaire

La plupart des applications, qu'elles soient scientifiques, financières, commerciales ou autres, manipulent de vastes ensembles de données qui doivent être traitées de manière adéquate, si l'on veut pouvoir profiter de l'accroissement de la puissance de calcul des processeurs. Or, à cause de l'écart entre la vitesse de traitement des processeurs d'une part, et les temps d'accès aux mémoires caches, vives et externes d'autre part, des goulets d'étranglement se produisent assez fréquemment. L'une des solutions à ce problème crucial passe par une analyse et une amélioration du comportement mémoire des programmes, et nous illustrons dans les paragraphes suivants de quelle manière le modèle périodique-linéaire contribue à cette analyse de comportement mémoire.

La section 4.1 précise les raisons du choix d'illustrer les applications du modèle périodique-linéaire par l'analyse du comportement mémoire des programmes, dans un premier temps. Après un rappel de la démarche emloyée pour représenter les traces étudiées, nous passons aux applications du modèle qui consistent en plusieurs expérimentations.

En section 4.2, nous montrons comment transformer de pointeurs en accès tableaux explicites et en 4.3 nous présentons une catégorie de traces de programmes au comportement mixte, c'est à dire déterministe seulement en partie ; nous parvenons tout de même à obtenir une modélisation de ces traces grâce à la forme hydride du modèle périodique-linéaire.

La section 4.4 présente quelques optimisations de programmes réalisées suite à l'application des instructions de préchargement de données, sous l'impulsion de notre modélisation.

Pour terminer, diverses applications dérivées de l'adjonction du modèle polyédrique sont présentées à la section 4.5, à savoir : la visualisation graphique des données, l'analyse de réutilisation des variables et l'analyse de la mémoire cache.

4.1 Modélisation du comportement mémoire

La compréhension et la maîtrise du comportement mémoire sont des tâches clés qui permettent par exemple d'optimiser les performances des programmes, de diminuer la consommation en temps et en énergie plus particulièrement des applications embarquées, de réduire les coûts de fabrication du matériel, et encore de satisfaire les contraintes inhérentes aux systèmes temps réel.

Cette maîtrise du comportement mémoire n'est de toute évidence pas un exercice facile à mener, encore moins statiquement, du moment où plusieurs des accès mémoire sont inconnus avant l'exécution, comme c'est le cas pour des accès via des structures de type pointeurs. De plus, la complexité croissante des architectures et le développement des techniques d'allocation dynamiques constituent également des obstacles quasi-impossibles à surmonter du point de vue de l'analyse statique. C'est pourquoi on utilise plutôt des instrumentations de codes pour générer des profils d'accès mémoire qu'on espère représentatifs du comportement à analyser, et l'un des défis de cette analyse réside dans la taille des traces qui peut atteindre des centaines de milliers de valeurs.

A la suite de l'instrumentation, nous représentons la trace générée à l'aide du modèle périodique-linéaire, sous la forme de boucles imbriquées dont les indices définissent un espace temps multi-dimensionnel ; les éléments de la trace correspondent aux itérations des boucles, chaque boucle étant associée à une *phase* du programme dans laquelle les accès correspondent à des motifs linéairement dépendants.

L'analyse des boucles obtenues permet d'obtenir des informations intéressantes sur le comportement du programme de départ, et partant de là, il peut s'ensuivre des stratégies visant à améliorer les programmes, ou encore des transformations optimisantes comme c'est le cas dans les exemples qui suivent.

4.2 Transformation de pointeurs en accès tableaux explicites

La spécialisation des subdivisons de mémoire dans les applications embarquées est une part importante dans la conception des tels systèmes, et permet d'atteindre les objectifs fixés en terme de performance et de consommation énergétique en particulier. Une grande attention a donc été portée à l'utilisation des mémoires brouillons ou encore mémoires blocs-notes (*scratch pad memories*) [78]. Ce sont des mémoires aux consommations électriques moindres que celles des mémoires caches, et qui possèdent en plus des latences facilement prévisibles, caractéristique extrêmement importante pour les applications temps-réel notamment.

Plusieurs des techniques automatiques de détermination des configurations de ces mémoires reposent sur l'analyse de programmes à la compilation. Malheureu-

semment, ces approches d'analyses limitent le champ des optimisations mémoires possibles, car elles supposent que :

- les programmes sont écrits d'une manière bien structurée, dans laquelle tous les accès répétés à la mémoire surviennent dans des boucles *for* ;
- tous les accès mémoire sont générés par des références à des tableaux dont les indices ont des expressions affines.

Or il n'en va pas toujours ainsi. Pour surmonter ces problèmes, nous montrons comment notre modèle peut être utilisé, pour transformer par exemple des accès pointeurs en références tableaux.

4.2.1 Instrumentation et traçage

Considérons le programme `fir2dim` dont la boucle principale est donnée à la figure 4.2. C'est un programme tiré de la suite de bancs d'essai DSPstone [85], et qui implémente le filtre d'une image $2D$ par une matrice de convolution 3×3.

L'instrumentation du code a visé dans un premier temps à recueillir les valeurs prises par les différents pointeurs `parray`, `parray2` et `parray3`, et la trace présentée à la figure 4.1 a été obtenue en fixant à 4 le paramètre `IMAGEDIM`.

Dans cette trace, les adresses référencées sont représentées sous forme d'entiers, et la première valeur a été soustraite de toutes les valeurs suivantes, ce qui fait que les adresses analysées sont donc toutes relatives à un certain déplacement.

```
0 1 2 6 7 8 12 13 14 1 2 3 7 8 9 13 14 15 2 3 4 8 9 10 14 15 16 3 4 5 9 10 11 15
16 17 6 7 8 12 13 14 18 19 20 7 8 9 13 14 15 19 20 21 8 9 10 14 15 16 20 21 22
9 10 11 15 16 17 21 22 23 12 13 14 18 19 20 24 25 26 13 14 15 19 20 21 25 26
27 14 15 16 20 21 22 26 27 28 15 16 17 21 22 23 27 28 29 18 19 20 24 25 26 30
31 32 19 20 21 25 26 27 31 32 33 20 21 22 26 27 28 32 33 34 21 22 23 27 28 29
33 34 35
```

FIG. 4.1 – Valeurs prises par les pointeurs `parray`, `parray2` et `parray3` dans le programme `fir2dim`, avec `IMAGEDIM`= 4.

```
for (k = 0 ; k < IMAGEDIM ; k++)
{
      for (f = 0 ; f < IMAGEDIM ; f++)
      {
       pcoeff = &coefficients[0] ;
       parray = &array[k*ARRAYDIM + f] ;
       parray2 = parray + ARRAYDIM ;
       parray3 = parray + ARRAYDIM + ARRAYDIM ;

       *poutput = 0 ;

       for (i = 0 ; i < 3 ; i++)
           *poutput += *pcoeff++ * *parray++ ;

       for (i = 0 ; i < 3 ; i++)
           *poutput += *pcoeff++ * *parray2++ ;

       for (i = 0 ; i < 3 ; i++) {
           *poutput += *pcoeff++ * *parray3++ ;

       poutput++ ;
       }
}
```

FIG. 4.2 – Boucle principale du programme `fir2dim`.

4.2.2 Analyse à l'aide du modèle périodique-linéaire et transformation

Les résultats présentés dans cette partie concernent la version du modèle périodique-linéaire dite des intervalles adjacents de taille constante.

1. En appliquant une première passe du modèle périodique-linéaire à la trace de la figure 4.1, on trouve la fonction d'interpolation suivante :

$$f_1(t_1) = 6 * t_1 + [0,1,2,6,7,8,12,13,14,1,2,3,7,8,9,13,\\ 14,15,2,3,4,8,9,10,14,15,16,3,4,5,9,10,11,15,16,17]_{t_2}, \quad \begin{array}{l} 0 \leq t_1 \leq 3, \\ 0 \leq t_2 \leq 35 \end{array}$$

2. Pour la seconde dimension de l'espace temps, on s'intéresse au deuxième coefficient noté b, car il est de période supérieure à 1.

$$b = [0,1,2,6,7,8,12,13,14,1,2,3,7,8,9,13,14,15,\\ 2,3,4,8,9,10,14,15,16,3,4,5,9,10,11,15,16,17]_{t_2}$$

Avec *PLI* on trouve :

$$f_2(t_2) = t_2 + [0,1,2,6,7,8,12,13,14]_{t_3}, \quad 0 \leq t_2 \leq 3 \text{ et } 0 \leq t_3 \leq 8$$

3. En interpolant le coefficient périodique $[0,1,2,6,7,8,12,13,14]_{t_3}$, on arrive à :
$$f_3(t_3) = 6 * t_3 + [0,1,2]_{t_4}, \quad 0 \leq t_3 \leq 2, \quad 0 \leq t_4 \leq 2$$
4. Et enfin, le périodique $[0,1,2]_{t_4}$ est interpolé par la fonction
$$f_4(t_4) = t_4, \quad 0 \leq t_4 \leq 2$$

Au final, la trace en figure 4.1 est représentée par les boucles suivantes :

for $t_1 = 0$ to 3
  for $t_2 = 0$ to 3
    for $t_3 = 0$ to 2
      for $t_4 = 0$ to 2
        $6 * t_1 + t_2 + 6 * t_3 + t_4$;

En faisant varier les valeurs de `IMAGEDIM`, nous avons pu modéliser plusieurs traces différentes et observer que dans tous les cas, les bornes supérieures des indices t_1 et t_2 correspondent à `IMAGEDIM` $- 1$, tandis que les coefficients des fonctions de référence pour les indices t_1 et t_3 correpondent à `ARRAYDIM=IMAGEDIM+2`.

Remarque 4.2.1 *Cette valeur de* `ARRAYDIM=IMAGEDIM+2` *vient du fait que la matrice de taille* `ARRAYDIM` $\times$ `ARRAYDIM` *représentant l'image à filtrer, est entourée de* 0 *avant l'application du filtre, d'où l'ajout de ces deux lignes et deux colonnes.*

La boucle finale générale est donc celle de la figure 4.3.

for $t_1 = 0$ to $IMAGEDIM - 1$
  for $t_2 = 0$ to $IMAGEDIM - 1$
    for $t_3 = 0$ to 2
      for $t_4 = 0$ to 2
        $ARRAYDIM * t_1 + t_2 + ARRAYDIM * t_3 + t_4$;

FIG. 4.3 – Boucles modélisant des adresses mémoires référencées à travers des pointeurs.

La fonction de référence $ARRAYDIM * t_1 + t_2 + ARRAYDIM * t_3 + t_4$ peut être perçue comme un accès à un tableau linéarisé, de taille $(ARRAYDIM \times ARRAYDIM)$, et de cette manière la fonction de référence au tableau initial de dimension 2 est de la forme $array[t_1 + t_3][t_2 + t_4]$.

4.2.3 Autres pointeurs et résultat final

De la même manière que précédemment, on modélise les accès effectués à travers les pointeurs `poutput` et `pcoeff` ; on aboutit finalement à une reécriture de la boucle principale de `fir2dim` en celle de la figure 4.4, dans laquelle tous les accès mémoire sont générés par des fonctions de référence à un tableau de dimension 2, avec des expressions affines des indices des boucles.

```
for (k = 0 ; k < IMAGEDIM ; k++)
      for (f = 0 ; f < IMAGEDIM ; f++)
      {
       output[k][f] = 0 ;
       for (i = 0 ; i < 3 ; i++)
           for (j = 0 ; j < 3 ; j++)
                output[k][f] += coefficients[i][j]
                              * array[k+i][f+j] ;
      }
```

FIG. 4.4 – Reécriture de la boucle principale du programme `fir2dim`.

4.3 Un modèle hybride

Il arrive parfois que des caractéristiques indépendantes des entrées de programme s'entrelacent avec d'autres caractéristiques dépendantes des entrées de programme, rendant ainsi la modélisation plus ardue, voire impossible. Dans ce contexte, en faisant abstraction des caractéristiques liées aux entrées de programme, on peut trouver le cas échéant un modèle hybride qui décrit le comportement linéaire et périodique des événements non déterministes.

Prenons par exemple le programme `ks` des bancs d'essai *pointer intensive* [12], qui implémente le partitionnement de graphes à l'aide des heuristiques de Kernighan-Lin ou de Kernighan-Schweikert. Dans la fonction la plus coûteuse en temps qui est `FindMaxGpAndSwap`, considérons les accès mémoire effectués à travers le pointeur `mrB`. Dans la trace obtenue on remarque que :

- les mêmes séquences d'adresses sont référencées successivement, de nombreuses fois ;

- elles sont suivies de nouvelles séquences qui leur sont d'ailleurs semblables, avec seulement un élément en moins, et qui sont aussi référencées successi-

vement de manière répétée ;

- au fur et à mesure qu'un élément est soustrait d'une séquence pour en former une nouvelle, on arrive à une séquence d'un seul élément ;

- ensuite, les 3 étapes précédentes recommencent sur une toute nouvelle séquence de `numModules/2` éléments, référencée de manière successive, et ainsi de suite (`numModules` est la seconde valeur du fichier d'entrée du programme) ;

- une séquence est accédée autant de fois qu'elle a d'éléments : une séquence de 10 éléments est accédée à 10 reprises, une séquence de 9 éléments 9 fois, etc ;

- il n'y a pas d'interpolation périodique-linéaire pour les valeurs contenues dans une séquence ;

- les éléments extraits des séquences au fil du processus ne peuvent être déterminés à l'avance et dépendent du fichier d'entrée de programme considéré.

Plusieurs traces de `ks` générées pour des fichiers d'entrée différents peuvent être trouvées en [7]. La trace dont un extrait est donné à la figure 4.5 comprend 7 grandes séquences au total, chacune d'entre elles donnant lieu à d'autres nouvelles séquences.

On voit qu'il est possible de prédire, dans ce cas de figure, des accès à des séquences non déterministes mais de tailles connues. Le modéle hybride va consister en une phase d'apprentissage pour stocker ces séquences d'adresses, suivie d'une phase prédictive qui donnera de manière exacte les adresses suivantes. Le résultat est représenté sous forme de boucles à la figure 4.6.

```
4591400 4591432 4591464 4591496 4591528 4591560 4591592 4591624 4591656 4591688
4591400 4591432 4591464 4591496 4591528 4591560 4591592 4591624 4591656 4591688
                                        ⋮   8 fois
4591432 4591464 4591496 4591528 4591560 4591592 4591624 4591656 4591688
4591432 4591464 4591496 4591528 4591560 4591592 4591624 4591656 4591688
                                        ⋮   7 fois
4591432 4591496 4591528 4591560 4591592 4591624 4591656 4591688
                                        ⋮   7 fois
4591432 4591496 4591528 4591560 4591592 4591624 4591688
                                        ⋮   6 fois
4591432 4591528 4591560 4591592 4591624 4591688
                                ⋮   5 fois
4591432 4591528 4591592 4591624 4591688
                                ⋮   4 fois
4591432 4591528 4591624 4591688
                                ⋮   3 fois
4591432 4591528 4591624
                                ⋮   2 fois
4591432 4591528
4591432 4591528

4591528

4591384 4591464 4591656 4591496 4591560 4591592 4591688 4591624 4591432 4591528
4591384 4591464 4591656 4591496 4591560 4591592 4591688 4591624 4591432 4591528
                                        ⋮   8 fois
4591384 4591464 4591496 4591560 4591592 4591688 4591624 4591432 4591528
4591384 4591464 4591496 4591560 4591592 4591688 4591624 4591432 4591528
                                        ⋮   7 fois
                                        ⋮
                                        ⋮
                                        ⋮
```

FIG. 4.5 – Extrait de la trace du programme `ks` avec un premier fichier d'entrée.

```
M = numModules/2 − 1
for t1 = 0 to N
  for t2 = 0 to M
    for t3 = 0 to 0                          // * Phase d'apprentissage *
      for t4 = 0 to M − t2
        T[t4] =addresses référencées ; // valeurs stockées dans un tableau de taille M
    for t3 = 1 to M − t2                     // * Phase predictive *
      for t4 = 0 to M − t2
        T[t4] ;
```

FIG. 4.6 – modèle hybride représentant le comportement mémoire du programme `ks`.

4.4 Préchargement des données

Considérons à présent le programme `mcf` des bancs d'essai Spec2000 [8], qui implémente la gestion du trafic d'un dépôt de véhicules par l'algorithme du simplex. Dans ce programme, la fonction `price_out_impl` prend 31% du temps total d'exécution, et l'analyse de son code source montre que 2 instructions principales font référence à des structures de données définies par des listes chaînées. On instrumente donc le code pour recueillir dans deux fichiers les adresses mémoire virtuelles référencées, et on exécute le programme avec le fichier d'entrée `test.in` fourni par les bancs d'essai.

Pour les deux instructions identifiées, on construit un modèle hybride d'intervalles adjacents de taille variable, contenus dans des intervalles adjacents de taille constante. Suivant la première dimension t_1 les intervalles de taille constante sont identiques. Suivant la deuxième dimension, les tailles des intervalles successifs croissent d'un élément à chaque intervalle, et les éléments linéairement dépendants entre intervalles successifs diffèrent de 120 pour la première instruction, et de 192 pour la seconde. De plus, à l'intérieur d'un intervalle, les valeurs sont décrémentées de 120 dans la première trace et de 192 dans la seconde, et ce jusqu'à atteindre la valeur 0. Le modèle en trois dimensions représentant entièrement la trace est présenté sous forme de boucles dans la table 4.1.

Les valeurs M et $N = nb\_timetabled\_trips - 2$ varient en fonction du fichier d'entrée utilisé par le programme. La valeur $M + 1$ représente le nombre d'intervalles de taille constante dans la première dimension, tandis que la valeur $N + 1$ représente leur taille.

Programme : fonction opt.	**Modèle**	**Orig. time**	**Opt. time**	**Speedup**
mcf : price_out_impl	for $t_1 = 0$ to M for $t_2 = 0$ to $nb\_timetabled\_trips - 2$ for $t_3 = 0$ to t_2 $120t_2 - 120t_3 + offset$;	512 sec.	405 sec.	20%
equake : smvp	for $t_1 = 0$ to $timesteps - 1$ for $t_2 = 0$ to $N - 1$ for $t_3 = 0$ to 2 $128t_2 + 32t_3 + offset$;	350 sec.	262 sec.	25%

TAB. 4.1 – Modèle de boucles imbriquées, temps d'exécution et accélération.

L'analyse des traces correspondant aux trois fichiers d'entrée `test.in`, `train.in` and `ref.in` fournis par les SPEC2000 permet d'identifier les valeurs de M et N associées et de valider l'adéquation du modèle. On remarque alors que les tailles des intervalles sont données directement par les premiers paramètres des fichiers d'entrée, et d'après la documentation de `mcf` ce premier paramètre est défini comme *le nombre de planning de voyages*, soit le nombre de véhicules. Il est égal à $N + 2$.

Par contre, le nombre d'intervalles ne peut pas être directement relié à un paramètre d'entrée, étant donné qu'il dépend de la vitesse de convergence du processus d'optimisation implémenté pour résoudre le problème de programmation linéaire posé.

Cependant, on peut tout de même construire un modèle global, puisque les valeurs des dimensions t_2 et t_3 ne dépendent pas du tout de t_1. De plus, l'utilisation de ce modèle pour des optimisations dynamiques n'est pas handicapée par l'ignorance de M, car le processus d'optimisation s'effectue jusqu'à la fin de l'exécution du programme.

On utilise donc les deux modèles générés pour les deux instructions pour implémenter un mécanisme de préchargement dynamique de données, ceci afin d'améliorer les performances du programme ; le processeur cible est un *Itanium-2*. Le mécanisme en question est construit comme deux fonctions de préchargement de données qui agissent 3 accès en avance, en fonction des adresses calculées par nos modèles. Ces fonctions sont appelées avant chaque accès mémoire de la fonction `price_out_impl`. En exécutant la nouvelle version du programme sur les données d'entrée de référence, on arrive à des accélérations non négligeables, comme on le voit à la table 4.1 (le programme original et la version optimisée ont été compilés avec l'option $-O3$).

De la même manière on modélise le programme `equake` pris dans les SPEC2000, et les résultats sont reportés à la table 4.1.

4.5 Utilisation du modèle polyédrique

Le modèle polyédrique de nids de boucles [36] est classiquement utilisé pour l'analyse des programmes contenant des boucles imbriquées, à la manière des programmes fortran. Il consiste a représenter géométriquement des boucles imbriquées sous forme de polyèdre borné (encore appelé polytope) dont les points entiers sont associés aux itérations des boucles. Les bornes de boucles définissent des équations qui à leur tour délimitent le polytope, et les coordonnées des points entiers sont données par toutes les valeurs possibles des indices de boucles. C'est ainsi qu'un nid de boucles de profondeur d peut être représenté par un polytope de dimension d, tandis qu'une succession de nids de boucles dont les domaines de variation des indices des boucles les plus externes sont disjoints sera quant à elle représentée par plusieurs polytopes disjoints.

Utilisé dans le cadre de l'analyse du comportement mémoire des programmes, ce modèle s'avère être un outil d'aide supplémentaire et précieux pour une plus grande compréhension, grâce aux représentations graphiques qu'il permet d'effectuer. Il fournit un moyen de visualisation des données au même titre que les techniques de visualisation développées en fouilles de données [28]. Dans ce cadre, il analyse la représentation sous forme de nids de boucles issue de l'application du modèle périodique-linéaire à une trace d'entrée.

Outre les adresses mémoires, la trace de départ peut contenir des adresses de blocs mémoire, des numéros de lignes de cache ou encore une liste de nombres de défauts de cache (obtenus soit par instrumentation, soit par échantillonnage des registres d'événements du processeur ou encore grâce à des simulateurs de cache). Le tableau de la figure 4.7 mentionne quelques unes des caractéristiques pour lesquelles l'on peut effectuer une représentation graphique ou exprimer paramétriquement des fonctions associées, à l'aide du modèle polyédrique.

4.5.1 Environnement d'analyse

La représentation graphique et le calcul des fonctions mentionnés ci-dessus font partie d'un environnement général d'outils d'analyse schématisé à la figure 4.8, intégrant les trois outils suivants :

- *PLI :* notre outil d'interpolation périodique-linéaire.
- *Polylib [6] :* librairie implémentant plusieurs fonctions pour la manipulation de polyèdres, paramétrés ou non, dans l'espace des nombres rationnels.
- *barvinok [82, 83] :* programme de calcul du nombre de points entiers d'un polyèdre paramétré, i.e le polynôme d'Ehrhart [26] de ce polyèdre.

Visualisations graphiques :	Fonctions paramétrées :
• phases d'accès mémoire	
• accès à une addresse A	• frequence des accès à une adresse A (adresse mémoire, indice de bloc, ...)
• comportement du cache	• fréquence des défauts de cache lors d'un accès à A
• adresse mémoire associée à une distance de réutilisation R	• addresse et distance de réutilisation du t^{eme} accès mémoire

FIG. 4.7 – Visualisations et fonctions à calculer grâce au modèle polyédrique.

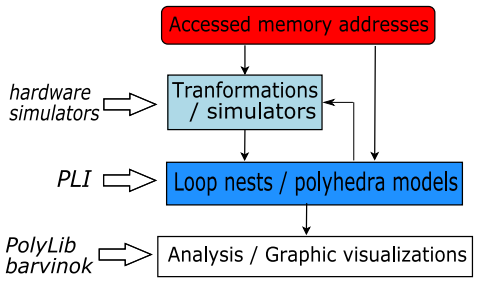


FIG. 4.8 – Outils composant l'environnement d'analyse

.

La trace de départ peut être transformée lors d'une phase de prétraitement, ou servir d'entrée à un simulateur matériel (simulateur de cache par exemple), dans le but de générer une nouvelle trace d'entrée. Elle peut également être directement analysée par l'outil d'interpolation périodique-linéaire et mise sous forme de boucles imbriquées, lesquelles boucles sont ensuite modélisées par des polytopes. Ainsi, plusieurs graphiques peuvent être représentés à partir des boucles, et on peut calculer plusieurs fonctions liées aux caractéristiques du comportement du programme.

Remarque 4.5.1 *Il est aussi possible d'instrumenter les nids de boucles eux-mêmes et de les exécuter afin de générer de nouvelles traces pour d'éventuelles analyses ultérieures. En effet, on peut éviter de re-exécuter le programme original pour observer une autre caractéristique du comportement, car il suffit alors de partir des nids de boucles obtenus. D'autre part, on peut contourner et résoudre un problème mathématique complexe en générant la trace de ses solutions et en effectuant plutôt la modélisation sur celle-ci.*

4.5.2 Visualisations et calcul de fonctions paramétrées

Phases d'accès mémoire

On considère une trace d'entrée, constituée d'adresses mémoire successivement référencées au cours d'une exécution de programme, soit en lecture soit en écriture. Après application du modèle périodique-linéaire, on obtient une séquence de boucles imbriquées où les boucles les plus internes représentent un polynôme dont les variables sont les indices de boucles et dont les valeurs calculées sont celles de la trace d'entrée. On rappelle que tous les indices de boucles définissent ensemble un espace-temps multidimensionnel, selon un ordre lexicographique qui détermine par ailleurs l'ordre d'accès aux adresses mémoire.

Cette représentation met en exergue le comportement mémoire du programme dont le profil d'accès mémoire a été analysé, et fait aussi apparaître les éléments suivants :

- les répétitions de motifs linéairement dépendants et de séquences d'accès à la mémoire, de même que le nombre de telles répétitions,
- la fréquence d'apparition des accès mémoire linéairement dépendants ou plutôt leur périodicité, qui est une fonction de l'étendue du domaine de variation des indices de boucles,
- les relations linéaires entre adresses mémoire dont les accès sont également espacés, données par les fonctions dans les boucles les plus internes,
- la séquence de phases imbriquées caractérisées par les propriétés précédentes, donnée par la succession de boucles imbriquées.

Puisque toutes les boucles imbriquées générées peuvent être représentées par des polytopes, chacun de ces polytopes est associé à une phase du comportement d'accès à la mémoire ; dans chaque phase, des motifs d'accès mémoire linéairement dépendants sont accédés et une représentation graphique des polytopes associés fournit alors une représentation visuelle claire de ce comportement mémoire.

Cas étudié :

Reprenons le programme `fir2dim` (voir figure 4.2) de la suite de bancs d'essai DSPStone [85], instrumenté cette fois avec `IMAGEDIM`= 100 pour obtenir une trace des pointeurs `parray`, `parray2` et `parray3`. Sachant que les objets pointés ont chacun une taille de 4 octets, on divise chaque valeur de la trace par 8 pour obtenir les références aux blocs mémoire qui ont eux une taille de 32 octets.

Soit donc une trace de 90000 accès successifs aux blocs mémoire ; le modèle périodique-linéaire produit les boucles de la figure 4.9, qui représentent exactement toute la trace d'entrée.

Les itérations des dimensions t_1, t_2, t_3 sont representées comme les points entiers contenus dans les deux polytopes de la figure 4.10, qui sont définis par les

```
offset = 530832;
for t1 = 0 to 99
  for t2 = 0 to 49
  {
    for t3 = 0 to 2
    {
      for t4 = 0 to 1
        accessed_block = 51t1 + t2 + 51t3 + offset;
      for t4 = 2 to 2
        accessed_block = 51t1 + t2 + 51t3 + 1 + offset;
    }
    for t3 = 3 to 5
    {
      for t4 = 0 to 1
        accessed_block = 51t1 + t2 + 51t3 − 153 + offset;
      for t4 = 2 to 2
        accessed_block = 51t1 + t2 + 51t3 − 152 + offset;
    }
  }
```

FIG. 4.9 – Boucles imbriquées représentant les accès aux blocs mémoire.

inéquations 4.1 :

$$\left\{ \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \leq t_1 \leq 99 \\ 0 \leq t_2 \leq 49 \\ 0 \leq t_3 \leq 2 \end{array} \right\} \cup \left\{ \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \leq t_1 \leq 99 \\ 0 \leq t_2 \leq 49 \\ 3 \leq t_3 \leq 5 \end{array} \right\} \tag{4.1}$$

Chaque polytope représente une phase différente d'accès aux blocs, selon la classification du modèle périodique-linéaire, et chaque phase est associée à une boucle différente indicée par t_3.

En fonction de la granularité choisie, chaque point entier représente 3 accès aux blocs mémoire, correspondant à 3 itérations des boucles en t_4. L'ordre d'accès est donné par l'ordre lexicographique de t_1, t_2, t_3.

A partir de cette représentation à l'aide de polytopes, plusieurs autres informations peuvent être mises en évidence.

Fréquence d'accès

Les points entiers associés aux accès à une adresse mémoire A sont définis comme les points contenus dans les intersections de chaque polytope avec les hyperplans définis par les équations stipulant que les fonctions de références des boucles les plus internes sont égales à A. Ces ensembles sont représentés par des hyperplans qui coupent les polytopes.

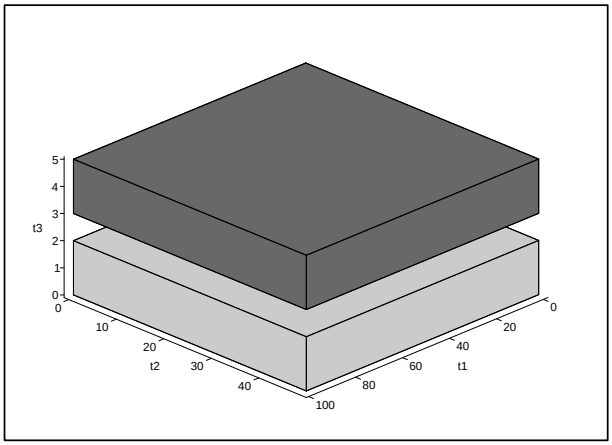


FIG. 4.10 – Polytopes representant deux phases d'accès aux blocs mémoire.

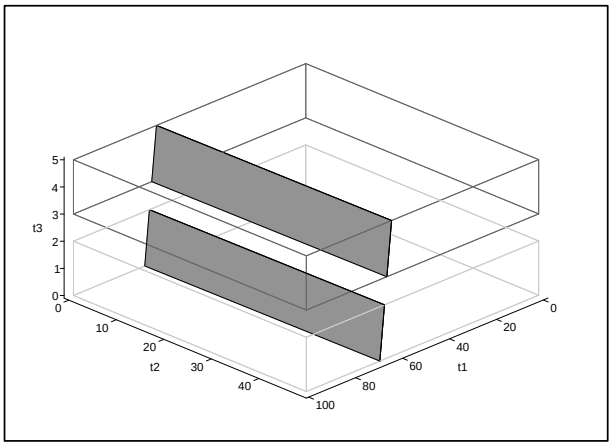


FIG. 4.11 – Plans représentant les accès au bloc mémoire 534332.

Exemple 1 On veut identifier tous les accès au bloc mémoire 534332. Il s'agit des points entiers des polytopes en figure 4.10 qui appartiennent également aux plans définis par les équations 4.2

$$\begin{cases} 51t_1 + t_2 + 51t_3 + offset = 534332 \\ 51t_1 + t_2 + 51t_3 + 1 + offset = 534332 \\ 51t_1 + t_2 + 51t_3 - 153 + offset = 534332 \\ 51t_1 + t_2 + 51t_3 - 152 + offset = 534332 \end{cases} \tag{4.2}$$

Ces plans sont représentés à la figure 4.11.

On peut définir la fréquence d'accès à une addresse A (considérée comme un

paramètre inconnu), comme étant le nombre de points entiers associés aux accès à cette adresse. C'est donc aussi le nombre de points entiers contenus dans une union paramétrée de polytopes, définie par intersection des polytopes initiaux et d'hyperplans : les polytopes de l'intersection sont ceux associés aux boucles, et les hyperplans sont déterminés par les équations indiquant que les polynômes des boucles les plus internes sont égaux à A.

Ce nombre de points entiers s'exprime sous la forme de plusieurs polynômes d'Ehrhart, correspondant à des intervalles disjoints de valeurs de A, et le calcul de ces polynômes d'Ehrhart se fait à l'aide du programme *barvinok* et de la librairie polyédrique *Polylib*.

Enfin, l'expression symbolique de la fréquence d'accès qui en résulte peut servir pour des analyses symboliques, ou pour générer plusieurs représentations visuelles des informations analysées, simplement en instanciant les paramètres.

Exemple 2 Pour connaître le nombre d'accès à un bloc mémoire B, on doit calculer le nombre de points entiers contenus dans l'union de polytopes paramétrés par B qui suit :

$$
\begin{aligned}
&\left\{ \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \le t_1 \le 99 \\ 0 \le t_2 \le 49 \\ 0 \le t_3 \le 2 \\ 0 \le t_4 \le 1 \\ 51t_1 + t_2 + 51t_3 + offset = B \\ B \ge 0 \end{array} \right\} \\
\cup &\left\{ \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \le t_1 \le 99 \\ 0 \le t_2 \le 49 \\ 0 \le t_3 \le 2 \\ t_4 = 2 \\ 51t_1 + t_2 + 51t_3 + 1 + offset = B \\ B \ge 0 \end{array} \right\} \\
\cup &\left\{ \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \le t_1 \le 99 \\ 0 \le t_2 \le 49 \\ 3 \le t_3 \le 5 \\ 0 \le t_4 \le 1 \\ 51t_1 + t_2 + 51t_3 - 153 + offset = B \\ B \ge 0 \end{array} \right\} \\
\cup &\left\{ \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \le t_1 \le 99 \\ 0 \le t_2 \le 49 \\ 3 \le t_3 \le 5 \\ t_4 = 2 \\ 51t_1 + t_2 + 51t_3 + 1 - 152 + offset = B \\ B \ge 0 \end{array} \right\}
\end{aligned}
$$

La solution est une liste de 16 polynômes d'Ehrhart définis sur 16 intervalles adjacents de valeurs de B. Notamment, tous les blocs B compris entre 530935 et 530983 sont référencés 18 fois, à l'exception des blocs pour lesquels $B \ mod \ 51 = 23$ qui eux sont référencés 6 fois, et des blocs tels que $B \ mod \ 51 = 24$ qui sont référencés à 12 reprises.

Grâce à ces polynômes, de nombreux graphiques peuvent donc être représentés, à l'instar des histogrammes de fréquences d'accès pour des intervalles quelconques de B (voir figure 4.12).

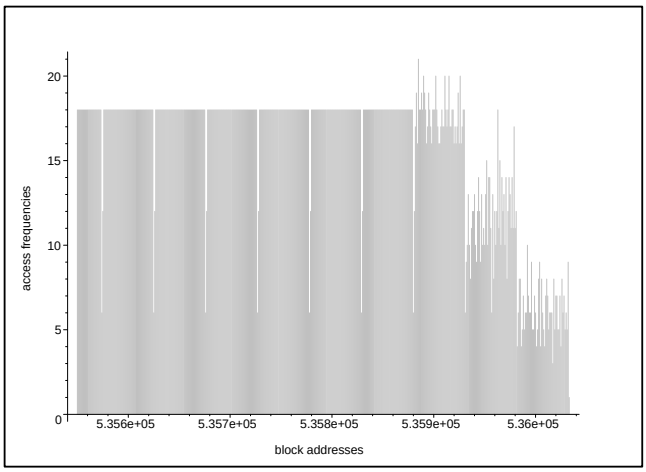


FIG. 4.12 – Fréquences d'accès aux blocs pour les blocs mémoire entre 535550 et 536033.

Analyse de la mémoire cache

Outre les adresses mémoire référencées par un programme, l'on dispose d'autres informations pour caractériser le comportement mémoire, telles que les défauts de cache générés par les accès. C'est ainsi qu'à la place de traces d'adresses mémoire, on peut analyser des traces de défauts de cache, contenant le nombre de défauts de cache identifiés après chaque accès mémoire. Il est intéressant de noter que les modèles générés par l'approche d'interpolation périodique-linéaire peuvent collaborer pour l'association des adresses mémoires et des défauts de cache correspondants. En effet, chacune des traces de départ contenant le même nombre d'éléments, les deux séquences de nids de boucles générées par le modèle périodique-linéaire représentent le même nombre d'itérations, et le i^{eme} accès mémoire correspond au i^{eme} nombre de défauts de cache. Le passage d'un modèle à l'autre est ensuite assuré par conversion des indices de boucles.

Pour générer une trace contenant le nombre de défauts de cache survenus après chaque accès mémoire, on utilise soit un simulateur de cache, soit les registres d'événements du processeur durant l'exécution du programme. L'avantage du simulateur est de pouvoir considérer plusieurs configurations de cache, et générer rapidement les traces correspondantes. Nous avons donc utilisé le simulateur de cache *DineroIV* [4] qui prend en entrée une trace d'adresses mémoire et permet de simuler plusieurs configurations habituelles de cache.

Ici encore, d'autres analyses peuvent être effectuées en instrumentant et en exécutant les boucles imbriquées obtenues par le modèle périodique pour générer de nouvelles traces. C'est ainsi que les boucles imbriquées issues de notre approche de modélisation permettent de construire un simulateur qui dispense de re-exécuter le programme analysé avec différentes instrumentations : toute information déduite des caractéristiques du comportement déjà modélisé est candidate pour une modélisation périodique-linéaire. A la suite de l'instrumentation des boucles imbriquées déjà générées, une exécution du programme résultant fournit une nouvelle trace à analyser, à la recherche de nouvelles informations.

Exemple 3 Nous avons simulé un cache à placement direct de $32KB$ contenant 1024 lignes et fonctionnant suivant la politique LRU, et nous avons utilisé *DineroIV* pour produire le nombre total de défauts de cache après chaque accès mémoire. La trace résultante est modélisée avec notre outil *PLI* sous la forme des boucles présentées à la figure 4.14, représentant la plus grande phase englobant 96% de tous les accès mémoire.

Une analyse attentive de ces boucles imbriquées montre que les accès mémoire générant des défauts de cache correspondent aux premières itérations des boucles les plus internes, puisque le nombre de défauts de cache reste constant à l'intérieur de ces boucles internes.

On représente graphiquement ces défauts de cache comme des polytopes dont les points entiers sont associés à des adresses mémoires successives, et dans lesquels les surfaces grisées (figure 4.13) représentent les accès provoquant des échecs de cache.

A partir des boucles obtenues, on peut également déterminer de manière précise quels sont les accès qui génèrent des défauts de cache (voir table 4.2), et déduire la distance entre deux adresses consécutives provoquant des échecs.

Par ailleurs, les adresses de blocs dont les références provoquent des échecs peuvent être identifiées en transformant les indices d'accès mémoire en indices des boucles de la figure 4.9, où chaque itération en t_1 en contient 900 autres (pour t_2, t_3, t_4), chaque itération en t_2 en contient 18 autres (pour t_3, t_4), etc. Soit un indice d'accès mémoire I, les indices (t_1, t_2, t_3, t_4) correspondant pour les boucles des adresses de blocs sont calculés de la manière suivante :

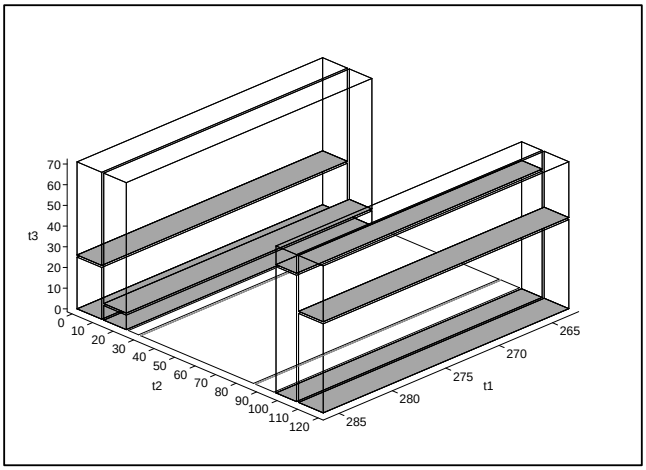


FIG. 4.13 – Polytopes représentant le comportement au niveau des défauts de cache.

$$I \longrightarrow \begin{cases} t_1 = \lfloor \frac{I}{900} \rfloor \\ t_2 = \lfloor \frac{I \ mod \ 900}{18} \rfloor \\ t_3 = \lfloor \frac{(I \ mod \ 900) \ mod \ 18}{6} \rfloor \\ t_4 = ((I \ mod \ 900) \ mod \ 18) \ mod \ 6 \end{cases}$$

Par exemple le 29186^{eme} accès mémoire, qui génère un défaut de cache comme il est mentionné à la table 4.2, est un accès au bloc mémoire dont les indices d'accès dans la boucle en figure 4.9 sont :

$$29186 \longrightarrow \begin{cases} t_1 = \lfloor \frac{29186}{900} \rfloor = 32 \\ t_2 = \lfloor \frac{29186 \ mod \ 900}{18} \rfloor = 21 \\ t_3 = \lfloor \frac{(29186 \ mod \ 900) \ mod \ 18}{6} \rfloor = 1 \\ t_4 = ((29186 \ mod \ 900) \ mod \ 18) \ mod \ 6 = 2 \end{cases}$$

ce qui correspond à un accès au bloc d'adresse :

$$51 \times 32 + 21 + 51 \times 1 + 1 + 530832 = 532537.$$

indices d'accès mémoire des défauts de cache	indices de boucles
$3600t_0 + 72t_1 - 943200$ $3600t_0 + 72t_1 - 943174$	$263 \leq t_0 \leq 286$ $0 \leq t_1 \leq 12$
$3600t_0 + 72t_1 - 943200$ $3600t_0 + 72t_1 - 943192$	$263 \leq t_0 \leq 286$ $13 \leq t_1 \leq 24$
$3600t_0 - 941400$ $3600t_0 - 941394$ $3600t_0 - 941338$	$263 \leq t_0 \leq 286$
$3600t_0 + 72t_1 - 948312$ $3600t_0 + 72t_1 - 948250$	$263 \leq t_0 \leq 286$ $97 \leq t_1 \leq 107$
$3600t_0 + 72t_1 - 948312$ $3600t_0 + 72t_1 - 948268$	$263 \leq t_0 \leq 286$ $108 \leq t_1 \leq 120$

TAB. 4.2 – Accès mémoires provoquant des défauts de cache.

Pour connaître le nombre de défauts de cache causés par une référence à un bloc mémoire B, une analyse directe des boucles ci-dessus et l'utilisation des formules de conversion entraînent des calculs et des réponses assez complexes, principalement à cause de leur non-linéarité.

Un autre moyen consiste alors à instrumenter les boucles en figure 4.14 pour construire un simulateur, et les exécuter pour générer une trace de toutes les adresses de blocs provoquant des défauts de cache. La modélisation de cette dernière trace sous forme de boucles permet ensuite de calculer le nombre de défauts de cache causés par la référence au bloc mémoire B.

Pour le simulateur, nous avons implémenté une fonction de conversion qui calcule à partir d'une valeur courante des indices (t_0, t_1, t_2, t_3), les adresses de blocs correspondantes. Cette fonction utilise la table 4.2, les formules donnant les indices correspondant aux boucles de la figure 4.9 et les fonctions d'adresses des boucles les plus internes de la même figure. La trace générée est modélisée par l'approche périodique-linéaire, qui fournit les boucles présentées à la figure 4.15.

Le nombre de défauts de cache provoqués par les références à un bloc mémoire B est donné par le polynôme d'Ehrhart de l'union des polytopes définis par les boucles imbriquées de la figure 4.15, limités aux points où les blocs accédés sont égaux à B. La solution donnée par *barvinok/Polylib* est une liste de 28 polynômes d'Ehrhart définis sur autant d'intervalles de valeurs de B. L'histogramme des fréquences de défauts de cache par adresses de blocs entre 531036 et 532000 se déduit de ces polynômes et est présenté à la figure 4.16.

```
for t0 = 263 to 286
{
  for t1 = 0 to 12
  {
    for t2 = 0 to 25
      #cache_misses = 51t0 + t1 + 77;
    for t2 = 26 to 71
      #cache_misses = 51t0 + t1 + 78;
  }
  for t1 = 13 to 24
  {
    for t2 = 0 to 7
      #cache_misses = 51t0 + t1 + 77;
    for t2 = 8 to 71
      #cache_misses = 51t0 + t1 + 78;
  }
  for t1 = 25 to 30
    #cache_misses = 51t0 + 102;
  for t1 = 31 to 86
    #cache_misses = 51t0 + 103;
  for t1 = 87 to 96
    #cache_misses = 51t0 + 104;
  for t1 = 97 to 107
  {
    for t2 = 0 to 61
      #cache_misses = 51t0 + t1 + 7;
    for t2 = 62 to 71
      #cache_misses = 51t0 + t1 + 8;
  }
  for t1 = 108 to 120
  {
    for t2 = 0 to 43
      #cache_misses = 51t0 + t1 + 7;
    for t2 = 44 to 71
      #cache_misses = 51t0 + t1 + 8;
  }
}
```

FIG. 4.14 – Boucles représentant l'accroissement du nombre de défauts de cache après chaque accès mémoire entre le 3601^{ieme} et le dernier.

```
for t0 = 0 to 23
{
  for t1 = 0 to 12
    for t2 = 0 to 1
      accessed_block = 204t0 + 4t1 + 53t2 + 531036;
  for t1 = 13 to 24
    for t2 = 0 to 1
      accessed_block = 204t0 + 4t1 + 52t2 + 531037;
  for t1 = 25 to 26
    accessed_block = 204t0 + 51t1 + 529863;
  for t1 = 27 to 38
    for t2 = 0 to 1
      accessed_block = 204t0 + 4t1 − 51t2 + 531085;
  for t1 = 39 to 50
    for t2 = 0 to 1
      accessed_block = 204t0 + 4t1 − 50t2 + 531085;
  for t1 = 51 to 51
    accessed_block = 204t0 + 531289;
}
```

FIG. 4.15 – Boucles représentant les adresses de blocs dont les accès provoquent des défauts de cache.

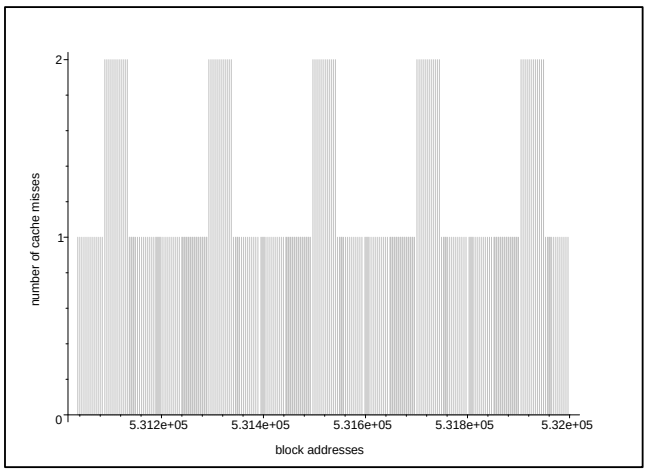


FIG. 4.16 – Fréquences des défauts de cache lors des références aux blocs compris entre 531036 and 532000.

Analyse de la distance de réutilisation des données

De la même manière que précédemment, on peut générer des traces pour déterminer les distances de réutilisation entre deux accès successifs au même bloc

mémoire. On construit un simulateur en instrumentant les boucles représentant les accès aux adresses mémoire et en les exécutant, on obtient une trace de toutes les distances de réutilisation. Cette trace est ensuite modélisée par notre approche périodique-linéaire. C'est ainsi que pour tout accès mémoire indicé par t, l'adresse de bloc correspondante et la distance de réutilisation s'obtiennent respectivement par conversion de t en indices des boucles représentant les accès aux blocs mémoire, et en indices des boucles représentant les distances de réutilisation.

4.6 Conclusion

A travers les différents types d'applications présentées ci-dessus, nous avons pu illustrer de manière concrète comment se fait l'utilisation du modèle périodique-linéaire dans l'analyse des traces de programmes. Lors des différentes expérimentations réalisées sur les programmes issus des bancs d'essai et en particulier sur ceux provenant des SPEC2000 [8], nous avons constaté qu'un nombre important d'accès via pointeurs se modélisent très simplement selon les différentes configurations de notre modèle, ce qui montre la pertinence du modèle périodique-linéaire pour la modélisation des accès mémoire.

En faisant intervenir le modèle polyédrique sur les boucles issues de notre modélisation, nous avons obtenu une représentation plus explicite encore du comportement du programme analysé, à savoir une représentation graphique.

Les nombreuses possibilités d'analyses offertes par le modèle polyédrique nous ont permis également d'étendre le champs d'applications de la modélisation périodique-linéaire à l'analyse de réutilisation de variables, l'analyse de la mémoire cache, etc. De cette manière a pu se développer la mise en oeuvre d'une collaboration fructieuse entre analyse statique et analyse dynamique.

Faisant suite à ce chapitre, nous présentons dans le chapitre suivant une vue d'ensemble du logiciel *PLI (Périodic-Linear Interpolation)* que nous avons implémenté au cours de ce travail, de même que les principales commandes d'utilisation correspondant aux différentes configurations du modèle périodique-linéaire.

Chapitre 5

Implémentation de PLI : Periodic-Linear Interpolation

Le logiciel *PLI* est écrit en langage C et contient les différentes fonctions d'analyse de données qui implémentent le modèle périodique-linéaire et ses extensions. Dans sa version actuelle il s'exécute en mode commandes sur les systèmes Unix et Linux, et prend en entrée un fichier de type ascii sur lequel il effectue les opérations indiquées par l'utilisateur, à travers des options. Le fichier de traces doit être constitué de valeurs entières, séparées par un ou plusieurs espaces. En fin d'analyse, le logiciel enregistre les résultats obtenus dans des fichiers de type ascii ou xml.

5.1 Fonctionnement

Dans sa forme la plus simplifiée, une commande se présente comme suit :

$./pli \quad -m$ modèle_recherché $\quad -i$ fichier_de_trace $\quad -o$ fichier_résultat

Le *modèle_recherché* correspond à l'une des configurations du modèle périodique-linéaire et peut prendre les valeurs suivantes :

- **af** lorsque la recherche porte sur des intervalles adjacents de taille fixe ;
- **av** pour les intervalles adjacents de taille variable ;
- **df** lorsqu'il s'agit de rechercher des intervalles distants de taille fixe ;
- **dv** pour les intervalles distants de taille variable.

L'identification de motifs répétés dont nous avons développé une première version en [25] se trouve assez facilement reproduite dans le cadre du présent modèle, car la recherche de motifs répétés s'effectue comme pour les configurations précédentes, avec les paramètres suivants :

- **ar** pour les motifs adjacents ;
- **dr** pour les motifs distants.

Exemple 5.1.1 *On crée le fichier de trace exemple1 suivant :*

```
3 3 7 13 11 23 15 33 19 43 23 53
```

On lance ensuite l'exécution de PLI sur ce fichier, et on édite le fichier résultat nommé result1 ainsi qu'il suit :

```
$ ./int/pli -m af -i exemple1 -o result1

 noise level=0.000000
 comp.time (CPU) 0.000000 seconds
 real 0.280000
 user 0.000000
 sys 0.000000

$ cat result1


sequence of 12 elements
(nbpi : number of points interpolated at least 3 times)

 phase : [0, 11]
(period = 2, 12 points, nbpi = 2/2, 0/10 errs or 0.000000 %)

 [4, 10] X +   [3, 3]     ( X = 0..5 )
```

Exemple 5.1.2 *On considère à présent la trace fir2dim_ blocs_ 32_ 100.txt de taille* $615Ko$*, extraite du programme* `fir2dim` *des bancs d'essai DSPStone [85] et disponible en [?]. On y recherche une interpolation périodique-linéaire suivant la configuration des intervalles adjacents de taille fixe et on obtient les résultats suivants :*

```
$./int/pli -m af -i fir2dim_blocs_32_100.txt -o resfir2


 noise level=0.000000
 comp.time (CPU) 760.700000 seconds
 real 960.250000
 user 0.970000
 sys 759.870000
```

```
$ cat resfir2


sequence of 90000 elements
 (nbpi : number of points interpolated at least 3 times)


 phase : [0, 89999] (period = 900, 90000 points, nbpi = 900/900,
                                  0/89100 errs soit 0.000000 %)
```

La fonction d'interpolation périodique ne peut être réprésentée ici à cause de la période élevée, mais elle est de la forme $[51]X + B$ *où* B *est un nombre périodique de* 900 *éléments.*

5.1.1 Autres options

Plusieurs options de la commande d'exécution permettent de réaliser les extensions du modèle périodique-linéaire au cas approximatif, ou encore au modèle de récurrences linéaires périodiques. Parmi ces options, on a :

1. $-e$ *pourcentage*
   où *pourcentage* est une valeur comprise entre 0 et 1, représentant le taux d'erreurs tolérées dans les intervalles, ou encore le pourcentage de points non interpolés pouvant être admis dans une phase ;
2. $-d$ *ordre_récursion*
   cette option est utlisée pour la recherche des récurrences linéaires-périodiques, et la valeur de *ordre_récursion* correspond à l'ordre de récursion souhaité. Si la trace considérée n'a pas suffisamment d'éléments pour l'ordre indiqué, on effectue tout de même l'analyse, mais selon l'ordre maximal possible ;
3. $-l$ *fichier_resultat*
   ici, on demande d'effectuer la génération automatique des boucles correspondantes au modèle analysé. Le résultat sera stocké dans deux fichiers de type ascii et xml respectivement, dont les noms sont indiqués par *fichier_resultat*. Le nom *fichier_resultat* doit être différent de celui qui est spécifié à la suite de $-o$;
4. $-s$
   cette option permet de lister les phases dans le fichier résultat par ordre de bornes de domaines croissants.

L'exemple suivant porte sur l'option $-d$ relative à l'extension *RPLI*, c'est à dire le modèle de récurrences périodiques-linéaires.

Exemple 5.1.3 *On crée un autre fichier nommé exemple2 dont le contenu est le suivant :*

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

On lance PLI à la recherche d'interpolations récurrentes périodiques-linéaire d'ordre 2 et on édite ensuite le fichier résultat spécifié lors de la commande, comme présenté ci-dessous ;

```
$ ./int/pli -m af -i exemple2 -o result2 -d 2

 noise level=0.000000
 comp.time (CPU) 0.000000 seconds
 real 0.040000
 user 0.000000
 sys 0.000000

$ cat result2


RPLI of order 2, sequence of 15 elements,
(nbpi : number of points interpolated at least 3 times)

phase : [0, 14]
(period = 1, depth = 2, 15 points, nbpi = 1/1,
                                  0/14 errs i.e. 0.000000 %)

  C == COEFFICIENTS MATRIX OF RECCURRENCE AS FOLLOWS :
           | 1 1 0 |
  U == INITIAL VALUES MATRIX FOR INDICES i=1..0 :
              U_1[i]
           |   1    |
           |   0    |
 CONSTANT  |   1    |

 for t1 = 2 .. 14
    for t2 = 1..1
        U_(t2)[(t1)] = SUM(j=0..1)[C_((t2)j)*U_(t2)[t1-j-1]]
                                                + C_((t2)2)
```

5.2 Les fonctions principales de *PLI*

Les traces à analyser sont constituées de valeurs entières. Comme il est possible que certaines de ces valeurs soient supérieures au plus grand entier représentable sur la machine utilisée, la plupart des fonctions de traitement sont implémentées dans *PLI* en deux versions : une version pour les entiers représentables et une deuxième version utilisant la librairie GMP [5].

Après l'interprétation de la commande utilisateur, la première fonction qui intervient est la lecture du fichier de trace.

5.2.1 Lecture de la trace

Le fichier d'entrée doit être constitué de valeurs entières, séparées par des espaces ; dans le cas contraire, la fonction de lecture signale l'erreur et met fin à l'exécution.

Par défaut, c'est la version pour entiers représentables qui est exécutée et dans le cas où l'on rencontre des entiers supérieurs à INT_MAX, le programme est interrompu et un message indique à l'utilisateur de le relancer avec les versions GMP (c'est à dire qu'il faut utiliser *./bigint/pli* au lieu de *./int/pli* pour introduire les commandes).

Les éléments de la trace sont intégralement stockés en mémoire dans un tableau de dimension 1.

5.2.2 Calcul d'autocorrélations

La fonction qui implémente le calcul des autocorrélations utilise la formule de l'autocorrélation simple (voir équations 2.13, 2.12), et se limite à l'ordre $k = n/3$, n étant la taille de la trace et les x_i étant ses éléments.

Cette fonction retourne la valeur de l'ordre d'autocorrélation maximale, et stocke dans un tableau les différentes valeurs d'autocorrélation calculées. Elle est appelée à partir de la fonction de recherche de phases présentée à la section suivante.

5.2.3 Recherche de phases

C'est ici que se traite principalement l'analyse de la trace ; la majorité des appels aux différentes autres fonctions de *PLI* y est effectuée.

Le premier appel se fait vers la fonction de calcul d'autocorrélations, et le tableau des coefficients obtenu en retour va être parcouru $n/3$ fois ; à chaque parcours,

la position du nouveau plus grand coefficient est retenue, car elle indique la période des intervalles à rechercher.

Chaque période testée est envoyée comme paramètre à une autre fonction implémentant l'un des algorithmes du modèle périodique-linéaire présentés au chapitre 3, en tenant compte de la configuration recherchée.

Chacune de ces fonctions effectue une recherche de phase et retourne la plus grande phase identifiée, pour la période indiquée.

A chaque retour d'une de ces fonctions, la valeur de la phase maximale est mise à jour en comparaison avec la phase précédente, et on obtient à la fin de tous les parcours la plus grande phase identifiée. Celle-ci est extraite de la trace et la fonction de recherche de phase recommence de manière récursive sur les éléments restants de la trace.

Remarque 5.2.1 *Si à une étape donnée de la récursion il n'y a pas de phase d'au moins trois intervalles interpolés, on lève la restriction sur le nombre d'intervalles minimum et on effectue une interpolation de période* $n/2$.

La fonction fait ensuite appel à d'autres fonctions d'écriture qui vont enregistrer les résultats dans le fichier de sortie ; ces fonctions d'écriture reçoivent comme paramètres la période et la configuration des intervalles identifiés.

5.3 Génération automatique de boucles

Le résultat d'une exécution simplifiée de *PLI*, c'est à dire limitée au premier niveau hiérarchique $i = 0$, comporte des phases pouvant être décomposées en sous-phases. Pour atteindre le niveau le plus bas et donc obtenir les boucles imbriquées représentant la trace étudiée, les différentes sous-phases doivent elles-mêmes servir d'entrée à *PLI* au cours de nouvelles exécutions. Ces itérations manuelles de l'exécution pouvant s'avérer fastidieuses, nous avons implémenté pour les éviter des fonctions de génération automatique des boucles finales, accessibles via l'option $-l$ de la commande d'exécution. Les principales d'entre elles sont présentées ci-dessous.

5.3.1 Développement des sous-phases

Cette première fonction utilise le fichier résultat du premier niveau hiérarchique $i = 0$, lequel contient toutes les phases identifiées pour la trace.

La tâche à effectuer (que nous appellerons T) consiste à exécuter *PLI* sur toutes les sous-phases de ce premier fichier, ce qui devrait résulter en la création de $2m_0$ nouveaux fichiers de phases (ou moins), m_0 étant le nombre de phases du fichier de niveau $i = 0$. Par la suite T doit être appelée récursivement sur ces $2m_0$ nouveaux fichiers, ce qui entraîne la création de $2m_0 * 2m_1$ autres fichiers, et ainsi de suite (voir figure 5.1).

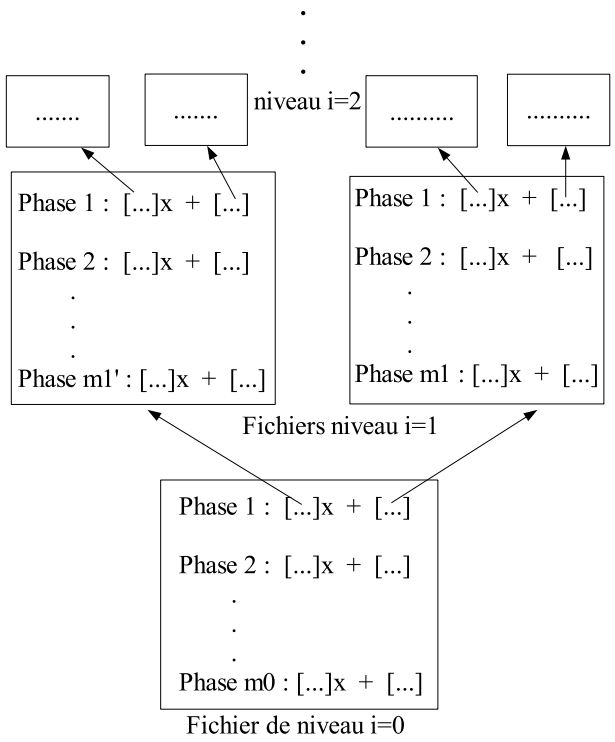


FIG. 5.1 – Croissance du nombres de fichiers pour l'exécution récursive de T

Afin d'éviter cette trop grande croissance des fichiers intermédiaires, nous choisissons de n'en créer que deux à chaque nouvelle étape i : toutes les sous-phases résultantes des premiers coefficients périodiques (coefficients en x) sont rassemblées en un seul fichier, de même pour celles qui résultent des seconds coefficients périodiques. Par exemple, pour un fichier *résultat* de niveau $i = 0$, on va créer deux fichiers nommés *resultata* et *resultatb*.

On obtient ainsi une arborescence binaire de fichiers comme cela est illustré à la figure 5.2.

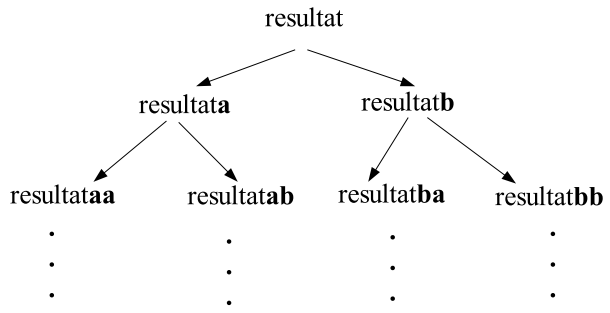


FIG. 5.2 – Arborescence de fichiers issus de T

Exemple 5.3.1 *On considère un fichier exemple3 dont le contenu est le suivant :*

```
3 4 5 1 7 14 21 8 11 24 37 15 2 5 7 9 3 5 9 11 13 4 8 13 15 17
5 11 17 19 21 6
```

On exécute PLI avec l'option -l et parmi les fichiers obtenus, on a le fichier result3 suivant, qui correspond au niveau $i = 0$:

```
sequence of 32 elements
(nbpi : number of points interpolated at least 3 times)

 phase : [0, 11] (period = 4, 12 points, nbpi = 4/4,
                                        0/8 errs soit 0.000000 %)
 [4, 10, 16, 7] X +   [3, 4, 5, 1]     ( X = 0..2 )

 phase : [12, 31] (period = 5, 20 points, nbpi = 5/5,
                                        0/15 errs soit 0.000000 %)
 [3, 4, 4, 4, 1] X +  [2, 5, 7, 9, 3]     ( X = 0..3 )
```

On obtient également un fichier nommé result3a, contenant les résultats d'interpolation périodique-linéaire de tous les premiers coeffcients périodiques de result3 :

```
//$  4 10 16 7  £
 phase : [0, 2] (period = 1, 3 points, nbpi = 1/1,
                             0/2 errs soit 0.000000 %)
 [6] X + [4]  (X = 0..2)
```

```
 phase : [3, 3] (période= 1, 1 point)
 [0]X + [7], (X = 0..0)

//$  3 4 4 4 1  £
 phase : [0, 0] (période= 1, 1 point)
 [0]X + [3], (X = 0..0)

 phase : [1, 3] (period = 1, 3 points, nbpi = 1/1,
                          0/2 errs soit 0.000000 %)
 [0] X +  [4]  (X = 0..2)

 phase : [4, 4] (période= 1, 1 point)
 [0]X + [1], (X = 0..0)
```

On a également un fichier nommé result3b, contenant les résultats d'interpolation périodique-linéaire de tous les seconds coeffcients périodiques de result3 :

```
( //$  3 4 5 1  £
 phase : [0, 2] (period = 1, 3 points, nbpi = 1/1,
                              0/2 errs soit 0.000000 %)
 [1] X +  [3]  (X = 0..2)

 phase : [3, 3] (période= 1, 1 point)
  [0]X + [1], (X = 0..0)

//$  2 5 7 9 3  £
 phase : [0, 0] (période= 1, 1 point)
  [0]X + [2], (X = 0..0)

 phase : [1, 3] (period = 1, 3 points, nbpi = 1/1,
                              0/2 errs soit 0.000000 %)
 [2] X +  [5]   (X = 0..2)
```

Ce choix de ne créer que deux fichiers à chaque niveau i impose donc de traiter la totalité des phases de même niveau avant de descendre dans la hiérarchie, ce qui constitue un obstacle à un traitement parallèle qui pourrait se baser sur le traitement complet d'une seule phase à la fois, jusqu'au niveau hiérarchique le plus élevé.

Cependant, notre choix se justifie à nouveau, dans la mesure où la tâche T est effectuée de manière récursive : le traitement complet d'une seule phase à la fois

demanderait de connaître **en totalité** les différents fichiers issus de l'interpolation des coefficients périodiques, et cela contredit le fait de ne traiter qu'une seule phase à la fois.

La tâche T qui vient d'être détaillée est confiée à une autre fonction ; au retour de cette fonction on dispose, en plus de l'arborescence des fichiers créés, d'une liste chaînée contenant les bornes de l'ensemble des phases et sous-phases correspondant à la trace de départ.

Ayant donc la possibilité d'identifier dans cette liste la sous-liste de chaque phase identifiée au niveau $i = 0$, on appelle l'une des fonctions qui implémentent l'union des phases selon la méthode présentée au chapitre 3, et ce pour chacune de ces sous-listes.

5.4 Conclusion et développements ultérieurs

Le développement du logiciel *PLI* s'est effectué tout au long de la construction du modèle périodique-linéaire, et l'outil s'est vu considérablement restructuré en fonction de la progression de notre travail. C'est l'une des raisons pour lesquelles il comporte un grand nombre de fonctions, par souci de modularité. Cela explique aussi le fait que certaines parties aient quelquefois requis une quantité de travail non négligeable, pour ensuite devoir être abandonnées parce que l'idée motrice l'était aussi.

D'autres fonctions encore comme celles relatives aux extensions du modèle ont été écrites dans une version simplifiée, et sont donc susceptibles d'être davantage approfondies.

Dans l'ensemble, nous envisageons un certain nombre de perspectives pour des développements ultérieurs et l'évolution du logiciel.

5.4.1 La parallélisation des fonctions

Plusieurs traitements au cours de l'exécution peuvent être parallélisés, comme la recherche de phases suivant des périodes différentes. Sur une architecture multiprocesseurs, la recherche selon une période donnée peut très bien être confiée à un processeur, ce qui réduirait le temps de calcul. Au niveau du calcul des boucles finales correspondant à la trace d'entrée, certains traitements peuvent aussi se faire de manière indépendante.

5.4.2 La gestion des données en mémoire

Pour l'instant les traces sont intégralement gérées en mémoire et stockées dans un tableau. Nous envisageons un autre mode de stockage à la place, comme par exemple une base de données et un gestionnaire adéquat permetant d'y accéder le plus rapidement possible et de manière optimisée. On aurait ainsi l'avantage de pouvoir traiter des traces de taille encore plus élevée.

Par ailleurs, la plupart des données intermédiaires obtenues lors de la génération automatique de boucles imbriquées sont conservées dans des structures allouées dynamiquement ; pour l'instant, on n'est pas encore en mesure de les libérer rapidement, car toutes les informations qu'elles contiennent sont nécessaires pour l'obtention du résultat final. Cela pourrait provoquer dans certains cas une interruption brutale du programme, s'il n'y a pas suffisamment d'espace de stokage sur la machine utilisée. Cette remarque est valable pour les différents fichiers créés au cours de l'exécution de *PLI*, car leur nombre et leur taille dépendent des traces analysées.

5.4.3 Développement de nouvelles stratégies

Nous avons mentionné le fait que la stratégie de modélisation adoptée dans *PLI* est une stratégie descendante, et qu'elle entraîne plusieurs parcours de la trace pour identifier la meilleure période d'interpolation.

Dans une approche ascendante, on n'aurait pas besoin d'identifier cette période à priori, et on effectuerait une interpolation linéaire non périodique pour compresser la trace. Le résultat de cette première interpolation serait interpolé à son tour, et ainsi de suite jusqu'à l'identification des phases et périodes finales.

Bien que moins englobante ou moins générale que la stratégie descendante, cette approche pourrait donner des résultats intéressants dans certains cas (très grande trace, très grande régularité des éléments, ...) et constituer une alternative à la précédente.

5.4.4 Développement d'un environnement d'analyse

Au niveau des applications de *PLI*, nous avons montré que l'utilisation du modèle périodique-linéaire en collaboration avec le modèle polyédrique est assez fructueuse. Pour faciliter l'exploitation conjointe de ces modèles, il est souhaitable de développer un environnement d'analyse intégrant les différents outils, à savoir :

- *PLI :* notre outil d'interpolation périodique-linéaire.
- *Polylib [6] :* la librairie implémentant plusieurs fonctions pour la manipulation de polyèdres, paramétrés ou non, dans l'espace des nombres rationnels.

– *barvinok [82, 83] :* le programme de calcul du nombre de points entiers d'un polyèdre paramétré, i.e le polynôme d'Ehrhart [26] de ce polyèdre.

Le schéma général de cet environnement est celui présenté au chapitre 4 et rappelé à la figure 5.3, et se trouve actuellemment en cours d'implémentation.

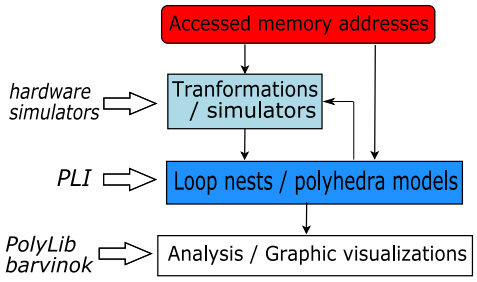


FIG. 5.3 – Outils composant l'environnement d'analyse.

D'autres perspectives plus générales concernant la modélisation périodique-linéaire et la suite de nos travaux font l'objet du chapitre suivant.

Chapitre 6

Conclusion

La compréhension et la modélisation du comportement des programmes est de plus en plus reconnue parmi les chercheurs comme une réponse nécessaire aux besoins de plus en plus forts de maîtrise de ce comportement. Pour mener à bien cette analyse, l'approche qui nous a intéressés dans ce travail se fonde sur une étude des traces de programmes, recueillies lors de leur exécution.

Dans cette thèse, nous nous sommes donc intéressés à l'analyse et la modélisation de traces issues de l'instrumentation de codes et constituées par une suite chronologique de valeurs entières. Pour parvenir à la compréhension et la maîtrise du comportement des programmes, nous nous sommes défini comme objectif l'obtention d'un modèle de représentation du profil d'exécution initial, susceptible en outre de se prêter à diverses analyses dans le but d'opérer des transformations ou des optimisations de code.

En assimilant nos traces de valeurs entières à des séries temporelles ou plus globalement encore à des informations ou données, nous avons exploré dans un premier temps des méthodes générales d'analyse assez bien connues, telles que la fouille de données (*ang. Data Mining*) et l'analyse des séries temporelles. Cette investigation nous a amenés à constater que les informations obtenues en fin d'analyse restaient assez générales, éloignées des mécanismes spécifiques de la programmation des ordinateurs, et donc difficilement exploitables, par rapport à nos objectifs.

C'est ainsi que nous avons recherché un autre type de modélisation plus spécifique, et proposé une approche basée sur une méthode d'interpolation périodique-linéaire, permettant d'exprimer le comportement à l'exécution d'un programme par un autre programme.

Apports et résultats

Nous exprimons le comportement d'un programme initial par un nouveau programme, nous focalisant ainsi sur la manière ou le "comment" du programme, plutôt qu'à sa fonctionnalité.

Le programme modèle obtenu est constitué d'une séquence de nids de boucles dans lesquelles les instructions des boucles les plus internes sont des fonctions polynomiales, les bornes des boucles sont soit des constantes, soit des fonctions affines des indices de boucles, et les fonctions de niveau les plus internes expriment les valeurs de la trace d'entrée à partir des indices de boucles.

L'approche développée représente donc bien une trace d'entrée, et ceci par une séquence de nids de boucles ; chaque séquence de nids de boucles correspond à une *phase* du programme.

Le modèle proposé, appelé *modèle périodique-linéaire*, comprend plusieurs configurations qui permettent de mettre en exergue différentes variantes du comportement des programmes : comportement unique, comportements entrelacés, comportement répétitif, comportements de durée variable, etc ; ce modèle est flexible en ceci qu'il conserve sa pertinence même lorsque des valeurs *perturbatrices* surviennent dans la trace, et cela est rendu possible par des extensions que nous avons effectuées.

Une autre extension permet d'ailleurs d'élargir le champs des comportements identifiables, en utilisant cette fois des fonctions de représentation sous forme de récurrences périodiques-linéaires.

Les différents algorithmes du modèle périodique-linéaire ont été entièrement implémentés et testés, et certaines parties du travail ont donné lieu à trois publications internationales [23, 24, 25].

Notre modèle se prête aisément aux analyses, comme nous l'avons illustré à l'aide de quelques applications issues de bancs d'essai (DSPStone, SPECINT2000, Olden) : l'étude du programme modèle obtenu a permis d'effectuer des tranformations sur les codes de départ et a également fourni des informations utiles, permettant d'aboutir à des optimisations.

Nous avons aussi montré que grâce à cette représentation, un plus grand nombre d'analyses sont possibles, et ce avec plusieurs outils qui s'y prêtent de manière adéquate, à l'exemple du modèle polyédrique d'analyse statique.

En appliquant ce modèle qui relève de l'analyse statique à l'analyse de notre modèle, nous avons pour ainsi dire mis en œuvre une certaine collaboration entre les méthodes d'analyse statique et dynamique.

Il n'en demeure pas moins que le modèle périodique-linéaire peut encore être enrichi grâce à diverses améliorations et extensions.

Limites du modèle et perspectives

Une des principales limites actuelles de notre outil de modélisation concerne la taille des traces pouvant être traitées et les temps de traitement. Nous avons pour cela proposé dans le chapitre précédent quelques voies d'amélioration : parallélisation des programmes, conception d'autres algorithmes, utilisation d'une base de données.

Parmi les perspectives d'extensions envisagées, nous pouvons mentionner l'utilisation de fonctions d'interpolation autres que linéaires ou même polynomiales, afin d'élargir encore davantage l'éventail de données pour lesquelles le modèle soit représentatif. On pourrait par exemple avoir une fonction de transition quelconque qui permette de mettre en exergue les variations de comportement du programme.

En outre, le modèle doit être appliqué à d'autres caractéristiques du comportement des programmes, telles que les branchements, les séquences d'instructions, les séquences d'appels de fonction, les valeurs de variables, la consommation électrique. Pour ce dernier cas en particulier, il est nécessaire d'acquérir auparavant le matériel requis pour effectuer des mesures à des fréquences régulières.

Comme autre perspective, on peut considérer la représentation des traces par des structures de contrôle différentes des boucles imbriquées, par exemple les structures de type *if-then-else* ou encore de type *while*, ce qui permettrait d'obtenir un modèle plus dynamique et dépendant des valeurs traitées.

Enfin, il serait intéressant de sortir le modèle périodique-linéaire de son cadre actuel pour l'utiliser plus globalement dans l'analyse de données issues d'autres domaines, en commençant par exemple par les domaines d'applications classiques de la fouille de données telle que l'analyse biologique.

Bibliographie

[1] http://www.intel.com/cd/software/products/asmo-na/eng/vtune/.

[2] http://www.hpl.hp.com/research/linux/perfmon/.

[3] Time Series Processor, version 4.4. User's guide and Reference Manual. TSP International, http://www.tspintl.com.

[4] Dinero IV : Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV.

[5] The GNU Multiple Precision Arithmetic Library. http://www.swox.com/gmp/.

[6] The Polyhedral Library. *polyLib* : http://icps.u-strasbg.fr/polylib/.

[7] Quelques traces de programmes. http://polytope.u-strasbg.fr/traces/.

[8] SPEC CINT2000 Benchmarks (Integer Component of SPEC CPU2000). http://www.spec.org/cpu/CINT2000/.

[9] G. Adelson-Velskii and E. M. Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk SSSR*, (146) :263–266, 1962.

[10] P. Adriaans and D. Zantinge. *Data Mining*. Addison-Wesley, 1996.

[11] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. In *SIGMOD '93 : Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, 1993. ACM Press.

[12] T. Austin. The Pointer-Intensive Benchmark Suite, version 1.1, September 1995. http://www.cs.wisc.edu/~austin/ptr-dist.html.

[13] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4) :1319–1360, July 1994.

[14] R. Berrerndorf, H. Ziegler, and B. Mohr. Performance Counter Library. Technical report, Central Institute for Applied Mathematics at the Research Centre Juelich, July 1999.

[15] G. E. P. Box and G. M. Jenkins. *Time Series Analysis. Forecasting and Control*. Holden Day, 1976.

[16] G. E. P. Box and D. A. Pierce. Distribution of Residual Correlations in Autoregressive-Integrated Moving Average Time Series Models. *Journal of the American Statistical Association 65*, pages 1509–1526, 1970.

[17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, CA, 1984.

[18] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 132, University of Wisconsin-Madison Computer Sci-ences, June 1997.

[19] M. C. Carlisle and A. Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the of the 5fh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[20] C. Chatfield. *The Analysis of Time Series : An Introduction.* Chapman and Hall, New York, fourth edition, 1989.

[21] I. Chihaia and T. Gross. Memperf : Effectiveness of Simple Memory Models for Performance Prediction. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04),*, 10-12 March 2004.

[22] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic Discovery of Time Series Motifs. In *KDD '03 : Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–498, New York, 2003. ACM Press.

[23] P. Clauss and B. Kenmei. Polyhedral Modeling and Analysis of Memory Access Profiles. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors, ASAP 2006*, Steamboat Springs, Colorado, September 2006. IEEE Computer Society.

[24] P. Clauss, B. Kenmei, and J. C. Beyler. The Periodic-Linear Model of Program Behavior Capture. In *Euro-Par 2005*, volume 3648 of *LNCS*, pages 325–335, Lisboa, September 2005. Springer.

[25] P. Clauss, B. Kenmei, and O. Rousselet. Modeling Program Tracing with Nested Loops. *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques, held in conjunction with ICS'03*, June 2003.

[26] P. Clauss and V. Loechner. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI Signal Process. Syst.*, 19(2) :179–194, 1998.

[27] T. M. Conte. *Fast Simulation of Computer Architectures.* Kluwer Academic Publishers, Norwell, 1995.

[28] C. Cépaduès. *Visualisation en Extraction des Connaissances.* RNTI-E-7, 2006. Rédacteurs invités : François Poulet, Pascale Kuntz.

[29] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP : Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[30] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. *SIGARCH Comput. Archit. News*, 30(2) :233–244, 2002.

[31] A. S. Dhodapkar and J. E. Smith. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society Washington, DC, 2003.

[32] D. A. Dickey and W. A. Fuller. Likelihood Ratio Statistics for Autoregressive Time Series with a Unit Root. *Econometrica Communications of the ACM*, 49(4), 1981.

[33] E. Diday, J. Lemaire, J. Pouget, and F. Testu. Eléments d'Analyse des Données. *Dunod-Bordas*, 1982.

[34] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *12th International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, September 2003.

[35] U. M. Fayyad, G. Piatetsky, and P. Smyth. From Data Mining to Knowledge Discovery in Databases. *Artificial Intelligence Magazine*, 17 :37–54, 1996.

[36] P. Feautrier. Automatic Parallelization in the Polytope Model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103. Springer, 1996.

[37] W. Fuller. Introduction to Statistical Times Series. *John Wesley*, 1976.

[38] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can Shared-Memory Model Serve as a Bridging Model for Parallel Computation ? In *ACM Symposium on Parallel Algorithms and Architectures*, pages 72–83, 1997.

[39] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM Model : Accounting for Contention in Parallel Algorithms. *SIAM Journal on Computing*, 28(2) :733–769, 1999.

[40] S. Graham, P. Kessler, and M. McKusick. gprof - a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN'82*, volume 17, pages 20–126. Symposium on Compiler Construction, June 1982.

[41] G. Hamerly, E. Perelman, and B. Calder. How to use SimPoint to pick Simulation Points. *SIGMETRICS Perform. Eval. Rev.*, 31(4) :25–30, 2004.

[42] J. D. Hamilton. *Time Series Analysis*. University Press, 1994.

[43] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase Shift Detection : A Problem Classification. Research report RC-22887, IBM, August 2003.

[44] HP. Trimaran : An Infrastructure for Research in Instruction-Level Parallelism. Technical report, Hewlett Packard, University of Illinois, New York University, July 1999.

[45] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, 1988.

[46] A. K. Jain, M. N. Murty, and P. J. Flynn. Data Clustering : A Review. *ACM Comput. Surv.*, 31(3) :264–323, 1999.

[47] R. Jin and G. Agrawal. Performance Prediction for Random Write Reductions : A Case Study in Modeling Shared Memory Programs. In *ACM SIGMETRICS intern.conf. on Measur. and Model. of computer systems*, pages 117–128, 2002.

[48] B. R. Kenmei. Vers un Profiling Adaptatif : Etat de l'art des Méthodes Actuelles d'Analyse de Prgrammes et Perspectives, November 2002. Université de Yaoundé I.

[49] A. Kertterlin. *Découverte de Concepts Structurés dans les Bases de Données.* PhD thesis, Université Louis Pateur, 1995.

[50] D. Knuth. *The Art of Computer Programming*, volume 3 : Sorting and Searching. Addison-Wesley, 3 edition, 1997. ISBN 0-201-89685-0, Pages 458-475 de la section 6.2.3 : Balanced Trees.

[51] Y. Kodratoff. Applications de l'apprentissage automatique et de la fouille de données. In *EGC'01*, Nantes France, Janvier 2001.

[52] J. Korczak and Z. Kuznicki. Apprentissage et Découverte de Connaissances dans des Bases de Donnees. Technical Report LSIIT 96/14, ULP, Strasbourg, 1996.

[53] J. KORCZAK and A. QUIRIN. Evolutionary Approach to Discovery of Classification Rules from Remote Sensing Images. In *5th European Workshop on Evolutionary Computation in Image Analysis and Signal Processing (EvoIASP2003)*, pages 388–398, Essex, 2003. Essex.

[54] N. LACHICHE. *Abduction and induction from the non-monotonic logic perspective*, chapter ;, pages 107–116. P Flach and A. Kakas, eds. - Kluwer, 2000. Abductive and Inductive Reasoning : Essays on their Relation and Integration.

[55] J. R. Larus. Abstract Execution : A Technique for Efficiently Tracing Programs. *Softw. Pract. Exper.*, 20(12) :1241–1258, 1990.

[56] J. R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5) :52–61, 1993.

[57] J. R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN*, pages 259–269. ACM Press, 1999.

[58] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, March 2005.

[59] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In *DMKD '03 : Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11, New York, 2003. ACM Press.

[60] G. M. Ljung and G. E. P. Box. On a Measure of Lack of Fit in Time Series Models. *Biometrika 65*, pages 297–303, 1978.

[61] B. Meister. Using Periodics in Integer Polyhedral Problems. Technical Report 03-01, LSIIT - ICPS UMR7005 ULP-CNRS, 2003.

[62] T. M. Mitchell. *Machine Learning.* 1997.

[63] J. L. Nading and G. D. Ripley. Instrumentation and Measurement of High-Level Programs. Research report, University of Arizona, Computer Science Department, October 1973.

[64] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.

[65] C. G. Nevill-Manning and I. H. Witten. Linear-time, Incremental Hierarchy Inference for Compression. In *Data Compression Conference*, pages 3–11, 1997.

[66] D. Ofelt and J. L. Hennessy. Efficient Performance Prediction for Modern Microprocessors. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 229 – 239. ACM Press, 2000.

[67] V. Pai, P. Ranganathan, and S. Adve. RSIM : An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1997.

[68] D. Pelleg and A. Moore. X-means : Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, San Francisco, 2000. Morgan Kaufmann.

[69] J. M. Queen. "some methods for classifications and analysis of multivariate observations". In *in proc. of the Fifth Berkeley Symposium on Mathematical statistics and probability*, volume 1, pages 281–297. University of California Press, August 1967.

[70] M. C. C. A. Rogers, J. H. Reppy, and L. J. Hendren. Early Experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20, London, 1994. Springer-Verlag.

[71] T. Sherwood and B. Calder. Time Varying Behavior of Programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[72] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *In the proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[73] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *SIGPLAN Not.*, 37(10) :45–57, 2002.

[74] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[75] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *the proceedings of the 30th Annual Intl. Symposium on Computer Architecture (ISCA 2003)*, June 2003.

[76] D. Sorin, V. Pai, S. Adve, M. Vernon, and D. Wood. Analytic Evaluation of Shared-Memory Systems with ILP Processors. In *International Symposium on Computer Architecture (ISCA)*, Barcelona, June 1998.

[77] N. N. Tran. *Automatic Arima Time Series Modeling And Forecasting Adaptive Input/Output Prefetching.* PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[78] S. Udayakumaran and R. Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad based Embedded Systems. In *CASES '03 : Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, New York, 2003. ACM Press.

[79] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation : A survey. *ACM Computing Surveys*, 29(2) :128–170, 1997.

[80] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8) :103–111, 1990.

[81] E. van der Deijl, G. Kanbier, O. Temam, and E. D. Granston. A Cache Visualization Tool. *IEEE Computer*, 30(7) :71–78, 1997.

[82] S. Verdoolaege. Barvinok : A Library for Counting the Number of Integer Points in Parametric and non-parametric Polytopes. `http://freshmeat.net/projects/barvinok/`.

[83] S. Verdoolaege, K. M. Woods, M. Bruynooghe, and R. Cools. Computation and Manipulation of Eumerators of Integer Projections of Parametric Polytopes. Report CW 392, Katholieke Universiteit Leuven, 2005.

[84] C. Wemmert. *Classification Hybride Distribuée par Collaboration de Méthodes non Supervisées.* PhD thesis, Université Louis Pateur, 2000.

[85] V. Zivojnovic, M. Velarde, and C. Schlager. DSPstone : A Dsp-oriented benchmarking methodology. October 1994.

Résumé

Cette thèse présente une nouvelle technique de représentation et d'analyse de traces d'exécution de programmes. Après une étude de quelques méthodes générales d'analyse de données, l'accent est mis sur une nouvelle modélisation qui consiste à exprimer le comportement à l'exécution d'un programme par un autre programme.

Le programme qui exprime le comportement est constitué d'une séquence de nids de boucles dans lesquelles les fonctions de niveau les plus internes expriment les valeurs de la trace d'entrée à partir des indices de boucles. Chaque séquence de nids de boucles correspond à une définition particulière d'une *phase* de programme, c'est à dire un ensemble d'intervalles dépendants les uns des autres, et ces intervalles sont identifiés grâce à une méthode d'interpolation périodique et linéaire. A partir de cette représentation, on montre qu'on est en mesure d'effectuer un assez grand nombre d'analyses et d'optimisations, en s'aidant notamment des outils comme le modèle polyédrique d'analyse statique. Au final, on vérifie bien que les informations obtenues suite à notre modélisation et à l'analyse statique des nids de boucles résultants sont plus précises, mieux adaptées, et plus exploitables qu'avec des méthodes générales d'analyse de données.

Abstract

This work presents a novel approach for program execution traces analysis and modeling. After a short introduction where it is shown that general methods for data analysis are not well-adapted to understand the behavior of programs, we focus on the design of our model, called the *Periodic Linear Model*. It is based on the represensation of a program behavior by another program.

The program that models the behavior of the first one is made of sequences of nested loops, in which the functions of the innermost level compute the values of the input program trace from the loop indices. Each sequence of nested loops correspond to a particular definition of a program *phase*, that is a set of linearly linked intervals ; these intervals are identifed through our periodic linear interpolation method.

From this representation, we show that many analysis and optimizations are possible in conjunction with some static analysis models like the polytope model, and also that the information resulting from our model are more precise, adapted and exploitable than those obtained by general data analysis methods.